Analysis of Software Test Item Generation
— Comparison between High Skilled and Low Skilled Engineers —

Masayuki HIRAYAMA[†], Tetsuya YAMAMOTO[†], Osamu MIZUNO[‡], and Tohru KIKUNO[‡]
[†] TOSHIBA Corporation, Japan. (E-mail: masayuki.hirayama@toshiba.co.jp)
[‡] Graduate School of Information Science and Technology, Osaka University, Japan
(E-mail: {o-mizuno,kikuno}@ist.osaka-u.ac.jp)

Abstract

Recent software system contains a lot of functions to provide various services. According to this tendency, it is difficult to ensure software quality and to eliminate crucial faults by conventional software testing method. Especially, in the conventional method, detail level of test items are widely deflected according with the engineers' skill, and this causes an immature software quality. In this paper, we discuss the effects of test engineer's skill on test item generation, and propose a new test item generation method, that enables the generation of test items for illegal behavior of the system. The proposed method can generate test items based on Use-case analysis, deviation analysis for legal behavior, and faults tree analysis for system fault situations. From the result of the experimental applications, we confirmed that test items for illegal behavior of the system were effectively generated, and also the proposed method could effectively assist test item generation of poor skill engineer.

Keywords: *software testing, deviation analysis, engineer's skill*

1 Introduction

Software testing is generally divided into three phases: unit testing, integrate testing, and system testing. Among them, the system testing checks functional behavior of the target software system [1, 7], and thus it is directly related to the users' satisfaction. The growing complexity of software structure coupled with users' desire for high quality has greatly increased demand for efficient system testing.

In order to execute the system testing, test cases or test items are required. Conventionally, test items are generated from software specification documents [5]. A test engineers checks the functions described in the specification documents, and derives test items. Test items enable detailed confirmation of each functional behavior specified in the specification document, and so a large number of test items are required.

Granularity of test items depends on the granularity of functional specifications. Also, though test items are executed in the order of the test specification, test details for each function are largely dependent on the skill or experience of the test engineer.

In order to establish highly effective testing, we have developed a new testing method, which reduces the effects of each test engineer's skill on generating test items. In this paper, we address the characteristics of software testing in industries. Then, we introduce a new method to generate test items based on the deviation analysis. In the experimental evaluation, we firstly show the analysis results of the differences between the test items generated by the engineer with high skill and the test items generated by the engineer with low skill. We then present the difference of two test items, generated by the proposed method and generated by above mentioned engineers.

2 Characteristics of Software Testing in Industries

Here, let us consider control software of electrical appliance as a typical example. If a software fault remains in an electrical appliance, and as a result, the appliance suddenly fails to work correctly, the impact may be grave. Therefore, it is becoming essential to ensure the reliability of embedded software in electrical appliances [8].

In the past, since software embedded in an electrical appliance tended to be relatively small in size, the formal method based on the finite state machine was used effectively in the development [8], and the test items were generated using transition sequences on the finite state machine.

Due to the growing sophistication of electrical appliances, the size of the software they contain has become large. So, applying the formal method for this large software requires large effort and much time. As a result, the formal approach is not effective in the development of many recent products, since it is applicable only to the core portion of the software in those products.

So, currently, in the software development field, most software in products is tested by the conventional method. In the conventional method, a test engineer checks the requirements for each function described in the specification document, and manually derives each functional requirement description as a test item. Then in the test phase, the test engineer executes generated test items in the order of the test specification documents. However, the conventional

method has a few problems as follows:

(1) Software specification documents (without using the formal method), the origin of the test specification, may contain ambiguities. These ambiguities of specification may cause granularity gaps in test items.

(2) Generally, a software specification document mainly describes legal behaviors. So, it is difficult to generate the test items for illegal behaviors.

(3) Many companies do not prepare methods for generating test items. This causes the difference in test item generation between a skilled engineer and a less skilled engineer. That is, generation of test items is highly dependent on the skill of the test engineer. A highly skilled engineer can generate various test items including the test items for illegal behaviors. On the contrary, a less skilled engineer tends to only generate test items for legal behaviors.

3 A New Method for Test Item Generation

3.1 Overview

As indicated above, software testing in current software development is subject to several problems which need to be resolved. From a detailed consideration of these problems, it is concluded that one of the most important requirements for software testing is to establish an effective testing method capable of detecting software faults effectively. In the software industry, there is a great demand for an effective testing method for embedded software.

In order to detect important faults effectively, it is important to check the test items for illegal behavior. In general, though legal behaviors tend to be checked sufficiently, illegal behaviors tend to be checked less thoroughly. Illegal behaviors tend to be the origins of many serious or important faults. So, in order to detect important faults, the present work focuses on the illegal behaviors of software. Therefore, the proposed method focuses on those functions that are of great importance from the viewpoint of reliability, and designs more detailed test items for them. The proposed method has two key elements: the user's viewpoint and a systematic approach. Concerning the testing from the user's viewpoint, the software specification from the user's viewpoint, which corresponds to illegal behaviors, is analyzed, and important reliability factors are related to the results of illegal behavior analysis. Reflecting this mapping information and illegal behavior analysis results, the use case description respecting illegal behaviors is summarized. Concerning the systematic approach, the test items are generated systematically by applying deviation analysis and fault tree analysis for software.

3.2 Procedure

The proposed method consists of five steps (Figure 1), and the function of each step is defined as follows:

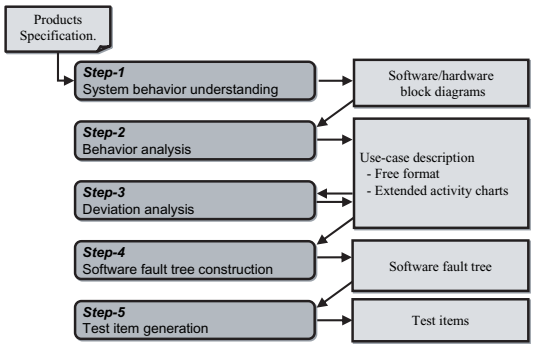

Figure 1. Overall steps of the proposed method

Step-1 (System behavior understanding) The software block diagram and hardware block diagram are described. By doing this, an outline of the functional behavior of the target software system can be understood.

Step-2 (Behavior analysis) Typical behavior of the target software is described using an activity chart and important reliability factors are clarified by use case analysis. Use case description and analysis are those used in the object-oriented development methodology.

Step-3 (Deviation analysis) According to guide words, unusual situations in the use case description are extracted and operations that deviate from the basic behavior and cause abnormalities are found.

Step-4 (Software fault tree construction) Situations that bring about undesirable illegal behaviors are analyzed by referring to analysis results and use case description. Then, the processing of software is successively considered and finally a fault tree is constructed.

Step-5 (Test item generation) By extracting the factor on the leaf of the software fault tree, test items that check this factor are generated.

3.3 Detailed procedure of the method

Here, we present the detail procedure of the method using the refrigerator control software as a typical target system.

Behavior analysis (Step-2) At first, we describe typical behaviors of target system in free format using natural language with referring to the block diagram described in Step-1. We then, based on a free format description, construct an extended activity chart [2]. The activity chart is a typical diagram in UML. Generally, an activity chart is suitable for expressing the specification from the user's viewpoint. That is, at this step, an analysis for user's operation is most important. So, user's operation should be clearly described by the activity chart. However, usually, only the legal behaviors or activities are described in a conventional activity

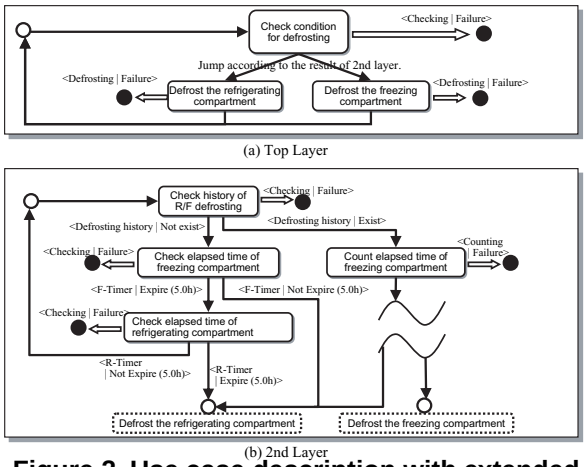

Figure 2. Use case description with extended activity chart (defroster function)

chart. In order to derive the test items for illegal behaviors, we pay equal attention to both illegal and legal behaviors. To do so, we extended the original activity charts defined by UML so that the illegal behaviors can be represented explicitly. The following procedure shows an outline to describe an activity chart.

In this procedure, we firstly describe an activity chart without any illegal behaviors in (1)–(2). We then append illegal behaviors explicitly using the extended notation in (3).

(1) The activities are represented by rectangles. The contents of an activity is defined by natural language. A circle shows an initial state of the activity flow.

(2) Transitions between activities are represented by an arrow: $\longrightarrow$. The arrow $\longrightarrow$ shows a usual or legal behavior.

(3) In addition, transitions that represent unusual or illegal behaviors are described as arrows $\Longrightarrow$. These arrows are added on all activities in an activity chart. Furthermore, a black circle shows an illegal state of the activity flow.

(4) For each transition, the corresponding condition is defined and is attached as a label with the form "<activity | condition>". Typically, "activity" is user's operation or hardware.

Figure 2 shows an example of a use case description. The example partially describes a defrosting operation of refrigerator control software.

In Figure 2(a), major functions such as "Check condition for defrosting", "Defrost the refrigerating compartment" and so on, are executed successively according to the trigger of conditions. Each major function or activity is evolved or defined in detail at the lower layer of the activity chart. As in Figure 2(b), "Check condition for defrosting" is defined in detail in the second layer. The "Check condition for defrosting" is evolved into four functions. Moreover, the trigger for execution of an activity is declared in the form of "<F-Timer | Expire (5.0h)>", "<F-Timer | Not Expire (5.0h)>", and so on.

Deviation analysis (Step-3) Next, a deviation analysis is performed for the use case description using guide words, and several unusual behaviors or operations, that is, deviations from legal situation are found [4, 6]. The candidates of system deviations are derived from each illegal behavior in the use case description (they are described as $\Longrightarrow$). Any deviation thus can be extracted to be a key factor for software failures. Here, guide words are prepared for failure of software, hardware and environment. The guide word includes words to represent illegal behaviors related to the system failure. For example, they are "be lost," "be too fast(slow)," "be incorrect," etc.

For example, concerning "Check elapsed time of freezing compartment" in Figure 2(b), the illegal behavior is considered as "<Checking | Failure>". The candidates of detailed errors for this illegal behavior are "Timer data is lost," "Timer speed was too fast," "Timer data is incorrect," and so on.

Construction of software fault tree (Step-4) A software fault tree is constructed for software failure which is related to the extracted software deviation [3, 8, 9]. The System and Software Fault Tree Analysis (SS-FTA) is divided into the following three phases:

(1) Define the root node: In the first phase, the fatal failure for target software is taken as a root of the software fault tree. Considering the troubles in the past and also imaging the troubles in the future, the most undesirable event for the target system is selected as the root node of the fault tree.

(2) Extract software function failure: In the construction of fault tree, each node is expanded into its son nodes based on the use case analysis. The second phase focuses on functional failures, which cause the failure specified at the root of the software fault tree. In order to obtain cause and result relations, we trace functional behavior flow in the use case description, and extract illegal behaviors at function level. Based on this analysis, we decompose a functional failure F into such functional failures $F'_1, F'_2, \cdots$ that each F'_i can be a cause of F.

(3) Evaluate software error: The third phase successively expands the software function failure into software errors in the implement of the target system. The analysis result by use case deviation and the detailed structure of software are reflected in this expansion. As a result, we get the software errors, which may be included in implemented software module, at the leaves of the fault trees.

Figure 3 shows an example of the software fault tree for the fatal failure: "refrigerator cannot defrost." As shown in Figure 3, the failure pattern "Failure in checking elapsed

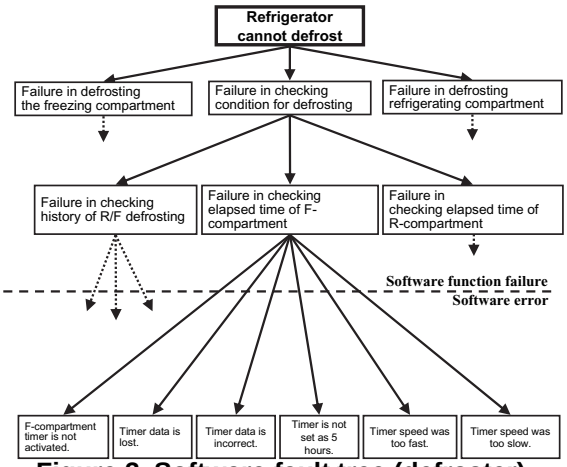

Figure 3. Software fault tree (defroster)

time of F-compartment" is evolved to 6 detailed factors. These factors include candidates of detailed errors extracted in Step-3.

Generation of software test items (Step-5) Test items are generated according to the software fault tree. For any software error specified in the leaf of the software fault tree, we list up the corresponding test items that check the errors [3, 8, 9]. For example, at the bottom of Figure 3, we can find two leaves, namely "F-compartment timer is not activated" and "Timer data is lost". These two leaves corresponds to two test items $t_A^{3.1}$ and $t_A^{3.2}$ shown in Figure 5, respectively. The enumerated test items thus constitute the most fundamental test items. Then if necessary, test items are generated for the interior node in the software fault tree. Finally, some of related test items, which are usually generated for a certain subtree, are grouped into one category.

4 Experimental Application

4.1 Purpose of the experiment

In order to evaluate the effectiveness of the proposed method, an experimental application was performed. The experimental application was divided into two experiments. First one is a preliminary experiment, and second one is a main experiment.

(a) Preliminary experiment At first, we set up a hypothesis that there are some differences in test item generation according to the engineer's skill level. In order to confirm this hypothesis, we perform a preliminary experiment. In the experiment, two engineers with different skill level were prepared. They generated test items for the target software by the conventional method. We compared the differences of generated test items' features.

(b) Main experiment We performed the main experiment in order to confirm the effectiveness of the proposed method. An engineer with low skill was prepared. He generated test items for the target software by the proposed method. We confirmed the differences between the test items generated by the proposed method and the test items generated by the conventional method.

		Condition	Expected behavior	Legal / Illegal
T_B	t_B^1	When the power is on and the time elapses 5 hrs.	RF-defrosting starts.	L
	t_B^2	Whenever the time elapses 8 hrs 30 min.	First F-defrosting starts, then RF-defrosting starts.	L
	t_B^3	During F-refrigerating or during compressor inactivation and ice is not removed or R-defrosting sensor senses 3.0 degrees centigrade.	Next cycle of RF-defrosting is activated.	L

		Condition	Expected behavior	Legal / Illegal
T_C	t_C^1	From the timing of power on to 4hrs 30min.	**Defroster shall not activate.**	IL
	t_C^2	After 4 hrs 30 min elapsed from power-on.	After waiting 20 min, RF-defrosting starts.	L
	t_C^3	From the timing of F-defrosting starting to 8 hrs 10 min, and the strong cooling is not continued for 6 hrs.	**Defroster shall not activate.**	IL
	...	...	...	...
	t_C^{12}	During the strong cooling mode, in case of no defroster is activated for 5 hrs.	After waiting 5 hrs from 2nd defrosting, then defrosting will start.	L

Figure 4. Overview of test items T_B and T_C

4.2 Experimental conditions

(a) Target system In the experiment, test items for refrigerator control software was generated using the new method based on the use case description. The target software is actual control software for refrigerator which developed in a certain company. It has about 32K bytes in ROM size, and we especially focused on the ice making function of the system. In the experiment, the specification for the defrosting function in the refrigerator control described in natural language was given to the engineers.

(b) Participants of the experiments Three engineers, Eng_A, Eng_B, and Eng_C, participated in the experiment and generated test items for sample target refrigerator control software.

Eng_A and Eng_B are considered to have roughly the same skill level. Although they have little experience of developing refrigerator software, they have developed other software systems, and developed test items for those systems. The skill level of Eng_C is higher than that of the other engineers.

As for the preliminary experiment, Eng_B and Eng_C participated in the experiment, and we confirm the differences of the generated test items according to the skill levels. In the main experiment, Eng_A generated test items with the proposed method, and then we evaluate the effects of the proposed method.

5 Evaluation of Experiments

5.1 Preliminary experiment

Preliminary experiment is an experiment that confirms the effects of the engineer's skill level on the test item generation. In the preliminary experiment, sets of test items T_B and T_C are generated by engineers Eng_B and Eng_C, respectively, using the conventional method (See Figure 4). T_B and T_C include 3 and 12 test items, respectively.

As Figure 4 shows, all of test items in T_B are related to the legal behavior of the target system. These test items are prepared to confirm functions' behaviors that are clearly

described in the specification document. That is, the descriptions of test items are almost the same as those in the specification.

For example, consider the first item of T_B (t_B^1), the specification for defroster activating condition is "If the power is on at the condition of elapsed timer count is 5 hours, F/R-defrosting should start." in the specification document. In Figure 4, the description of the test item t_B^1 is almost the same description of this specification document. In an actual behavior of the system, some tolerance for activation timing inevitably occurs and many faults in the embedded system are caused by timing error depended on these tolerance. But the test item t_B^1 only pointed out the legal behavior clearly described in the specification and did not touch with these tolerance for activation timing of defroster, so it is difficult to detect the illegal behavior depended on these timing error. As is easy to assume from this example, it was not possible to generate test items concerning illegal behaviors not described in the specification with the conventional method.

On the other hand, as shown in Figure 4, Eng_C assumed the software architecture or user's illegal operations, and then generated the detailed test items relating to illegal behavior. For example, consider the first item of T_C (t_C^1) in Figure 4, this test item corresponds to t_B^1 that related to the defrosting start behavior. In the specification document, the defrosting start timing is only instructed at elapsed timer count is 5 hours. However, Eng_C noticed that it is undesired to activate the defroster before the timer count elaspes 5 hours. Thus, he generates the item t_C^1 in which such undesired (that is, illegal) behavior is included[1].

From this experiment, we can conclude that there are some effects of engineer's skill on the test item generation.

5.2 Main experiment

(a) Generated test items In the main experiment, engineers Eng_A received detailed explanation of the proposed method. Then, a set of test items T_A is generated by Eng_A using the proposed method (See Figure 5). Since T_A is generated by the systematic method, the style of test items is slightly different from that of in Figure 4. After generating test items, we extract test items for the defrosting function of the refrigerator and compare the obtained test items, and also analyze the coverage of test items.

As a result, the engineer Eng_A generated 55 test items. From the detail investigations of the test items, we confirm that T_A includes 9 legal test items (e.g. $t_A^{1.1}$) and 4 illegal test items (e.g. $t_A^{3.13}$), and the other test items can be considered as conditions to check general defects. Table 1 shows the number of test items for legal and illegal behaviors.

Here we analyze the feature of the generated test items. For example, consider the category "3: First defrosting after power-on defrosting", which is generated by Eng_A. Six

[1]Moreover, Eng_C considered the tolerance of activating timing and referred the timer data table which was described in another part of the specification document by his own judgment. Then, he understood that the defroster shall not be activated before 4 hours 30 minutes in elapsed timer by considering the error margin of timer. Thus, the threshold of the timer count in T_C is slightly different from that of T_B and T_A.

	Category	Condition and Expected behavior		(L)egal / (IL)legal / (P)re-condition
T_A	1 Initial defrosting	$t_A^{1.1}$	Is history checking activated?	L
	2 Checking history of R/F defrosting	$t_A^{2.1}$	Does defrosting history exist?	L
		$t_A^{2.2}$	Is defrosting history correct?	L
	3 First defrosting after power on defrosting	$t_A^{3.1}$	Is F-timer activated?	P
		$t_A^{3.2}$	Are elapsed time data for F-timer lost?	P
		$t_A^{3.3}$	Are elapsed time data for F-timer correct?	P
		$t_A^{3.4}$	Is F-timer set for 5 hrs?	P
		$t_A^{3.5}$	Does F-timer count correctly?	P
		$t_A^{3.6}$	Is R-timer activated ?	P
		$t_A^{3.7}$	Are elapsed time data for R-timer lost?	P
		$t_A^{3.8}$	Are elapsed time data for R-timer correct?	P
		$t_A^{3.9}$	Is R-timer set for 5 hrs?	P
		$t_A^{3.10}$	Does R-timer count correctly?	P
		$t_A^{3.11}$	In case of F-timer > 5 hrs, is F-defroster activated?	L
		$t_A^{3.12}$	In case of R-timer > 5 hrs, is R-defroster activated?	L
		$t_A^{3.13}$	In case of either F- or R-timer < 5 hrs, is either F- or R- defroster not activated, respectively?	IL
	4 Regular defrosting	$t_A^{4.1}$	Is F-timer activated?	P
		$t_A^{4.2}$	Does F-timer count correctly?	P
		$t_A^{4.3}$	Is F-timer set for 8.5 hours?	P
		⋮	⋮	
		$t_A^{4.10}$		
	5 R/F-Timer > 8.5 h	$t_A^{5.1}$	Is Defrosting-counter activated?	L
		⋮	⋮	
		$t_A^{5.6}$		
	6 R/F-Timer < 8.5 h	$t_A^{6.1}$	Is Moist/Defrost mode activated?	L
		$t_A^{6.2}$	⋮	
		$t_A^{6.3}$		
	7 Moist/Defrost mode	$t_A^{7.1}$	Is compressor checking mode activated?	L
		⋮	⋮	
		$t_A^{7.15}$		
	8 R-timer > 6.0 h	$t_A^{8.1}$	Is elapsed time set as half mode?	L
		⋮	⋮	
		$t_A^{8.5}$		

Figure 5. Overview of test items T_A

Table 1. Number of test items for defrosting function

	T_A
Test items for legal behaviors	9 ($t_A^{1.1}$, $t_A^{2.1}$, $t_A^{2.2}$, $t_A^{3.11}$, $t_A^{3.12}$ $t_A^{5.1}$, $t_A^{6.1}$, $t_A^{7.1}$, $t_A^{8.1}$)
Test items for illegal behaviors	4 ($t_A^{3.13}$, $t_A^{4.9}$, $t_A^{5.5}$, $t_A^{8.4}$)
Pre-condition test items	42 ($t_A^{3.1}$, $t_A^{3.2}$, $\cdots$)
Total	55

test items are included; $t_A^{3.2}$, $t_A^{3.5}$, $t_A^{3.7}$, $t_A^{3.10}$, $t_A^{3.11}$, and $t_A^{3.12}$. In those test items, test items $t_A^{3.2}$, $t_A^{3.5}$, $t_A^{3.7}$, and $t_A^{3.10}$ should be considered pre-conditions[2] for checking $t_A^{3.11}$ and $t_A^{3.12}$. These test items related to the pre-condition checking are said to be useful for performing the test in concretely. This can be said that test items by the proposed method are more concretely extracted, and that all necessary items are explicitly enumerated.

(b) Checking illegal behaviors In this paper, we define conditions which deal with transitions with $\Longrightarrow$ in the use case description as test items for illegal behaviors. On the other hand, conditions for $\longrightarrow$ are called test items for legal behaviors.

For example, consider test items related to the third category; "Initial defrosting after switch on," again. Then engineer Eng_A generated the test item "$t_A^{3.13}$: In case of either F- or R-timer $<$ 5 hours, is either F- or R-defroster *not* activated, respectively?", as well as "$t_A^{3.11}$: In case of F-timer

[2]Pre-condition means a condition that should be checked to ensure that other test items work correctly.

Table 2. Number of test items for defrosting start function

	T_A	T_B	T_C
Test items for legal behavior	6	3	8
Test items for illegal behavior	4	0	4

$>$ 5 hours, is F-defroster activated?" and "$t_A^{3.12}$: In case of R-timer $>$ 5 hours, is R-defroster activated?" (These items correspond to the test items t_C^1 and t_C^2 generated by the experienced engineer.)

Specifications for software in general are likely to contain normal or legal behaviors, so test items from these specifications are inevitably focusing on the legal behaviors. In this case,it is normal behavior that the defroster activates after 5 hours in the timer count. On the other hand, in many cases, there is no description of defroster behavior before 5 hours in the timer count, because the experienced engineer will easily assume this behavior from above legal behavior. For example, in case that the defroster is activated before 5 hours in the timer count, this behavior is considered an illegal behavior.

Generally, conventional method gave test items for legal functions only. But the proposed method can generate test items for illegal behaviors such as $t_A^{3.13}$. The result of this experiment shows that the test items for checking both legal and illegal system behaviors were generated by the proposed method.

5.3 Comparison with conventional method

Here, we discuss the following two viewpoints — (1) Comparison between T_A and T_B, and (2) Comparison between T_A and T_C. Eng_A, who is a low skill engineer generated the test set T_A using the proposed method. Eng_B, who is a low skill engineer, and Eng_C, who is a high skill engineer generated the test sets T_B and T_C, respectively, using the conventional method.

(a) Comparison between T_A and T_B Table 2 shows the number of test items concerning the defrosting start function, which can be treated as a comparison domain in common. From this table, we can confirm that the number of test items in T_A (that is, 6) is more than that of T_B (that is, 3)[3]. Especially, T_A includes 4 test items for illegal behaviors. On the contrary, T_B did not include any test items for illegal behaviors. Since it was confirmed that Eng_B has roughly the same skill level as Eng_A, it is reasonable to think that the proposed method has some advantages for generating test items for illegal behaviors.

(b) Comparison between T_A and T_C From the result of comparison in Table 2, we confirm that the numbers of test items in T_A and in T_C are almost the same. As mentioned before, Eng_C is an engineer with considerable experience of development and testing, and it is found that using the conventional method he can generate test items which are as detailed as those generated by the proposed method.

(c) Discussion The proposed method enables engineers with low skills to generate detailed test items for illegal behaviors comparable to those generated by highly skilled engineers.

[3]The number of legal test items are smaller than that of Table 1, since we focus on the "defrosting start" function.

6 Conclusion

This paper discussed a difference of test item generation between an engineer with high skill and an engineer with low skill. From the experimental application, we confirmed that there are some differences in the way of generating illegal test items. Moreover, we propose a new generation method of test items, which mitigates the effects of skills on generating the test items. From the experimental evaluation, we also confirmed that the proposed method can generate almost the same test items without depending on the skill of the test engineers. As a future work, we will investigate what test coverage for illegal behaviors can be obtained by this method.

References

[1] B. Beizer. *Black-Box Testing*. John Wiley & Sons, New York, 1995.

[2] H. E. Eriksson and M. Penker. *UML toolkit*. John-Wiley & Sons, 1997.

[3] T. Fukaya, M. Hirayama, and Y. Mihara. Software specification verification using fta. In *Proc. of FTCS-24*, pages 131–133, 1994.

[4] N. G. Leveson. *Safeware: System safety and computers*. Addison-Wesley, MA, 1995.

[5] J. D. Musa. *Software Reliability Engineering: Faster Development and Testing*. McGraw-Hill, 1998.

[6] J. D. Reese and N. G. Leveson. Software deviation analysis. In *Proc. of 19th International Conference on Software Engineering*, pages 250–260, 1997.

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

[8] K. Tamura, J. Okayasu, and M. Hirayama. A software testing method based on hazard analysis and planning. In *Proc. of 9th International Symposium on Software Reliability Engineering*, pages 103–110, 1998.

[9] T. Tsuchiya, H. Terada, E. M. Kim, and T. Kikuno. Deviation of safety requirements for safety analysis of object-oriented design specification. In *Proc. of 21st Annual International Computer Software & Applications Conference*, pages 252–255, 1997.