Generating Test Items for Checking Illegal Behaviors in Software Testing

Masayuki HIRAYAMA[†‡], Jiro OKAYASU[†], Tetsuya YAMAMOTO[†]
[†] Research and Development Center, 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

Even for electrical appliances, testing for illegal behaviors becomes difficult since the software system in an electrical appliance has already become large in size. Actually, the conventional method cannot generate sufficient test items for illegal behaviors. But testing illegal behaviors becomes more and more important, since the failure of electrical appliances would cause fatal effects on our daily life. We therefore propose a new method for generating appropriate test items to check illegal behaviors, which consists of the following steps: (1) Describe software behavior using use case notation, (2) Analyze illegal behavior by the deviation analysis technique, (3) Construct a software fault tree using the above information, and (4) Generate test items from the software fault tree.

This paper also reports the experimental applications to actual development of an electrical appliance. The evaluation results identified that all necessary test items for illegal behaviors are included in the resultant test items.

1 Introduction

In this paper, we try to develop a practical approach to attain the reliability of software. We consider that the practical approach must have the following properties: (1) it is easy to understand the approach, (2) it is easy to apply the approach to the target software, and (3) the cost or effort needed is reasonable. In that sense, the software testing[3, 5, 8, 11] is a well known practical approach to assure the quality of target software. Truly speaking, we cannot find any other useful methods than software testing from the viewpoints of practical software development.

However, the recent software development regards that even the cost in the software testing should be reduced. So the automatic or systematic generation of the test items for the software systems were expected to mitigate the development cost. For the automatic generation of test items, several approaches (such as a method based on the formal specification[7], a method by the source code analysis[12], and so on) have been proposed. However, the method based on the formal approach was too complex to apply to the actual development field, and the method based on the source code analysis was only applied to toy examples. So, we have to devise a method that is applicable to the practical software developments.

On the other hand, the quality of the resultant software after software testing is greatly influenced by the amounts and kinds of test items[5]. According to purposes of software testing, we can classify test items into two: (1) the one related to legal behaviors of target software and (2) the one related to illegal behaviors of target software. The test items in the first type are extensively checked in an ordinary software testing. But in order to assure high reliability of the software system, the violations of the second type, that is illegal behaviors, should be avoided, and so exhaustive enumeration of test items are tried.

Recently, it is remarkably observed that so many computer systems or software systems are introduced into our social or daily life[2], and then, most of electrical appliances have embedded software in them. If software faults are remained in an electrical appliance and the electrical appliance suddenly don't work correctly, then the scope of its effect or damage may become enormously large. Therefore even for electrical appliances, ensuring the reliability of software becomes an essential requirement[9].

Now we briefly summarize the development of electrical appliances. While software embedded in an electrical appliance is relatively small in size, the formal method based on finite state machine is used effectively in the development[2]. All legal behaviors written in the functional specification are formally defined by the finite state machine, and the corresponding test items are generated using transition sequences on the finite state machine. On the contrary, test items for illegal behaviors are successfully derived and checked carefully by experienced developers.

Since the customers' demands for electrical appliances have increased rapidly, the size of software has already become large. Thus the formal approach doesn't work effectively in the development of the recent products. The formal method then can be applied only to the core part of the system. Thus test items only for legal behaviors are generated by the formal method. We have not yet have good solutions to generate test items for illegal behaviors. Additionally, since the consumers' demand generally changes within a short period, the development must be completed timely. The timeliness makes the difficult situations concerning illegal behaviors more serious.

In this paper, we propose a new method for generating test items to check illegal behaviors in the development of

electrical appliances. We then conducted an experiment that applies the proposed method to actual development of an electrical appliance in a certain company. The evaluation results show that (1) the sets of test items generated by two engineers are almost the same, and (2) the set of test items constructed by the proposed method covers the set of test items that were used conventionally at the project.

2 Method for Generating Test Items

2.1 Outline of proposed method

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

Step-1 (System behavior understanding): We describe the software block-diagram and hardware block-diagram. By doing this, we can understand an outline of the functional behavior of the target software system.

Step-2 (Use case analysis): We describe typical behavior of the target software using activity chart and clarify important reliability factors by applying use case analysis. Use case description and analysis are borrowed from the object-oriented developing methodology (UML).

Step-3 (Deviation analysis): According to guide words, we extract unusual situations in the use case description and find such operations that deviate from the basic behavior and cause abnormalities.

Step-4 (Software fault tree construction): We analyze the situations that bring undesirable illegal behaviors by referring to analysis results and use case description. We then successively consider the internal processing of software, and finally construct a kind of fault tree.

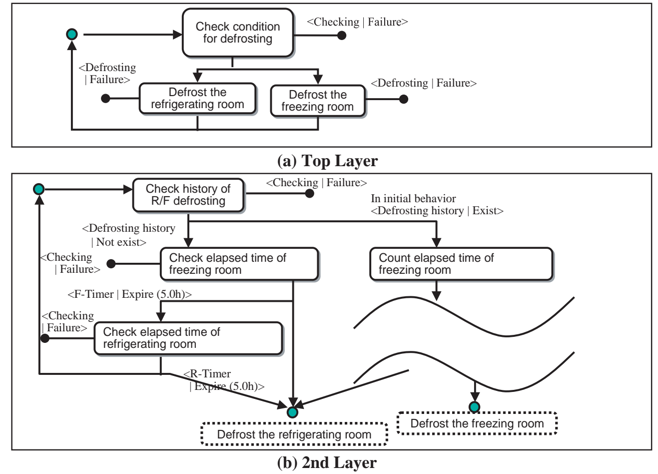
Step-5 (Test item generation): By extracting the factor on the leaf of the software fault tree, we generate test items that check this factor.

In this study, the size of target systems for this method is assumed to be not so large. Thus we can obtain the activity charts and the fault trees with small effort. We will explain our method using the refrigerator control software as a typical target system (to be described in subsection 3.2).

2.2 Use case analysis

At first, we describe typical behaviors of target system in free format using natural language. We then, based on a free format description, construct an extended activity chart[1]. The following extension was done for the original activity charts defined by UML, in order to facilitate the reliability requirement analysis.

(1) The activities are represented by rectangles. The contents of an activity is defined by natural language.



(a) Top Layer

(b) 2nd Layer

Figure 1. Use case description (defroster)

(2) The transition between activities is represented by two kinds of arrows: $\longrightarrow$ and —•. An arrow $\longrightarrow$ shows a usual or legal behavior, but the other arrow —• shows an unusual or illegal behavior.

(3) For each transition, the corresponding condition is defined and is attached as a label with the form "<activity | condition>".

Figure 1 shows an example of a use case description. The example partially describes a defrosting operation of refrigerator control software. In this example, basic activities such as "Check condition for defrosting", "Defrost refrigerating room" and so on, are executed successively. Moreover, the trigger for execution of an activity is declared in the form of "<Defrosting history | Not exist>", "<F-Timer | Expire (5.0h)>" and so on.

Next a deviation analysis is performed for the use case description using guide words, and several unusual behaviors or operations are found as deviations from legal situation[4, 6]. 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.

2.3 Construction of software fault tree

A software fault tree is constructed for software failure which is related to the extracted software deviation[2, 9, 10]. The fatal failure for target software is taken as a root of the software fault tree. In the construction of fault tree, each node is expanded into its son nodes based on the use case analysis. The software fault tree is divided into the two parts:

(1) Software Function Failure This part focuses on functional failures, which causes 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.

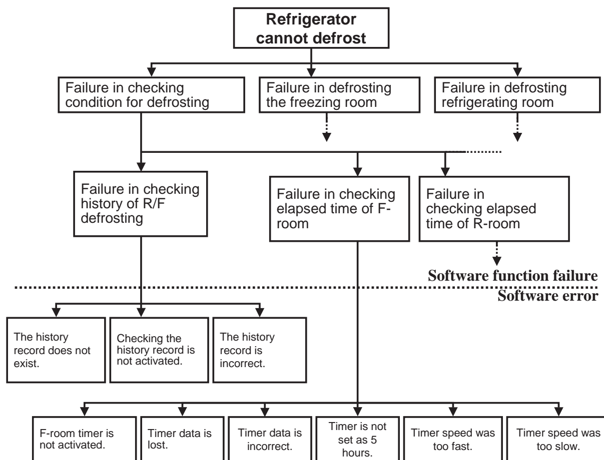


Figure 2. Software fault tree (defroster)

(2) Software Error This part 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 2 shows an example of software fault tree for the fatal failure: refrigerator cannot defrost.

2.4 Generation of software test items

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[2, 9, 10]. 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 condition.

3 Experimental Application

3.1 Design of experiment

The new method for generating software test items based on the use case description was applied to a refrigerator control software. The following shows the characteristic of the experiment.

(1) Test engineers E_A and E_B: Two persons, E_A and E_B, joined the experiment and generated test items for sample target refrigerator control software. They are considered to have almost the same skill level. Additionally, they have little experience of developing refrigerator software. But they have developed some other systems and thus already generated test items for them.

(2) Outline of the experiment: The purpose of the target system is to implement the defroster function in the refrigerator control software. Before generating test items, engineers were explained about the details of the proposed method. They were presented a specification described in natural language. They then generated test items by using the proposed method.

(3) Evaluation of the result: In order to evaluate the experimental results, all documents are reserved and the working time is recorded. Moreover, the comments with respect to difficulty and easiness of the proposed method are freely written on the memorandum.

3.2 Target system

The experimental target was the control software embedded in the refrigerator. The control software monitors various sensor data and performs real-time control of hardware on the 8-bit microcomputer. In the experiment, the specification for the defroster function in the refrigerator control was given to the engineers.

The refrigerator has two cooling rooms: a freezing room and a refrigerating room. The defrosting function is constructed by two basic activating operations for the defrosting heater and defrosting fan, and is controlled based on the states of two rooms, elapsed time of defrosting, and rooms' conditions. During defrosting, both the defrosting heater and fan must operate at suitable time intervals to prevent the rooms of the refrigerator from reaching too high temperature.

3.3 Resultant documents

Here we show how test items are generated using some examples. We assume that a use case description for the defrosting function of the refrigerator shown in Figure 1 is given. We then construct actually a software fault tree shown in Figure 2, and generate test items shown in Figure 3 from the software fault tree. These documents are actually described by the engineer E_A in the experiment.

At Step-1, the engineers E_A and E_B read carefully the given functional specification to understand the functions of the defroster unit. They then described software block diagrams for the defroster.

At Step-2, the use case description for the defroster function is described based on the software block diagrams and the functional specification. First, the top-level layer of the use case description(shown in Figure 1(a)) is described by tracing the block diagrams roughly. The engineers then described a detailed description. For example, the activity "Check condition for defrosting" in Figure 1(a) is extended into the detailed description shown in Figure 1(b) using the functional specification.

At Step-3, the illegal behaviors of the defroster are extracted by tracing the activity chart shown in Figure 1. The guide words support to extract the detail of an illegal behavior. For example, consider the activity "Check history of R/F defrosting." The guide words for the "history" are "not exist" and "be incorrect," and the guide word for "check" is "be

Condition	Test Items
A1 Switch on defrosting	**A1.1** Is history checking activated?
A2 Checking history of R/F defrosting	**A2.1** Does defrosting history exist?
	A2.2 Is defrosting history correct?
A3 Initial defrosting after switch on	**A3.1** Is F-room timer activated?
	A3.2 Does F-room timer count?
	A3.3 Does F-room timer count correctly?
	A3.4 Is F-room timer set for5 hours?
	A3.5 Is neither F- nor R-defroster activated, so far as F- and R-room timer < 5 hours?
	A3.6 Is F or R-defroster activated, so far as timer count is larger than 5 hours?
	......
......	......
A8 Defrosting timer > 6 hours	**A8.1** Is the remaining time correct?
	A8.2 Is the timer set by half of the remaining time?
	......
A9 Check for residual ice	......

Figure 3. Generated test items

not activated." Note that the prepared guide words are not necessarily complete, thus new guide words are appended by engineers incrementally if necessary. These guide words are extensively used in Step-4.

At Step-4, by tracing the activity chart in Figure 1, the software fault tree shown in Figure 2 is constructed as follows: First, the root node is determined. Since the most crucial failure for the defroster is that the frost cannot be removed, the failure "Refrigerator cannot defrost" is adopted as the root. According to the top layer description, three failures "Failure in checking condition for defrosting," "Failure in defrosting freezing room" and "Failure in defrosting refrigerating room" are chosen as the next level nodes. Then, for the "Failure in checking condition for defrosting," the failures "Failure in checking history of R/F defrosting," "Failure in checking elapsed time of F-room" and so on are derived as the next level nodes. Then, using the result of Step-3, each node is expanded to some nodes successively.

Finally, the expansion reaches a situation that the expanded node corresponds to the fundamental or undividable error. At this time, the expansion will stop and software fault tee is completed. In this case, the subtree with the root "Failure in checking history of R/F defrosting" has three leaves as shown in Figure 2. These leaves correspond to activities "The history record does not exist," "Checking the defroster history is not activated" and "The history record is incorrect," respectively.

At Step-5, test items are derived from the leaves and interior nodes of the software fault tree. From the leaves with light gray in Figure 2, the test items A1.1 to A3.4 shown in Figure 3 are generated.

As a result, the engineer E_A generated 55 test items (which are grouped into A1, A2, $\cdots$, A9), and the engineer E_B generated 40 test items (which are grouped into B1, B2, $\cdots$, B10). On the contrary, the conventional method generated 8 test items (which are grouped into C1, C2, C3 and C4).

4 Experimental Evaluation

4.1 Comparison between E_A and E_B

Here we analyze the coverage of test items between both engineers E_A and E_B. As mentioned before, the engineer E_A generated 55 test items and the engineer E_B generated 40 test items. Thus there exist at least a big difference in the total number of test items. After the investigations on the test items, it is found that they focused on the different part of the specification each other.

Thus, it is not a good way to compare all the items. We therefore take the same functions and compare the test items for the selected functions between them. Then, 35 test items and 21 test items are chosen from the engineers E_A and E_B, respectively.

Let us show some examples. Figure 4 shows the correspondence between test items by E_A and E_B. Consider the test items for checking the initial defrosting condition. Although there are some differences in expression between engineers E_A and E_B, similar test items, such as A3.11 and B6.2.1, and A3.13 and B6.1.1, were extracted. Next consider the test items for the defrosting timer in freezing room. Then both A3.2 and A4.2 are semantically the same as item B2.2.1, and both A3.3 and A4.3 are semantically the same as B2.3.2. These can be considered as typical test items for illegal functions.

By analyzing the correspondence of test items, 17 out of 35 test items by E_A correspond to 11 out of 21 test items by E_B as shown in Table 1. Almost half of test items are identified as the same ones. Then, looking details of the rest of test items, these are mostly identified as closely related testing items, that should be performed beforehand. Thus we can say that the interpretation of the free format description (see subsection 2.2) is performed successfully and the proposed method can generate almost the same test items by any engineers.

Table 1. Comparison between E_A and E_B

	Engineer E_A	Engineer E_B
Test items generated by E_A or E_B exclusively	18	10
Test items generated by E_A and E_B	17	11

4.2 Comparison with conventional method

4.2.1 Granularity of test items

As mentioned before, the engineer E_A generated 55 test items. However, only 8 test items were conventionally used in the development. The correspondence between them is shown in Figure 5.

For example, consider the test item "C1: Initial behavior" in the conventional check item. Then the related test items generated by E_A include not only A3.11 and A3.12 for defrosting behavior but also A3.2, A3.3, A3.7 and A3.8 for the timer behavior as shown in Figure 5. The test items A3.2, A3.3, A3.7 and A3.8 should be considered pre-conditions for

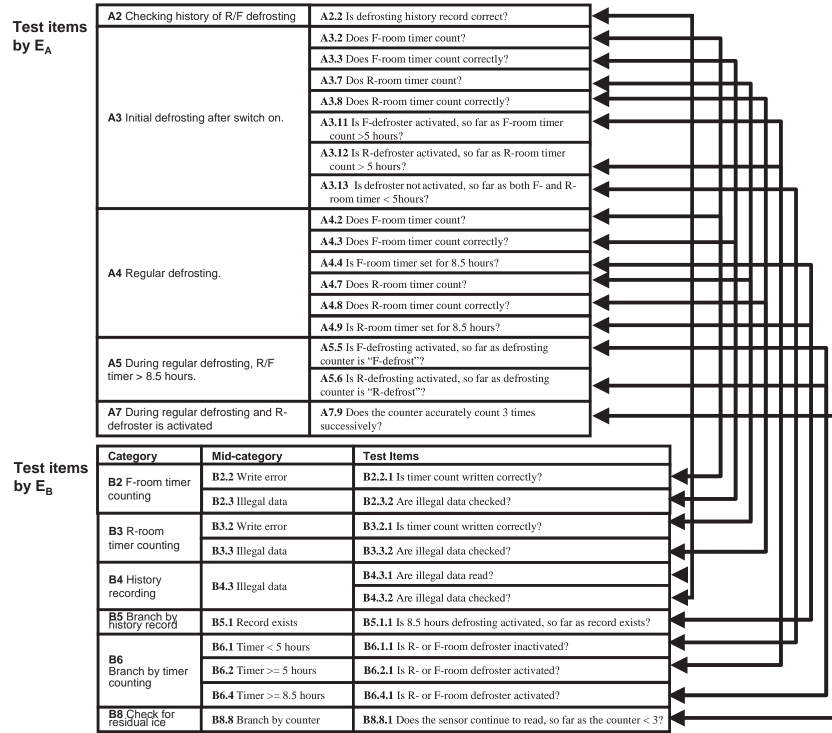


Figure 4. Correspondence between E_A and E_B

checking the C1 test item. The conventional test items list up only essential items that summarize several detailed test items, and thus know-how or information for checking test items are invisible(by this abstraction). On the other hand, it can be said that test items by the proposed method are more concretely extracted, and that all necessary items are explicitly enumerated. It is almost the same for the conventional test item C2.

4.2.2 Checking illegal behaviors

In this paper, we define conditions which deal with transitions with ——• 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 item C1.1 "Is R/F-defroster activated, so far as F-room timer or R-room timer > 5.0 hours?" in the conventional test items. Then engineer E_A generated test item A3.13 "Is defroster not activated, so far as both F- and R-room timer < 5 hours?" as well as A3.11 "Is F-defroster activated, so far as F-room timer counts > 5.0 hours?" and A3.12 "Is R-defroster activated, so far as R-room timer counts > 5.0 hours?". Generally, conventional method gave test items for legal functions only. The test engineers have enough know-how and heuristics, and thus they can manage successfully testing illegal behaviors based on their experience. But the proposed method can generate test items for illegal behaviors such as A3.13. Table 2 shows the analysis result of total number of test items for legal and illegal behaviors.

Table 2. Test items for behaviors

	Conventional	Engineer E_A	Engineer E_B
Test items for legal behaviors	5	9	5
Test items for illegal behaviors	0	4	4

4.2.3 Advantage of proposed method

In the conventional development, there exist two software defects D_1(checking error of sensor data) and D_2(data error in setting timer), which test engineers couldn't find by the system test. The details of D_1 and D_2 are shown in Table 3. The defects D_1 and D_2 were detected by the final quality assurance test phase, and thus were considered to be serious or hard to detect.

Fortunately, the test items generated by engineer E_A contain test items A7.5 and A7.6 for the defect D_1 and test items A8.4 and A9.4 for the defect D_2. It is thus expected that test items generated by the proposed method will help test engineers to detect such defects D_1 and D_2.

5 Conclusion

This paper has proposed a new method for generating test items to check illegal behaviors. The following were concluded from an experimental application of the proposed method to the refrigerator control software.

(1) With relatively a little additional effort, we can generate test items suitable to check behaviors (especially, illegal behaviors) of the target system.

Table 3. Analysis of defects

Detected defects	Conventional	Test items generated by E_A
D_1: R/F-defrosting sensor is not checked	No test items	A7.5 Does R-defrosting sensor exist? A7.6 Is R-defrosting sensor correct?
D_2: Timer was not set by half at freezing mode.	No test items	A8.4 Is freezing timer set for 8 hours? A9.4 Is timer set by half, so far as freezing timer > 6 hours?

Conventional Test items

Item	Test Items
C1 Initial behavior	**C1.1** Is R/F-defroster activated, so far as F-room timer or R-cooling timer > 5.0 hours?
C2 Regular defrosting	**C2.1** Is defroster activated at 8.5 hours interval?
	C2.2 So far as F-room is freezing and R-defrosting sensor < 3C, will both F- and R-room be defrosted on the next timing?
	C2.3 So far as freezing is enforced for 6 hours, is the remaining defrosting time set by half?
C3 Timer counting	**C3.1** Is defrosting timer set by 5 hours ± 9 minutes?
	C3.2 Is defrosting timer set by 8.5 hours ± 15 min.?
	C3.3 Is enforced freezing timer set by 6 hours ± 11 minutes?
C4 Behavior of timer	**C4.1** Is timer inactivated at the time compressor stops.

Test items by E_A

Condition	Test Items
A3 Initial defrosting after switch on.	**A3.2** Does F-room timer count?
	A3.3 Does F-room timer count correctly?
	A3.7 Does R-room timer count?
	A3.8 Does R-room timer count correctly?
	A3.11 Is F-defroster activated, so far as F-room timer counts >5 hours?
	A3.12 Is R-defroster activated, so far as R-room timer counts > 5 hours?
......	......
A5 During regular defrosting & R/F timer < 8.5 hours.	**A5.1** Is defrosting counter checked?
	A5.2 Does counter data exist?
	A5.3 Is counter set correctly?
	A5.5 Is F-defrosting activated, so far as defrosting counter is "F-defrost"?
	A5.6 Is R-defrosting activated so far as defrosting counter is "R-defrost"?
......	......
A7 Regular defrosting & R-defroster's activation.	**A7.10** Will both F- and R-room be defrosted on the next timing, so far as F-room is freezing and R-defrosting sensor < 3 degrees centigrade?
......	......
A9 Enforced freezing timer > 6.0 hrs.	**A9.4** Is timer set by half of the remaining time?

Figure 5. Comparison with conventional test items

(2) The proposed method generates test items in more detailed descriptions, which seem to reflect and thus correspond to the implementation of the target software.

(3) The proposed method generates systematically all necessary test items without omission(if a given specification is well written and complete).

As future research works, we are now extending our approach so that it can be applied to larger software systems. We are also planning to assign priorities to test items and choose appropriate set of test items according to the given restrictions on the cost or resources.

Acknowledgment

The authors would like to thank Mr. Takuya Kishimoto, Home Appliances Company of TOSHIBA Corp., for providing our case study examples. Thanks also Mr. Kazuyoshi Tamura and other researchers of System Engineering Laboratory, TOSHIBA Corp. for their helpful suggestions.

References

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

[2] T. Fukaya, M. Hirayama and Y. Mihara, "Software specification verification using FTA," Proc. of FTCS-24, pp.131–133, 1994.

[3] Y. Kim and C. R. Carlson, "Scenario based integration testing for object-oriented software development," Proc. of 8th ATS, pp.283–288, 1999.

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

[5] B. Marick, *The craft of software testing: subsystem testing including object-based and object-oriented testing*, Prentice Hall, 1995.

[6] J. D. Reese, et al., "Software deviation analysis," Proc. of 19th International Conference on Software Engineering(ICSE'97), pp.250–260, 1997.

[7] M. A. Sanchez, "Integrating testing with a formal development process," Proc. of 9th International Symposium on Software Reliability Engineering, Fast abstract and Industrial Practices, pp.205–213, 1998.

[8] K. -C. Tai and H. K. Su, "Theory of fault-based predicate testing for computer programs," IEEE Trans. on Software Engineering, vol.22, no.8, pp.552–562, 1996.

[9] K. Tamura, J. Okayasu and M. Hirayama, "A software testing method based on hazard analysis and planning," Proc. of ISSRE'98, pp.103–110, 1998.

[10] T. Tsuchiya, H. Terada, E. M. Kim and T. Kikuno, "Deviation of safety requirements for safety analysis of object-oriented design specification," Proc. of COMPSAC'97, pp.252–255, 1997.

[11] Z. Xinjun and Y. Tashiro, "An approach to automated program testing and debugging," Proc. of Fourth Asia-Pacific Software Engineering Conference(APSEC'99), pp.582–589, 1999.

[12] H. Yin, Z. Lebne-Dengel and Y. K. Malaiya, "Automatic test generation using checkpoint encoding and antirandom testing," Proc. of 8th International Symposium on Software Reliability Engineering(ISSRE'97), pp.84–95, 1997.