Systematic Generation of Software Test Items Based on System Behavior from User's Viewpoint

Masayuki Hirayama, Tetsuya Yamamoto
R&D Center System Engineering Lab., TOSHIBA Corporation
1 Komukai Toshiba-cho, Saiwai-ku, Kawasaki, 212-8582, Japan
masayuki.hirayama@toshiba.co.jp
Takuya Kishimoto
R&D Group, Home Appliance Company, TOSHIBA Corporation
Osamu Mizuno, Tohru Kikuno
Dept. of Informatics and Mathematical Science, Osaka University

Abstract

This paper proposes a new design method for software test, which will produce test cases based on Use-Case Analysis, and then generate test items systematically using Software Fault Tree Analysis. The attractive features of the proposed method are summarized as follows: (1) use cases are described from the end users' viewpoint, (2) Illegal behaviors on the use-case description are transformed into test items to be checked, and (3) test items are generated systematically without their duplication. The experimental application to defrosting subsystem of refrigerator confirms the effectiveness of the proposed method.

1. Introduction

Recently, many computer systems or software systems are introduced into our social or daily life. For example, 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[6].

When an embedded software in electrical appliances is relatively small in size, the formal method based on finite state machine is used effectively[2]. The test items for legal behaviors 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.

PSAM5

Since customers' demands for electrical appliances have increased rapidly, the size of software has already become large. Then the formal approach doesn't work effectively in the development. The formal method can be applied only to the core part of the system.

In this paper, we propose a new method for generating test items to check illegal behaviors in the development of electrical appliances. The proposed method has the following attractive features.

(1) Description from users' point of view: In order to describe the target system's behavior from user's point of view(rather than developer's point of view), the use case description proposed by UML[1] is applied.

(2) Systematic detection of illegal behaviors: In order to detect systematically the main factors of illegal behaviors of the target system, the deviation analysis[3, 4] is performed on the use case description.

(3) Stepwise generation of test items: The software fault tree[2, 6] for the target system is constructed based on the deviation analysis, and then test items for checking illegal behaviors are generated stepwisely from the fault tree.

Similar approach has already been proposed for general software by Smidts et al.[5]. They implemented an architecture based software reliability model and extensively used fault trees in the analysis. But it seems to be difficult to apply their method to the development of electrical appliances.

2. New Design Method

The proposed method consists of the following five steps:

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.

Furthermore, we construct a keyword dictionary and place such restriction on the words in Step 2, Step 3 and Step 4 that they must be taken from the keyword dictionary. By this restriction, we can avoid miscommunication among Step 2, Step 3 and Step 4.

3. Application to Refrigerator

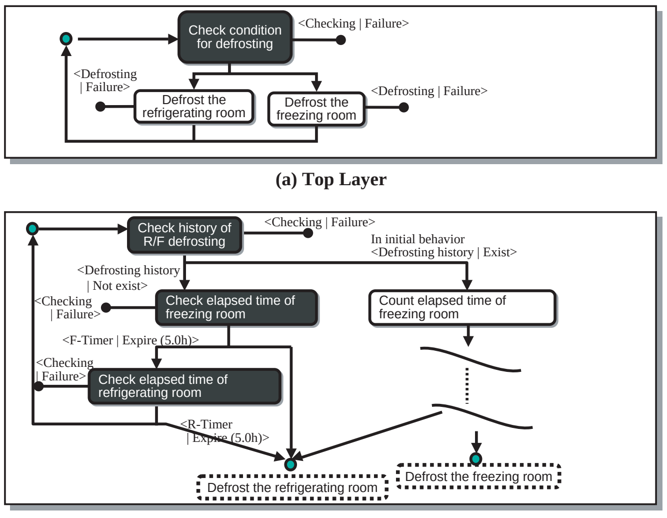
In order to confirm the effectiveness, we have analyzed real development of refrigerator. In the analysis, we consider software for a defrosting subsystem (See Figure 1). The refrigerator has two cooling rooms: a freezing room and a refrigerating room. The defrosting function is constructed by two basic activating operations for the defrosting heater and defrosting fan, and it is controlled based on the states of two rooms, elapsed time of defrosting, and rooms' conditions.

At Step-1, the engineer reads carefully the given functional specification to understand the functions of the defroster unit, and then describes software block diagrams for the defroster.

- Refrigerator system
  - Freezing room
    - Defrosting timer
    - Defrosting fan
    - Defrosting heater
    - Room thermometer
  - Refrigerating room
    - Defrosting timer
    - Defrosting fan
    - Defrosting heater
    - Room thermometer

Fig. 1. Organization of refrigerator system

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 2(a)) is described by tracing the block diagrams roughly. The transition between activities is represented by two kinds of arrows: $\longrightarrow$ and $\relbar\!\!\relbar\!\bullet$. An arrow $\longrightarrow$ shows a usual or


Check condition for defrosting
<Checking | Failure>
<Defrosting | Failure>
Defrost the refrigerating room
Defrost the freezing room
<Defrosting | Failure>
Check history of R/F defrosting
<Checking | Failure>
In initial behavior <Defrosting history | Exist>
<Defrosting history | Not exist>
<Checking | Failure>
Check elapsed time of freezing room
Count elapsed time of freezing room
<F-Timer | Expire (5.0h)>
<Checking | Failure>
Check elapsed time of refrigerating room
<R-Timer | Expire (5.0h)>
Defrost the refrigerating room
Defrost the freezing room

(a) Top Layer

(b) 2nd Layer

Fig. 2. Use case analysis

legal behavior, but the other arrow —● shows an unusual or illegal behavior. The engineers then described a detailed description. For example, the activity "Check condition for defrosting" in Figure 2(a) is extended into the detailed description shown in Figure 2(b) using the functional specification.

At Step-3, the illegal behaviors of the defroster are extracted by tracing the activity chart shown in Figure 2. The guide words, prepared for the experiment, 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 not activated." These guide words are extensively used in Step-4.

At Step-4, by tracing the activity chart in Figure 2, the software fault tree shown in Figure 3 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 nodes "Failure in checking condition for defrosting," "Failure in defrosting freezing room" and "Failure in defrosting refrigerating room" are chosen as the next level nodes of the software fault tree. Then, for the "Failure in checking condition for defrosting," the nodes "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 the deviation analysis, each node is expanded to some nodes successively.

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 3, the test items

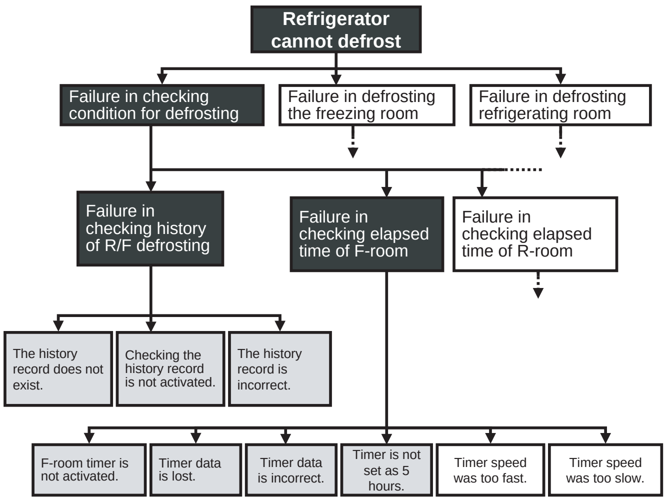

Refrigerator cannot defrost
Failure in checking condition for defrosting
Failure in defrosting the freezing room
Failure in defrosting refrigerating room
Failure in checking history of R/F defrosting
Failure in checking elapsed time of F-room
Failure in checking elapsed time of R-room
The history record does not exist.
Checking the history record is not activated.
The history record is incorrect.
F-room timer is not activated.
Timer data is lost.
Timer data is incorrect.
Timer is not set as 5 hours.
Timer speed was too fast.
Timer speed was too slow.

Fig. 3. Software fault tree

A1.1 to A3.4 shown in Figure 4 are generated.

4. Experimental Evaluation

The following were concluded from an experimental application of the proposed method to the refrigerator control software.

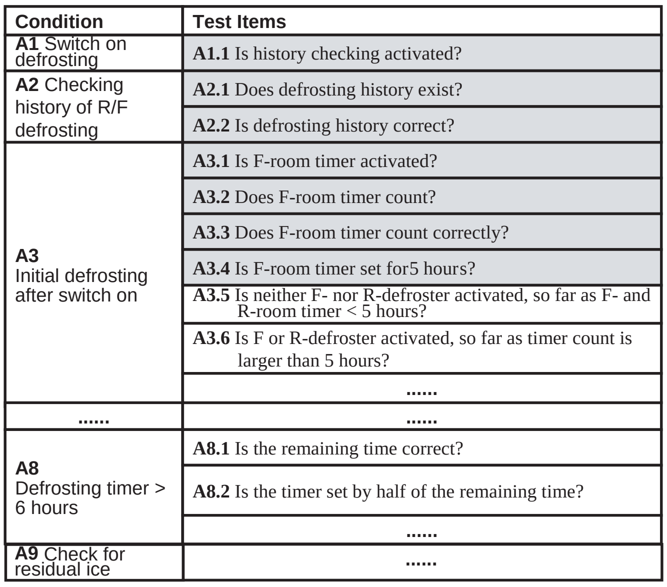
(1) With relatively a few additional effort, we can generate test items suitable to check behaviors (especially, illegal behaviors) of the target system.

(2) The proposed method generates test items in more detailed descriptions, which reflects and thus corresponds 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).

5. Discussion

In software testing, checking of such illegal behaviors that were overlooked at test design needs a lot of extra costs. Generally speaking, essential functions of the products would be occasionally overlooked when user's viewpoint comes first. But, since the proposed method is currently developed only for embedded software in electrical appliances and the fault tree analysis is applied at the final stage of the proposed method, such defects may be avoided and test items generated by



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	......

Fig. 4. Generated test items

the proposed method cover all essential functions. Thus, as a result, we can expect test costs might be shortened compared with the one by conventional method.

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

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

[5] C. Smidts and D. Sova, "An Architectural Model for Software Reliability Quantification: Sources of Data," Reliability Engineering and System Safety, vol.64, 279–290, 1999.

[6] 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.