法政大学学術機関リポジトリ
HOSEI UNIVERSITY REPOSITORY

PDF issue: 2024-07-28

Assessment and Application of a ”Functional Scenario-Based” Test Case Generation Method

LI, Cen Cen

(出版者 / Publisher)
法政大学大学院情報科学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編 / 法政大学大学院紀要. 情報科学研究科編
(巻 / Volume)
8
(開始ページ / Start Page)
55
(終了ページ / End Page)
60
(発行年 / Year)
2013-03
(URL)
https://doi.org/10.15002/00009544

Assessment and Application of a "Functional Scenario-based" Test Case Generation Method

Cencen Li
Graduate School of Computer and Information Sciences
Hosei University
Email: cencen.li.js@stu.hosei.ac.jp

Abstract—**Specification-based testing enables us to detect errors in the implementation of functions defined in given specifications. Its effectiveness in achieving high path coverage and efficiency in generating test cases are always major concerns of testers. The automatic test case generation approach based on formal specifications proposed by Liu and Nakajima is aimed at ensuring high effectiveness and efficiency, but this approach has not been formally assessed and used in practical testing environments. In this paper, we first statically analyze the characteristics of the test case generation approach, and then perform experiments to use this approach in two different real testing environments. The two practical testing cases include a unit testing and an integration testing. We perform the testing not only for assessing Liu's approach in practice, but also trying to get some experience of using this approach in practice. The static analysis and the results of experiments indicate that this test case generation approach may not be effective in some circumstances, especially in integration testing. We discuss the results, analyze the specific causes for the ineffectiveness, namely the low path coverage, and propose some suggestions for improvement.**

I. Introduction

Specification-based testing enables us to detect errors in the implementation of the functions defined in given specifications. Since performing specification-based testing is usually time consuming, automatic specification-based testing is always attractive to the software industry. An automatic test cases generation approach based on formal specification known as *functional scenario-based testing*(FSBT) was first introduced in Liu's paper [1], which is aimed to ensure high effectiveness and efficiency of specification-based testing.

This automatic specification-based testing approach includes an improved test strategy over the commonly used disjunctive normal form strategy [2], and a decompositional method for automatic test case generation. The approach is applicable to any operation specified in terms of pre- and post-conditions. The essence of the test strategy is to guarantee that every functional scenario defined in a specification is implemented "correctly" by the corresponding program. A functional scenario of an operation defines an independent relation between its input and output under a certain condition, and usually expressed as a predicate expression. The predicate expression is used as a foundation of automatic test case generation algorithm. The function defined by a functional scenario is actually a function of the software system, and it should be implemented in the program. Therefore, by using the test cases derived from functional scenarios, the implementation of functions defined in scenarios are expected to be tested consequently.

Although the automatic testing approach proposed by Liu is interesting in theory, it has not been empirically assessed and used in practice. In this paper, we first statically analyze the characteristics of this approach, specifically the relations between the functional scenarios and execution paths. And then carry out several experiments to apply this approach in real testing environments and assess it by measuring the coverage of execution paths. We specify the formal specifications, implement the program based on the specifications, perform testing for the program, and count how many parts of the implementation program can be tested. The result indicates that this testing approach is unlikely to be sufficient for testing all parts of programs. The ineffectiveness is caused by the drawback of the specification-based testing and is hardly overcome if test cases are generated without analyzing the structure of program. But the effectiveness can be improved by considering the relation between the formal specification and the corresponding program when test cases are generated.

Most of the improvement proposed by us is based on the analysis of the relations between specification and program, specifically the relations between the functional scenarios and execution paths. An execution path is the implementation of the function defined by a functional scenario. As described in details in Section IV, we statically analyze these relations in general to ensure that the improvement can be used generally. Since some specific causes of ineffectiveness of the testing approach are associated with the specification itself, we propose some suggestions that can be adopted if the target system of testing is specified in the same specification language or has the similar structure. The formal specification used in our experiment is specified in SOFL (Structured Object-oriented Formal Language)[3] and the program is implemented in Java.

The remainder of this paper is organized as follows. We first introduce some related work in Section II. Section III includes brief introduction of the original decompositional testing approach, including the test strategy and the test case generation algorithm. In Section IV we describe the relations between functional scenarios and execution paths in the program, which are two basic concepts in our experiment. The experiment will be introduced in Section V including purpose, environment,

Supervisor: Prof. Shaoying Liu

result and result analysis. We will propose the new criteria for testing strategy in Section VI, and we will also express the test results of the extended approach in this section. In Section VII we conclude the paper and point out future research directions.

II. Related Work

Specification-based testing methods have been well researched based on different specification techniques. The test case generation method [1], which underlies our experiment is applicable to any operation specified in terms of pre- and post-conditions.In this section, we introduce some testing approaches and test case generation methods that are similar to our functional scenario-based approach.

The test case generation method based on algebraic specifications is introduced in [4], and the method of generating test case from reactive system specification is described in [5]. Cheon et al [6] use the assertions derived from formal specification in Object Constraint Language (OCL) as test oracles, and combine random tesing and OCL to carrying out automated testing for Java program. Michlmayr et al. introduce a framework of performing unit testing of publish/subscribe applications based on LTL specification in [7]. Bandyopadhyay et al. [8] improve the existing test input generation method based on sequence diagrams of UML specification by considering the effects of the messages on the states of the participating objects.

Some approaches are proposed to enhance the effectiveness of the specification-based testing. Fraser et al. [9] investigate the effects of the test case length on the test result. Based on their experiments of specification based testing for reactive systems, they find a long test case can achieve higher coverage and fault detecting capability than a short one. They intend to improve the effectiveness of specification-based testing by changing the length of test case. In [10], Liu et al. propose a technique called "Vibration" method to ensure all of the representative program paths of the program are traversed by the test cases generated from formal specification. This method provides an effective way of specification-based test case generation.

III. Basic Concepts

A. Test Case Generation Algorithm

In order to explain the test case generation algorithm, we need to define the formal specification and functional scenario first. For simplicity, let $S(S_{iv}, S_{ov})[S_{pre}, S_{post}]$ denote the formal specification of an operation S, where S_{iv} is the set of all input variables, S_{ov} is the set of all output variables, and S_{pre} and S_{post} are the pre and post-condition of S, respectively. For the post-condition S_{post}, let $S_{post} \equiv (C_1 \wedge D_1) \vee (C_2 \wedge D_2) \vee ... \vee (C_n \wedge D_n)$, where each $C_i (i \in \{1, ..., n\})$ is a predicate called a "*guard condition*" that contains no output variable in S_{ov} and $\forall_{i,j \in \{1,...,n\}} \cdot i \neq j \Rightarrow C_i \wedge C_j = false$; D_i a "*defining condition*" that contains at least one output variable in S_{ov} but no guard condition. Then, a formal specification of an option can be expressed as a disjunction expression $(\sim S_{pre} \wedge C_1 \wedge D_1) \vee (\sim S_{pre} \wedge C_2 \wedge D_2) \vee ... \vee (\sim S_{pre} \wedge C_n \wedge D_n)$. A conjunction $\sim S_{pre} \wedge C_i \wedge D_i$ is realized as a functional scenario. We treat a conjunction $\sim S_{pre} \wedge C_i \wedge D_i$ as a functional scenario because it defines an independent behavior: when $\sim S_{pre} \wedge C_i$ is satisfied by the initial state (or input variables), the final state (or the output variables) is defined by the defining condition D_i.

TABLE I
Test Cases Generation Algorithm

No. of Algorithms	$\ominus$	Algorithms of test case generation for x_1
1	$=$	$x_1 = E$
2	$>$	$x_1 = E + \Delta x$
3	$<$	$x_1 = E - \Delta x$
4	$\leq, \geq, \neq$	similar to above

Since test case generation usually depends on the precondition and guard condition, and the defining condition D_i usually does not provide the main information for test case generation. The defining condition D_i is eliminated from the functional scenario. The conjunction after eliminating defining condition is $\sim S_{pre} \wedge C_i$, called *testing condition*. For each atomic predicate Q in testing condition, the input variables involved in each atomic predicate expression Q can be generated by using an algorithm that deals with the following three situations, respectively.

- **Situation 1:** If only one input variable is involved and Q has the format $x_1 \ominus E$, where $\ominus \in \{=, <, >, \leq, \geq, \neq\}$ is a relational operator and E is a constant expression, using the algorithms listed in Table I to generate test cases for variable x_1.
- **Situation 2:** If only one input variable is involved and Q has the format $E_1 \ominus E_2$, where E_1 and E_2 are both arithmetic expressions which may involve variable x_1, it is first transformed to the format $x_1 \ominus E$. And then apply Criterion 1.
- **Situation 3:** If more than one input variables are involved and Q has the format $E_1 \ominus E_2$, where E_1 and E_2 are both arithmetic expressions possibly involving all the variables $x_1, x_2, ..., x_w$. First randomly assigning values from appropriate types to the input variables $x_2, x_3, ..., x_w$ to transform the format into the format $E_1 \ominus E_2$, and then apply Situation 2.

Note that if one input variable x appears in more than one atomic predicate expressions, it needs to satisfy all the expressions which it is involved in.

B. Formal specification Language

The formal specification used in our experiment is written in SOFL. SOFL is one of the formal specification languages that specifies operations in terms of pre- and post-conditions. In principle, the assessment and the improvement are not dependent on specific specification language, but we need a specific one to specify the target system in our experiments and to express the improvement.

Due to using mathematical notations, SOFL specifications are precise. The structure of SOFL specification has its own

```
module ICcardSystem
type
Date = composed of
            year: nat
            month: nat
            day: nat
        end
ICcardInfo = composed of
                 ...
                 activatedDate: Date
                 monthlyExpireDate: Date
             end
var
ext _CurrentDate: Date
ext _ICcardBase: set of ICcardInfo

inv
∀iccardInfo is_T(ICcardInfo) ·
isBefore(ICcardInfo.activedDate, ICcardInfo.monthlyExpireDate)
```

Fig. 1. Example of specification and invariant

characteristics, note that the following concepts of SOFL specification involved in our experiment and following discussion.

- **process:** A process in SOFL specification defines an independent operation. It includes a list of input variables, a list of output variables, pre-condition, and post-condition. In the following of this paper we use term "process" replacing "operation" to keep consistent with SOFL specification.
- **module:** A module is an assemblage of processes, and each process can be decomposed to create a new module including a group of lower level decomposed processes.
- **external variable:** External variables are variables belonging to the whole specification, and all of the processes in the specification can use external variables without listing them in the input variables list.
- **invariant:** An invariant is a predicate, it expresses a property of types and variables. The invariant must be sustained throughout the entire specification.

For example, Figure 1 shows a part of "module" definition used in our experiments. It is . The words in boldface are reserved words in SOFL, under the key word "***ext***" is the definition of external variables, and the key word "***inv***" indicates that the following predicate is an invariant and should be satisfied throughout the entire specification.

IV. Static Analysis

The objective of program testing is to test all parts of the program, to achieve this target need all execution paths in the program be executed at least once. The execution path presents an sequence of statements from the start state of the program to the termination. For any set of values of input variables, an execution path must exist to process the input data. For the given values of set of input variables, there is an unique execution path in program, and it expresses one specific function of software system. Therefore, we can guarantee that all parts of the program are tested if all the executable paths are executed. Since our major concern in the experiment is the effectiveness of the testing approach or how many parts in the program of target system can be tested, we use the coverage of the executable paths to measure the effectiveness of the testing approach.

Based on the definition of functional scenario, an execution path can be realized as an implementation of a functional scenario. Theoretically, one functional scenario should correspond to one and only one execution path if the program is implemented by following the formal specification exactly. But in practice, the relation between functional scenario and execution path may not be a one-to-one correspondence. In order to figure out how the test cases derived from a functional scenario influence the coverage of execution paths, we define that a scenario and a execution path have relation to each other if all of the test cases derived from the scenario can be accepted by the execution path. Although this definition is not sufficient to describe various relations between scenarios and paths, it is enough for the purpose of our experiment. The summary of the relations between functional scenarios and execution paths are listed in the following.

- **One Scenario to No Path:** If the function defined by the functional scenario is not implemented in the program or implemented incorrectly, there is no path being executed by applying the test cases derived from the scenario. It may be caused by the programmer misunderstanding the specification, or mistake made by the programmer during programming.
- **One Scenario to One Path:** This is the ideal situation, the program is implemented according to the specification exactly. No refinement is made in the specification or program.
- **One Scenario to Multi Paths:** This is the most common situation. It is usually caused by the refinement, which may occur in specification or program. Since some specifiers use top-down approach when they specify specifications, they will define the more general or more abstract process with less details first, and then decompose the process into more than one lower level process with more details. When we try to find execution paths for the functional scenarios extracted from a more abstract level specification, it is possible to find more than one paths corresponding to one specific scenario if the program is implemented based on the lower level specification. If the information in lower level specification is not considered in test cases generating process, some of the relative paths will not be tested. Another kind of refinement occurs in the program. It is usually made by the programmer for different reasons, like improving the effectiveness of program, complying with the special programming rules, etc.

- **Multi Scenario to One Path:** The relation multi scenarios to one path is a reverse relation of one scenario to multi paths, it usually occurs when programmer abstracts some functions defined in the specification. The best reason for programmer to abstract the function is to simplify the program.
- **Paths to No Scenario:** This situation is very common in practice, but unfortunately it will often be ignored in specification-based testing. According to the concept of specification-based testing, the process of testing is on the basis of what the specification says. But, in the real testing environment, even all of the execution paths which implement all of the defined functions are tested, it does not mean that all parts of the program have been tested. The most possible reason of the occurrence of this kind of relation is the incompleteness of specification. The incompleteness can be caused by lacking ideas or limitation of time, etc. But, in the meantime, the programmer may try to, or have to, handle some exceptions or add some functions undefined in the specification. One specific case is that the program needs to process the input variables even the values of the input do not evaluate the predicates of the scenarios to be true. This is because the specification just defines what kind of inputs can be handled while the program must respond to all of the possible inputs. Usually we think these kind of paths *relative to process*.

V. Experiments

In this section, we present two experiments. Each experiment performs a testing by applying the functional scenario-based test case generation method. One of the experiments is unit testing, and the other is an integration testing. Carrying out these two testing experiments is not aim to compare the effectiveness of the test cases generation method among specific cases, but to assess the effectiveness of the method under different test environments and attain some experience in using the method in practice.

In our cases, if the functional scenarios used to generate test cases are from a lower level module, the testing can be realized as a "unit testing". On the contrary, if the functional scenarios used to generate test cases are from a higher level module, we consider the testing as an "integration testing". Different researchers may have different understanding under their viewpoint. Here we just use the concepts of unit testing and integration testing to different two kind of testing cases, one is based on the relatively higher level specification and the other one is based on the relatively lower level specification.

A. Target Systems

1) Income Tax Calculation System: The Income Tax Calculation system is the target system of the unit testing. This system is aim to help tax payers to calculate their amount of tax. According to [11], the tax payers are divided into two categories based on the type of their income. The two categories of tax payers use different formulas to calculate the amount of tax, but they have the similar process. The tax payer first calculates his or her "amount of income", and then calculates the"deduction of income". Finally, the "amount of tax" is calculated based on the the difference between "amount of taxable income" and "deduction of income".

The specifications of the Income Tax Calculation system are separated into three levels. The top level, or the first level, specification contains 2 processes. Each process, which presents a tax calculation process for one category of tax payers, is decomposed into a module containing 3 processes in the second level specification. These 3 processes in each second level module correspond to the three steps in tax amount calculation mentioned previously. Each process in the second level modules is decomposed to construct the lowest level, or the third level, specification. There are 61 processes contained in the lowest level specification. And totally 69 processes are defined in all three level specifications, which are formally specified in SOFL. The implementation of the system is developed by using Java under the Eclipse environment. The implementation program contains 15 classes, and more than 2500 lines code.

2) IC Card System: The target system of the integration testing generating test cases from specification directly is an IC card system. The IC card can be used to take the public transportations, and it associates with a bank account. Since card holders can use the card without authority, the maximum amount that can be deposited in the IC card is limited to prevent the potential economic loss from losing the card. Customers can swipe the cards to take transportation or use the cards to buy train tickets. If the balance in card is not enough, the customers can reload by cash or from associated banks account. The customers can also transfer the money back to the bank accounts, but they can not get cash from the IC cards directly. For the customers who need commute, they can set monthly payment for one route to get discounts.

We design and define 6 processes in the top level module. This module presents the most abstract definition of the IC card system, and each of the 6 processes represents a function of the system described previously. Based on the top-down concept, we decompose each of these processes for further defining. There are total 12 lower level processes defined in the specification and some of them are reused to construct higher level processes. All of these 18 processes are specified formally by using SOFL, and the implementation program contains 14 classes, 2200 lines code.

B. Unit Testing

The Income Tax Calculation system is the target system of the unit testing. To perform the testing, we derive test cases from the processes defined in the lowest level specification. Totally 29 processes derived from the same process in the first level specification are used to generated test cases for testing their corresponding program units. Total 615 test cases are generated from the 207 functional scenarios, which are extracted from the 29 processes in the testing. The target program units contain 223 execution paths, and 211 paths are tested

TABLE II
EXCERPT OF UNIT TESTING RESULT

No.	Process Name	Number of Scenario	Relative Paths	Test Cases	Tested Paths	Coverage(%)
1	IFDSTAT_A	2	2	8	2	100
2	EI_A	11	12	35	11	91
3	MI_A	20	24	63	20	83
4	OI_A	2	2	8	2	100
5	T_A	1	3	1	1	100

TABLE III
INTEGRATION TESTING RESULT

No.	Process Name	Number of Scenario	Relative Paths	Test Cases	Tested Paths	Coverage(%)
1	RailwayTravel	5	12	35	6	50
2	PurchaseTickets	2	3	13	2	67
3	ReloadByCash	3	5	17	3	60
4	SetMonthlyPayment	2	78	77	28	36
5	ReloadFromAccount	3	7	34	3	43
6	TransferToAccount	2	7	16	2	29

by applying the 615 test cases. The average path coverage is approximately 94 percent. The excerpt of the testing results are listed in Table II. Item "***Number of Scenario***" identifies the scenario in a process; "***Relative Paths***" shows how many execution paths in program have relations with the scenarios; "***Test Cases***" denotes how many test cases are generated from this scenario and "***Tested Paths***" indicates how many paths are tested; "***Coverage***" shows the coverage of executable paths.

C. Integration Testing

The target system of the integration testing is the IC card system. Only the functional scenarios extracted from the 6 processes in the top level are used to generate test cases. There are 112 execution paths in the program correspond to the functions defined in the 6 processes. Total 192 test cases are generated from the 17 functional scenarios in the testing, and the brief statistics of the results are listed in Table III. The details of this integration testing can be found in [12].

D. Results Analysis

The result of the experiment indicates that the functional scenario-based test cases generation method is more effective if the test cases are generated based on relatively lower level specification. But the data in Table II shows that even the test cases are derived from the lowest level specification, some execution paths still cannot be tested. We think the ineffectiveness is caused by two reasons. The first reason is that the specification is not well defined, and the second reason is the refinement in program. These two reasons are also the causes of the relation "one scenario to multi paths" and "paths to no scenarios" mentioned in Section IV. The results confirms our analysis that the existence of these two relations usually reduces the rate of testing coverage.

Comparing to the unit testing, the situation faced by integration testing is more complex. The reason is that the test cases used in the testing are generated based on relatively higher level specification. Therefore, the ineffectiveness may be caused by the refinement in specification. In that case, the coverage of paths can be improved by considering the lower level specification in test cases generation process.

VI. IMPROVEMENT PROPOSALS

Based on the previous analysis, the ineffectiveness is usually caused by the appearance of the relation "paths to no scenario" and "paths to multi scenario". To test the execution paths in relation to "paths to no scenario", we should derive test cases from the functions undefined in specification. Based on the concept of FSBT, test cases derived from one defined function are actually derived from the *testing condition* $\sim S_{pre} \wedge C_i$, and the test cases generated from the scenario can be presented as $G(\sim S_{pre} \wedge C_i)$. Here we use $G(p)$ to denotes the set of test cases derived from predicate p. In order to derive test cases for undefined function, we use **Proposal 1** to extend the contents of the set of test cases derived from one defined function.

Proposal 1: Let $\sim S_{pre} \wedge C_i \wedge D_i$ be a functional scenario in specification, extend set of test cases $G(\sim S_{pre} \wedge C_i)$ into $G((\sim S_{pre} \wedge C_i) \vee \neg(\sim S_{pre} \wedge C_i)) = G(\sim S_{pre} \wedge C_i) \cup G(\neg(\sim S_{pre} \wedge C_i))$

We use predicate $\neg(\sim S_{pre} \wedge C_i)$ to present the functions undefined in the scenario $\sim S_{pre} \wedge C_i \wedge D_i$. The test cases derived from this predicate can be used to test the execution paths implementing functions that are not defined in the scenario. Note that this predicate can be constructed into a predicate expression in which the conjunction clauses may be the *testing condition* of other functional scenario in the same operation. And this will result in that the test cases derived from the predicate may satisfy other scenarios. Although it is possible to generate test cases from the same *testing condition* more than once, the duplication of generation do not affect the test coverage.

In order to test the paths in relation "one scenario to multi paths", we must handle the problems causing this relation.

Two causes of the occurrence of this relation are refinement in program and refinement in specification. To handle the first cause we must analyse the structure of the program. The second cause indicates that the functional scenario used to generate test cases may be defined in a higher level specification but the program is implemented based on the refined lower level specification. Therefore the test cases generation process should consider the refined specification, as reflected in **Proposal 2**.

Proposal 2: Let $F(x)$ be the disjunction of functional scenarios that contain input variable x. And let $F'(x)$ be the disjunction of functional scenarios which are in the decomposed module containing x. The test cases generated from the scenarios in higher level module should satisfy the condition: $\forall T_c' \in G(F'(x)) \cdot \exists T_c \in G(F(x)) \cdot T_c(x) = T_c'(x)$.

The notation T_c in **Criterion 2** denotes one test case of high level specification while the notation T_c' indicates one test case of lower level specification. $T_c(x)$ and $T_c'(x)$ present the value of input variable x in the test case T_c and T_c' respectively. The criterion requires all of the values of x generated from lower level specification should be contained in the test cases derived from higher level specification. Since the values of x is generated from refined specification, the test cases containing all of these values can be realized as generated by considering refined specification.

In addition to the factors affecting the effectiveness of testing, we also find that some test cases generated from functional scenarios are invalid. By invalid we mean these test cases are not satisfied with invariants, which are a predicate need to be sustained throughout the entire specification. The invalid test cases exist because the test cases are generated without considering invariant. To avoid generating invalid test cases, **Proposal 3** can be applied in the test case generation process.

Proposal 3: Let an invariant on type T be $I_t = \forall t \in T \cdot Q(t, w)$. Then replace $G(\sim S_{pre} \wedge C_i)$ with $G(\sim S_{pre} \wedge C_i \wedge \forall x \in S_{iv} \cdot is_T(x) \Rightarrow Q(t, w)[x/t])$.

In the former predicate, x stands for the variables of scenario which are in type T. The criterion requires that if the scenario has input variables in type T, the test cases generated from the scenario should satisfy the invariant. So that we can avoid generating invalid test cases violating the invariant.

Finally, we perform the experiments again with the three proposal. For the sake of space, we just present the comparison of integration testing in Figure 2. The y-axis presents the coverage and the x-axis indicates each process involved in the experiments. Obviously, the effectiveness is improved by using our proposal.

VII. Conclusions and Future Work

In this paper, we performed two experiments to assess the functional scenario-based test cases generation method. Based on the static analysis and test results we find that this method is effective when the specification is well defined, but it may be ineffective if the specification is not well defined. We propose some improvement when the higher level specification are

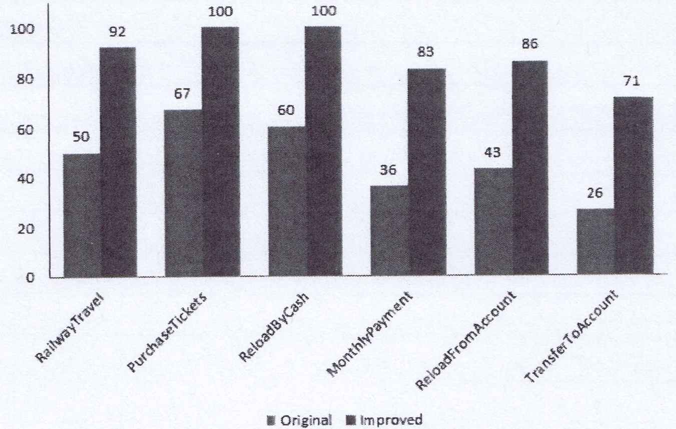

Fig. 2. Result of Improved FSBT

used in test case generation to ensure more execution paths can be tested. The final results show that our proposal can improve the effectiveness of the testing method. In the future, we intend to use a large-scale system to assess the test cases generation method and the proposals farther, and build a software tool to implement the testing method with our proposals.

References

[1] Shaoying Liu, and Shin Nakajima. A Decompositional Approach to Automatic Test Case Generation Based on Formal Specification. Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement, pages 147-155, 2010.

[2] J. Dick, and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-based Specifications. In Proceedings of FME '93: Industrial-Strength Formal Methods, pages 268-284, Odense, Denmark, 1993. Springer-Verlag Lecture Notes in Computer Science Volume 670.

[3] Shaoying Liu. Formal Engineering for Industrial Software Development Using the SOFL Method. Springer-Verlag, ISBN 3-540-20602-7, 2004.

[4] M. C. Gaudel and P. Le Gall. Testing Data Types Implementation from Algebraic Specifications. In R. Hierons, J. Bowen, and M. Harman, editors, Formal Methods and Testing, pages 209-239. LNCS 4949, Springer-Verlag, 2008.

[5] M. Broy, B. Jonsson, J. -P. Katoen, M. Leucker, and A. Pretschner (eds.). Model-based Testing of Reactive Systems. LNCS 3472, Springer-Verlag, 2005.

[6] Yoonsik Cheon, and Carmen Avila. Automating Java Program Testing Using OCL and AspectJ. 7th International Conference on Information Technology, pages 1020-1025, 2010.

[7] Anton Michlmayr, Pascal Fenkam, and Schahram Dustdar. Specification-Based Unit Testing of Publich/Subscribe Applications. Proceedings of the 26th IEEE International Conference on Distributed Computing Systems Workshops, pages 34-34, 2006.

[8] Aritra Bandyopadhyay, and Sudipto Ghosh. Test Input Generation using UML Sequence and State Machines Models. International Conference on Software Testing Verification and Validation, pages 121-130, 2009.

[9] Gordon Fraser, and Angelo Gargantini. Experiments on the Test Case Length in Specification Based Test Case Generation. ICSE Workshop on Automation of Software Test, pages 18-26, 2009.

[10] Shaoying Liu, and Shin Nakajima. A "Vibration" Method for Automatically Generating Test Cases Based on Formal Specifications. 18th Asia-Pacific Software Engineering Conference, pages 5-8, 2011.

[11] http://www.nta.go.jp/tetsuzuki/shinkoku/shotoku/tebiki2010/pdf/43.pdf

[12] Cencen Li,Shangying Liu, and Shin Nakajima. An Experiment for Assessment of a "Functional Scenario-based" Test Case Generation Method. Proceedings of International Conference on Software Engineering and Technology, pages 64-71, 2012.