International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:6, No:4, 2012

An Experiment for Assessment of a "Functional Scenario-based" Test Case Generation Method

Cencen Li, Shaoying Liu, and Shin Nakajima

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 cases 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 empirically assessed. In this paper, we present an experiment for assessing Liu's testing approach. The result indicates that this testing approach may not be effective in some circumstances. We discuss the result, analyse the specific causes for the ineffectiveness, and describe some suggestions for improvement.

Keywords—experiment, functional scenario, specification-based, testing.

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 in practice. In this paper, we design an experiment to apply this approach in a real testing environment and assess it by measuring the coverage of execution paths. We specify the formal specification, implement the program based on the specification, 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 analysing 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 III, we analyse 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. Section II includes brief introduction of the original decompositional testing approach, including the test strategy and the test case generation algorithm. In Section III we describe the relations between functional scenarios and execution paths in program, which are two basic concepts in our experiment. The experiment will be introduced in Section IV including purpose, environment, results and results analysis. We will propose the new criteria for test strategy in Section V, and we will also express the test results of the extended approach in this section. Section VI is related work, and in Section VII we conclude the paper and point out future research directions.

Cencen Li is with the Faculty of Computer and Information Science, Hosei University, Tokyo, 184-8584, Japan. E-mail: cencen.li.js@stu.hosei.ac.jp.

Shaoying Liu is with the Software Engineering Laboratory for Dependable Systems Department of Computer Science, Faculty of Computer and Information Sciences Hosei University, Tokyo, 184-8584, Japan. E-mail: sliu@hosei.ac.jp.

Shin Nakajima is with Information Systems Architecture Science Research Division, National Institute of Informatics, Tokyo, 101-8430, Japan. E-mail: nkjm@nii.ac.jp.

II. Introduction to Decompositional Approach to Automatic Test Case Generation

A. Test Strategy

As mentioned previously, the essence of the test strategy is to ensure that every functional scenario defined in a specification is implemented "correctly" by the corresponding program. In order to explain the test strategy precisely, 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 whose values are not changed by the operation, S_{ov} is the set of all output variables whose values are produced or updated by the operation, 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. Note that we use $\sim x$ and x to represent the initial value before the operation and the final value after the operation of external variable x, respectively. The decorated pre-condition $\sim S_{pre} = S_{pre}[\sim x/x]$ denotes the predicate resulting from substituting the initial state $\sim x$ for the final state x in pre-condition S_{pre}. 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.

When performing testing, the tester must generate test cases for each functional scenario. As we have mentioned that a functional scenario expresses an independent function of an operation, generating test cases for each scenario can guarantee all of the functions defined in the specification are tested. Since one test case may not enough to test one function efficiently, the tester is required to generate more test cases for each scenario. The automatic test generation algorithm will make this process effective.

B. Test Case Generation Algorithm

As mentioned in the previous subsection, a functional scenario is expressed as a conjunction $\sim S_{pre} \wedge C_i \wedge D_i$. To generate test cases from this scenario, it must be decomposed first. The decomposing process is divided into following two steps:

- **Step 1: Eliminate Defining condition.** The defining condition D_i is eliminated first since the execution of program only requires input values. Test cases generation depends on the pre-condition and guard condition, and defining condition usually do not provide the main information for test case generation. The conjunction after eliminating defining condition is $\sim S_{pre} \wedge C_i$, called *testing condition*.

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

- **Step 2: Convert to disjunctive normal form.** The remainder of scenario is translated into an equivalent disjunctive normal form (DNF) with form $P_1 \vee P_2 \vee ... \vee P_n$. A P_i is a conjunction of atomic predicate expressions, say $Q_i^1 \wedge Q_i^2 \wedge ... \wedge Q_i^m$.

Let $Q(x_1, x_2, ..., x_w)$ be one of the atomic predicate expressions $Q_i^1, Q_i^2, ..., Q_i^m$ mentioned previously. The variables $x_1, x_2, ..., x_w$ is a sub set of all the input variables. Test cases for the input variables involved in each atomic predicate expression Q can be generated using an algorithm that deals with the following three situations, respectively. Here we are using variables of numerical types as examples for convenience.

- **Situation 1:** If only one input variable is involved and $Q(x_1)$ 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 $Q(x_1)$ 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 $Q(x_1, x_2, ..., x_w)$ 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 Criterion 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.

C. 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 experiment and to express the improvement.

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

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

inv
∀iccardInfo ∈ _ICcardBase · is_ ICcardInfo (iccardInfo) ∧
isBefore(iccardInfo.actived Date, iccardInfo.monthlyExpireDate)
```

Fig. 1: Example of specification and invariant

"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 fragment of specification of the target system in our experiment. It is an IC card system, card holders can take public transportations and set monthly payment. In this specification fragment, we defined a data type named "*ICcardInfo*" saving information of each card. The words in boldface are reserved words in SOFL, under the key word "***ext***" is the definition of external variables. "*_CurrentDate*" is an external variable in type *Date*, and it presents the current date; "*_ICcardBase*" is a set of variables which are in type *ICcardInfo*, and it contains all of the IC card information in the system. The key word "***inv***" indicates that the following predicate is an invariant and should be satisfied throughout the entire specification. "***is_T***" is a build-in function checking whether the variable in the following brackets is in type ***T***. ***T*** stands for a specific type, and in this case the type is *ICcardInfo*. The "***isBefore***" is a function defined by user to check if one date is before another. The invariant here requires that all of the variables in *_ICcardBase* should be in type *ICcardInfo* and date on which the IC card is activated must be before the date monthly payment of the card expires.

```
process sort (array: Array )
pre ∀i ∈ inds(array) · is_int (array[i]) ∧ array.length <= 1000
post   array.length <= 100 ∧ is_bubbleSort(array) ∧ ∀i, y ∈ inds(array)
∧ i <= y· array[i] <= array[y]
∨
array.length > 100 ∧ is_insertionSort(array) ∧ ∀i, y ∈ inds(array) ∧ i <=
y· array[i] <= array[y]

process is_bubbleSort(array: Array) var: bool
/*sorting the array using bubble sort algorithm and return true*/

process is_insertionSort(array: Array) var: bool
/*sorting the array using insertion sort algorithm and return true*/
```

Fig. 2: Specification of Sort an Array

III. Functional Scenario and Execution Path

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 Table II.

A. 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

TABLE II: Test Cases Generation Algorithm

No. of relations	Notation	Discription	Main Reasons
1	1...0	One Scenario to No Path	Implementing incorrectly
2	1...1	One Scenario to One Path	Implementing correctly, based on the specification exactly
3	$1...1^+$	One Scenario to Multi Paths	Refinement by specifiers or programmers
4	$1^+...1$	Multi Scenarios to One Path	Abstraction by programmers in program or incorrect implementation
5	0...1*	Paths to No Scenario	Incompletement of specification or implementing incorrectly

```
1  public void sort(double[] array){
2      int index = 1;
3
4      while(index < array.length){
5          for(int i = 0; i < index; i++){
6              if(array[i] > array[index]){
7                  double temp = array[index];
8                  array[index] = array[i];
9                  array[i] = temp;
10             }
11         }
12         index++;
13     }
14 }
```

Fig. 3: Example program of "One Scenario to No Path"

```
1  public void sort(int[] array){
2      if (array.length <= 100){
3          bubbleSort(array);
4      } else if (array.length > 100 &&
5                          array.length <= 1000){
6          insertSort(array);
7      }
8  }
9
10 public void bubbleSort(int[] array){
11
12 public void insertSort(int[] array){
```

Fig. 4: Example program of "One Scenario to One Path"

programmer during programming. Figure 2 shows the specification of process "***sort***", which defines an operation of sorting items of an array in ascending order. "***is_bubbleSort***" and "***is_insertionSort***" are two processes defined by user, they sort an array by using bubble sort algorithm and insertion sort algorithm respectively. Two functional scenario exists in the process shown in Figure 2:

1) $\forall i \in inds(array) \cdot is\_int(array[i]) \wedge array.length \leq 1000 \wedge array.length \leq 100 \wedge is\_bubbleSort(array) \wedge \forall i, y \in inds(array) \wedge i \leq y \cdot array[i] \leq array[j]$
2) $\forall i \in inds(array) \cdot is\_int(array[i]) \wedge array.length \leq 1000 \wedge array.length > 100 \wedge is\_insertionSort(array) \wedge \forall i, y \in inds(array) \wedge i \leq y \cdot array[i] \leq array[j]$

Figure 3 shows the implementations of process "***sort***", and just one path exists in the program. Since the specification requires an array with items in "***int***" as input variable while the program accepts array with items in "***double***" as input, the implementation is incorrect obviously. In this case, the test cases derived from either of the scenarios will not be accepted by the path, and neither of the functions defined in the two scenarios has corresponding execution path in the program.

B. One Scenario to One Path

This is the ideal situation, the program is implemented according to the specification exactly. Figure 4 illustrates the implementation in which each functional scenario in process "***sort***" has one and only one relative execution path.

C. One Scenario to Multi Paths

This situation always happens in real software development projects. 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 by using FSBT.

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. Figure 5 shows another possible implementation of the specification shown in Figure 2. In this case, the programmer makes three different methods to sort the array with different length other than two in specification. The test cases derived from the second scenario can be accepted by two execution paths in program.

D. Multi Scenario to One Path

The relation multi scenario to one path is a reverse relation of one scenario to multi paths, it usually occurs when programmer abstracts some functions defined in specification. The best reason for programmer to abstract the function is to simplify the program, figure 6 shows a different implementation of process "***sort***" which uses a plain way to handle the sorting instead of invoking two external methods. In this case, the two scenarios in specification have the same relative execution path in program.

```
1   public void sort(int[] array){
2       if (array.length <= 100){
3           bubbleSort(array);
4       } else if (array.length > 100 &&
5                           array.length <= 500){
6           insertSort(array);
7       } else if (array.length > 500 &&
8                           array.length <= 1000){
9           quickSort(array);
10      }
11  }
12
13  public void bubbleSort(int[] array){
14
15  public void insertSort(int[] array){
16
17  public void quickSort(int[] array){
```

Fig. 5: Example program of "One Scenario to Multi Paths"

```
1     public void sort(int[] array){
2         int index = 1;
3
4         while(index < array.length){
5             for(int i = 0; i < index; i++){
6                 if(array[i] > array[index]){
7                     int temp = array[index];
8                     array[index] = array[i];
9                     array[i] = temp;
10                }
11            }
12            index++;
13        }
14    }
```

Fig. 6: Example program of "Multi Scenarios to One Path"

E. 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. The executable path, statements {1, 2, 4, 7, 8}, in the program shown in Figure 4 is a path without relative functional scenario. It will be executed if the length of input array is larger than 1,000. Usually we think these kind of paths *relative to process*.

IV. Experiment

To investigate the effectiveness of Liu's automatic specification-based testing approach, we take an IC card system as a target to perform testing. We count the number of execution paths in the implementation program and calculate the coverage of paths tested by using the testing approach. The results indicate that this testing method may not be effective in some circumstances. We analyse the results, discuss the specific causes for the ineffectiveness, and propose some suggestions for improvement. The specification and implementation of this IC card system are completed by different researchers. The test cases are generated by another researcher based on the generation criteria and algorithms.

A. Experiment Background

The target system of our experiment 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 by losing the card. Customers can swipe the card to take transportation or use the card to buy train tickets. If the amount in card is not enough, the customers can reload by cash or from associated bank account. The customers can also transfer the money back to the bank account, but they can not get cash from the IC card directly. For the customers who need commute, they can set monthly payment for one route to get discount.

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. All the 6 processes are described briefly in the following list. 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 is developed by using Java under the Eclipse environment. The implementation program contains 14 classes, 2200 lines code, and 112 execution paths.

- **RailwayTravel**: This process handles the situation when card holder swipes card to enter or leave station. When customers enter the station, system checks whether the card is in the IC card database and if the balance in IC card is more than the pre-defined minimum amount. When customers try to leave station, system first checks whether the route just taken is marked as monthly payment. If it is, then open the gate to let the customer leave, otherwise, discount the fare from the balance of the card.
- **PurchaseTicket**: This process defines the function that IC card is used to buy tickets. The fare of the selected route will be calculated first and then be discount from the balance of the card. If the fare is more than the balance, the purchase will fail.
- **ReloadByCash**: The function reloading money to IC cared is defined is this process. Note that the reloading

TABLE III: Information of Scenarios and Corresponding Executable Paths

No.	ProcessName	Scenarios	Input	Decomposed	Paths
1	RailwayTravel	5	2	3	12
2	PurchesTicket	2	3	3	3
3	ReloadByCash	3	3	3	5
4	MonthlyPayment	2	5	5	78
5	ReloadFromAccount	3	4	3	7
6	TransferToAccount	2	4	3	7

will fail if the sum of reloading amount and the balance of the card is more than the pre-defined maximum amount.

- **SetMonthlyPayment**: The function defined in this process is used to set monthly payment. The card holders select a route, a beginning date and number of months they want to set. If the beginning date is the end of month, the expire date should be the end of month. Otherwise the two dates should be the same day in a month.
- **ReloadFromAccount**: The function defined in this process is similar to the function defined in process "***ReloadByCash***". The difference is the reloading money is from the associated account.
- **TransferToAccount**: Reverse of function defined in process "***ReloadFromAccount***". The transfer amount should be less than the balance of the IC card.

If the functional scenarios used to generate test cases are from a lower level module, the testing can be realized as a unit testing. This is because each of the processes in lower level module corresponds to a specific part of the program. If the functional scenarios used to generate test cases are from a higher level module, the testing can be realized as a integration testing.

In our experiment, only the functional scenarios extracted from the 6 processes in the top level are used to generate test cases. The information of the scenarios and their relation with corresponding execution paths are listed in Table III. Item "***Process Name***" indicates the name of the processes in IC card system specification; "***Scenarios***" expresses how many functional scenarios are included in the process; "***Input***" shows the number of input variables of the process; "***Decomposed***" denotes the number of processes which are decomposed from the process and "***Paths***" indicate how many execution paths in program correspond to the process.

B. Results

Total 192 test cases are generated from the 17 functional scenarios in the test, and the statistics of the result are listed in Table IV. Item "***Scenario No.***" identifies the scenario in a process; "***Relative Paths***" shows how many execution paths in program have relation with the scenario; "***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. For instance, the first line in Table IV indicates that the first scenario in process "***RailwayTravel***" has one relative path in implementation program, 5 test cases derived from it make the only one path are tested, the path coverage is 100% in this case. Note that the total of "***Relative Paths***" column is 113 because there is one path shared by the second and third scenario of process "ReloadFromAccount".

V. Analysis and Proposal

The result of the experiment indicates that the FSBT is effective in most circumstances, but it is unlikely to be effective in some specific situations. The data in Table IV show that all of the functional scenarios in the specification are implemented in program, so the relation "one scenario to no path" described in Section III does not exist in our experiment. All of the paths that are in relation with "one scenario to one path" or "multi scenarios to one path" are tested as we analysed in Section III. The paths which are not tested in experiment are the paths in relation to "one scenario to multi paths" or "paths to no scenarios", and it confirms our analysis that the existence of these two relations usually reduces the testing coverage of FSBT.

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 **Criterion 1** to extend the contents of the set of test cases derived from one defined function.

Criterion 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 DNF in which the conjunction clauses may be the *testing condition* of other functional scenario in the same operation. So that the test cases derived from this predicate may satisfiey other scenarios. For example, considering the first scenario of the process specification shown in Figure 2. The pre-condition in this scenario is $\forall i \in inds(array) \cdot is\_int(array[i]) \wedge array.length \leq 1000$ and the *guard condition* is $array.length \leq 100$. By decomposing the predicate $\neg(\forall i \in inds(array) \cdot is\_int(array[i]) \wedge array.length \leq 1000 \wedge array.length \leq 100)$, we can get one of the conjunction clause in its DNF, $\forall i \in inds(array) \cdot is\_int(array[i]) \wedge array.length \leq 1000 \wedge array.length > 100$, and it is exactly the *testing condition* of the second scenario. 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

TABLE IV: Test Result

No.	Process Name	Scenario No.	Relative Paths	Test Cases	Tested Paths	Coverage(%)
1	RailwayTravel	1	1	5	1	100
2	RailwayTravel	2	6	15	4	67
3	RailwayTravel	3, 4, 5	1	15	1	100
4	RailwayTravel	null	4	0	0	0
5	PurchaseTickets	1	1	8	1	100
6	PurchaseTickets	2	1	5	1	100
7	PurchaseTickets	null	1	0	0	0
8	ReloadByCash	1	1	7	1	100
9	ReloadByCash	2	1	5	1	100
10	ReloadByCash	3	1	5	1	100
11	ReloadByCash	null	2	0	0	0
12	MonthlyPayment	1	72	60	26	36
13	MonthlyPayment	2	3	17	2	67
14	MonthlyPayment	null	3	0	0	0
15	ReloadFromAccount	1	1	8	1	100
16	ReloadFromAccount	2	1	6	1	100
17	ReloadFromAccount	3	2	20	2	100
18	ReloadFromAccount	null	4	0	0	0
19	TransferToAccount	1	2	8	1	50
20	TransferToAccount	2	1	8	1	100
21	TransferToAccount	null	4	0	0	0

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 **Criterion 2**.

Criterion 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 FSBT, 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, **Criterion 3** can be applied in the test case generation process.

Criterion 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 experiment again with the three criteria we proposed. The results of two experiments are shown in Figure 7. The y-axis in Figure 7 is coverage and the x-axis is the circumstances in which the coverage of paths in not 100 percent by using original FSBT. Note that we group the paths in relation “paths to no scenario”. Obviously, the effectiveness is improved by using our proposal.

VI. 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.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

Fig. 7: Result of Improved FSBT

applications based on LTL specification in [7]. Khrushid et al. [8] present a framework named TestEra for testing Java program automatically based on specification. It employs Alloy analyzer for instance enumeration to generate all non-isomorphic test data [9]. Bandyopadhyay et al. [10] improve the existing test input generation method based on sequence diagrams of UML specification by consider 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. [11] 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 change the length of test case. In [12], 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 a effective way for specification-based test case generation.

VII. Conclusions and Future Work

In this paper, we performed an experiment to assess FSBT method. Based on the test results we find that this method is effective when the specification is complete, but it may be ineffective if the specification is incomplete. In the cases that the specification is incomplete, we propose some criteria for test case generation to ensure more execution paths can be tested. The final results show that our criteria can improve the effectiveness of the testing method. In the future, we intend to use a large-scale system to assess the FSBT method and the proposed criteria farther, and build a software tool to implement the testing method with our criteria.

Acknowledgement

This work is supported by NII Collaborative Research Program. Shaoying Liu is also supported by the NSFC Grant (No. 60910004), and 973 Program of China Grant (No. 2010CB328102).

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] S. Khurshid, and D. Marinov. TestEra: Specification-based Testing of Java Program using SAT. Automated Software Engineering, 11(4), pages 403-434, 2004.
[9] S. Khurshid, D. Marinov, I. Shlyakhter, and D. Jackson. A Case for Efficient Solution Enumeration. Proceedings of SAT2003, pages 297-298. LNCS 2919, Springer-Verlag, 2003.
[10] 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.
[11] 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.
[12] 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.