DOI: 10.2298/CSIS120301050L

A Formal Approach to Testing Programs in Practice

Shaoying Liu[1], Wuwei Shen[2], and Shin Nakajima[3]

[1] Department of Computer Science, Hosei University, Japan
sliu@hosei.ac.jp
[2] Department of Computer Science, Western Michigan University, USA
wshen@wmich..edu
[3] NII, Japan
nkjm@nii.ac.jp

Abstract. A program required to be tested in practice often has no available source code for some reason and how to adequately test such a program is still an open problem. In this paper, we describe a formal specification-based testing approach to tackle this challenge. The principal idea is first to formalize the informal requirements into formal operation specifications that take the interface scenarios of the program into account, and then utilize the specifications for test case generation and test result analysis. An example and case study of applying the approach to an IC card system is presented to illustrate its usage and analyze its performance.
Keywords: Specification-based testing, Formal specification, Black-box testing

1. Introduction

Programs required to be tested in practice often have no source code for some reasons (e.g., confidentiality) but testing such programs is necessary and important for software quality assurance in industry. A traditional approach to dealing with this problem is black-box testing (or functional testing) [1], but how to carry out an adequate black-box testing in this situation still remains a challenge.

The central issue in tackling this problem is how to make the tester thoroughly understand the desired functionality of the program under testing. One way is to discover it from the requirements that are usually written informally, but due to the ambiguities of the requirements, this may not be easily achieved. Another possibility is to use some test cases to run the program to detect the input-output relations. Each input-output relation can be reflected by a program interface trace, which we call an *interface scenario*. Each interface scenario indicates the implementation of a functional scenario (a sequence of operations), but it does not provide a complete definition of the functional scenario due to the fact that the test cases do not cover all necessary input values for the implementation of the functional scenario.

In this paper, we describe an approach to using formalization to help the tester understand the desired functionality of the program and build a firm foundation for testing the program. Specifically, our approach suggests that the requirements be formalized into a set of formal operation specifications whose signatures are constructed based on the program interface scenarios. The formalization will force the tester to draw an accurate, complete, and precise picture about the desired behaviors of the program and to prepare a solid foundation for testing. To make our approach work effectively, the formal specification must be ensured to be internally consistent and valid with respect to the informal requirements. This task can be fulfilled by means of formal specification inspection [2] and/or specification animation [3]. Since this issue is beyond the scope of this paper, we omit further discussions. The reader can refer to the related work for details of the inspection and animation techniques. In this paper, we focus on the descriptions of how the interface scenarios of a program are derived, how the formal operation specifications are constructed based upon the program interface scenarios, and how testing is carried out based upon the formal specifications.

The remainder of the paper is organized as follows. Section 2 describes the essential ideas of the testing approach and the process of testing. Section 3 discusses how to formalize the informal requirements based upon the program interface scenarios. Section 4 focuses on the test case generation and test result analysis. Section 5 presents an example to illustrate how our approach is applied to test an IC Card software. Section 6 introduces the related work and compares it with our approach. Finally, in Section 7 we conclude the paper and point out future research directions.

2. Essential Ideas

The essential ideas of our approach is first to formalize the requirements and then to carry out a testing. When formalizing the requirements, we abstract the program under testing into a set of operations, each defining an independent service, such as $Withdraw$ or $Deposit$ in an automated telling machine (ATM) software. The goal of the formalization is therefore to write a formal specification for each operation. To this end, we need to determine the signature of each operation, including the input variables, output variables, and the related external variables (or state variables). This signature is expected to be consistent with the interface of the program in order to ensure that test cases generated from the specification can be directly adopted for testing. Since there might be a gap between the informal requirements and the interface of the program, it is usually difficult to achieve the consistency without examining the program. However, since the source code of the program is assumed not to be available, it is impossible to examine the implementation details. The only information about the program we can grasp is its interface scenarios and its dynamic behaviors when it is executed. For this reason, our approach suggests that the formation of each operation's signature takes the *interface scenarios* of the program into

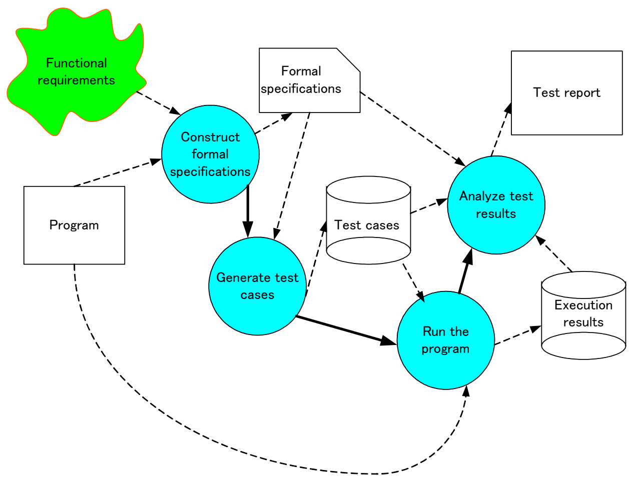


Fig. 1. The testing process using our approach

account. As far as defining the functionality of the operation is concerned, we use pre- and post-conditions, the most commonly applied notation in the literature [4], to specify it, based on the informal requirements, as shown in an example later. One may argue that writing such a formal specification may not be a routine exercise in practice and therefore our proposed approach depending on formal specifications may not be practical enough. While this might be true for the current industry where formal methods are rarely used and software crisis continues, the future software development would be significantly improved when some of the activities, such as testing, are undertaken, with effective tool support, by well trained practitioners or consultants in formal methods.

An interface scenario of the program is represented by a sequence of inputs and the corresponding output of the program. It can be identified by executing the program using sample inputs, but the precise relation between the sequence of inputs and the output is unknown. The goal of testing is to identify whether such a relation is implemented correctly in the program with respect to its definition in the specification. Assume that we have achieved formal specifications for the program under testing, what we need to do next is to generate test cases based upon the specifications to test the program. The whole process of testing using our approach is illustrated in Figure 1.

3. Construction of Formal Specifications

The construction of a formal specification for the program under testing takes the following three steps:

A software system for a simplified Automated Teller Machine (ATM) needs to provide two services for customers. ***The services include withdrawing money from the customer's bank accoun***t and inquiry for the account balance. ***To withdraw money, the customer's card and password as well as the requested amount are required. The bank policy does not allow overdraft when the service for withdrawal is used.*** For an inquiry about the customer's account balance, the system should provide the current balance of the customer's account.

Fig. 2. A description of informal requirements for the simplified ATM.

1. Find important functional scenarios described in the informal requirements, where each functional scenario describes an independent, desired function in terms of the input-output relation.
2. Form an operation to define all of the identified functional scenarios, but the signature of the operation must be formed based on both the requirements and the interface scenarios of the corresponding program.
3. Write the pre- and post-conditions for the operation using a formal specification language.

Next, we discuss these three issues, respectively.

3.1. Finding functional scenarios

The first step is to find the functional scenarios important to the end user from the requirements description. This can be done by reviewing and analyzing the informal requirements specification. We define each identified functional scenario as a single operation.

Let us consider a simplified ATM system as an example. Suppose the informal requirements for the system are given in Figure 2 and we want to find all of the important functional scenarios. We analyze the document to identify the relevant requirements and highlight the related text segments using the bold italic font, as indicated in the figure. Through the analysis, we realize that this functional scenario is concerned with withdrawing money from a bank account. Therefore, we form an operation and name it $Withdraw$. Applying the same principle, all of other functional scenarios can be identified and defined as appropriate operations.

3.2. Identifying interface scenarios

The next step is to determine the signature of the operation. As discussed previously, this signature needs to be kept consistent with that of the program in order to facilitate the direct adoption of test cases to be generated from the specification. To this end, we need to identify the corresponding interface scenarios when the program is executed. For this purpose, we can select test cases from

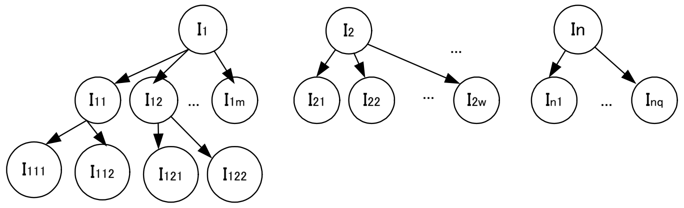


Fig. 3. Illustration of an interface tree

relevant types randomly. During the execution of the program, we continue to supply test cases if they are required by the program (e.g., via its GUI). Such an activity continues until the final execution result is provided. Since this technique is not aimed at really testing the program but at finding potential interface scenarios, the test cases needed should be as merely necessary as possible. This can be interpreted differently in practice when practical constraints are taken into account, but the following specific criterion can be considered as a principle for striking a balance between finding interface scenarios and generating test cases at this stage.

Let the first level (top level) interface of the program under testing, called $Interface_1$, show n exclusive input patterns: $I_1, I_2, ..., I_n$, indicating n different choices for the input of the program. For each selected pattern I_i $(i = 1...n)$, another lower level interface showing m input patterns $I_{11,}\ I_{12,}\ ...,\ I_{1m}$ will be displayed. Applying this principle to all interface patterns at all levels, an interface tree for the whole program could be built, as illustrated in Figure 3. Then, a set of test cases need to be generated to cover every path from one of the top nodes (denoting one possible input pattern on the top level interface) to one of the bottom nodes (denoting one possible input pattern on the related bottom level interface), such as $path_1 = I_1,\ I_{11},\ I_{111}$ and $I_1,\ I_{12},\ I_{121}$. Note that there might be many functional scenarios (behaviors) under each interface scenario (input). Creating test cases to cover all of such functional scenarios would increase the cost while missing identifying any interface scenario would jeopardize the functional coverage in formalizing the specification. In this sense, the criterion for test case generation described above provides a good balance.

Assume that we have obtained all interface scenarios, the next issue is how to represent them so that they will present a clear guideline for the formalization of the corresponding functional requirements. We adopt the following finite state machine (FSM) notation for this purpose.

Definition 1. *A finite-state machine (FSM) is a quintuple $(\Sigma, S, s0, \delta, F)$, where Σ is a non-empty set of input symbols, S is a finite, non-empty set of states, $s0 \in S$ is an initial state, δ is the state-transition function, and $F \subseteq S$ is the set of final states.*

In the FSM representing interface scenarios of a program, the five elements are interpreted as follows. Each state S denotes one page of the GUI of the

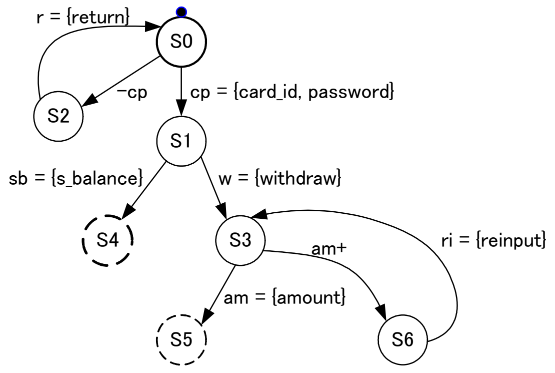


Fig. 4. An example of the finite state machine for a simplified ATM

program, where a page of GUI is an image with certain layout and contents shown on the display of computer. Each input symbol in Σ denotes a set of input items whose concrete values are necessary for executing the program. $s0 \in S$ denotes the very top page of the GUI of the program. δ defines transitions of pages of the GUI based upon the input items. F represents the set of final pages showing the execution result of the program.

Figure 4 shows a simple example of FSM for a simplified Automated Teller Machine (ATM) program. The state $s0$, marked with a black dot at the top, is the initial state, representing the top page of the GUI of the ATM system, and the states $s4$ and $s5$ are both final states which are deliberately drawn in broken line circles. The FSM describes two interface scenarios. One is the scenario for withdrawing money from the bank account, and another is to show the account balance. The scenario for withdrawing money starts from the initial state $s0$. When correct user's $card\_id$ and $password$ are provided, which is denoted by symbol cp, the state will transfer to $s1$; otherwise, if any of them is incorrect, denoted by symbol $-cp$, the state will transfer to state $s2$ and will return to state $s0$. At state $s1$, if the input symbol is w (denoting the $withdraw$ command), the state will transfer to state $s3$; otherwise, the state will transfer to the final state $s4$ denoting the final page of GUI in which the requested account balance is shown. At state $s3$, if the requested amount smaller than the account balance is provided, denoted by am, the state will transfer to the final state $s5$; otherwise, if the amount is greater than the account balance, the state will transfer to state $s6$, and then returns to state $s3$ for re-input of appropriate amount.

In the context of the FSM, an interface scenario is in fact defined as a set of state transition sequences. Each sequence has the form: $[s0, s1, s2, ..., sn]$, where $s0$ must be the initial state of the FSM and sn must be a final state. For example, the scenario for withdrawal discussed above is defined by the set of state transition sequences:

$Scenario_1 = \{[s0, s1, s3, s5], [s0, s1, s3, s6, s3, s5],$

$[s0, s1, s3, s6, s3, s6, s3, s5]$, ...,
$[s0, s1, s3, s6, s3, s6, ..., s3, s5]\}$.

Obviously, this interface scenario is infinite, but the input values associated with the state transitions can be classified into a finite set based on which corresponding input variables can be declared in the signature of the corresponding operation. For example, three input variables for representing inputs *card_id*, *password*, and *withdrawal amount* can be declared for the signature of the $withdraw$ operation. As a result, the operation has the following format:

```
process $Withdraw(card\_id : string,$
                  $password : nat0,$
                  $amount : nat0)$
        $cash : nat \mid error\_message : string$
end_process
```

where the output variables $cash$ and $error\_message$ are created for the need of writing the formal specification.

The operation contains three input variables $card\_id$, $password$, and $amount$. The $card\_id$ is a string, and both $password$ and $amount$ are a natural number (including zero). The output of the operation has two exclusive possibilities: either $cash$ or $error\_message$. The pre- and post-conditions of the operation are empty at the moment, but will be completed in the next step. The operation is written in the SOFL formal specification language [5] for the sake of our expertise. SOFL is an extension of VDM-SL [6] to have more capability for modelling of practical systems. Note that in spite of using SOFL for discussion, our approach presented in this paper is independent of specification languages; it can be applied to any formal notation using pre- and post-conditions.

3.3. Construction of Formal Specifications

Having understood all of the implemented interface scenarios, the next key issue is how to formally define the identified functional scenarios as operation specifications. As mentioned in Section 3.1, the potential behavior of each operation can be derived from the informal requirements, but the understanding of it may remain limited due to the ambiguities of the informal expressions. Writing a formal specification for the operation can force the tester to consider every possible aspect and to establish a firm foundation for testing. The key issue in writing the formal specification is how to provide appropriate pre- and post-conditions for the operation.

This can be done by analyzing the corresponding functional scenario. We first target on the post-condition because it focuses on the function of the operation by defining the relation between input and output. During this process, we may find that some state variables (or external variables as we call in both VDM and SOFL) are necessary for expressing the post-condition. We then try to figure out the pre-condition to ensure that the post-condition defines a valid final state when the pre-condition is satisfied by an initial state. Consider the operation $Withdraw$ mentioned previously as an example. Applying this technique, we provide the following specification:

```
process $Withdraw(card\_id\colon string,$
                $password\colon nat0,$
                $amount\colon nat0)$
        $cash\colon nat \mid error\_message\colon string$
ext wr $accounts\colon set\ of\ Account$
pre $exists![acc\colon accounts] \mid$
        $acc.card\_id = card\_id\ and$
        $acc.password = password$
post $let\ acc\colon accounts \mid$
        $acc.card\_id = card\_id\ and$
        $acc.password = password$
    $in$
     $amount \leq acc.balance\ and$
     $cash = amount\ and$
     $accounts =$
     $union($
          $diff(\tilde{}accounts, \{acc\}),$
           $\{modify(acc, balance \rightarrow$
                    $acc.balance - amount)\}$
         $)$
    $or$
    $not\ (amount \leq acc.balance)\ and$
    $\mathbf{bound}(error\_message)\ and$
    $accounts = \tilde{}accounts$
end_process
```

where the logical operator “and” is used for $\wedge$ and “or” for $\vee$.

In this specification, an external variable $accounts$ is declared. This variable is used to hold a set of customers’ bank accounts, each being a member of the type $Account$ that is declared as a composite type with three fields: $card\_id$, $password$, and $balance$. The pre-condition requires the existence of a unique account in $accounts$ such that its $card\_id$ and $password$ are the same as the input $card\_id$ and $password$, respectively. The post-condition defines two cases: one is for a successful withdrawal and the other is a failed withdrawal. When the requested amount is not greater than the balance, the withdrawal will be successful and the requested amount will be delivered, which is denoted by the variable $cash$. The balance will also be updated in the account to reflect the reduction of the requested amount. On the other hand, when the withdrawal is failed (i.e., the amount is greater than the account balance), an error message denoted by the variable $error\_message$ is provided. For testing purpose, we are not interested in the specific content of the error message implemented by the program, therefore, we use the expression $\mathbf{bound}(error\_message)$ in the post-condition to mean that $error\_message$ is made available but its content is yet to be determined by the programmer later.

4. Test Case Generation and Result Analysis

The goal of testing is to check whether each specification scenario defined in the specification is correctly implemented by the program. Theoretically, for any *specification scenario*, there should be an *implementation scenario* in the program to correctly implement it if the program refines the specification. A natural strategy for test case generation is therefore to let the generated test cases cover all the specification scenarios. To effectively apply this strategy, we have worked out a decompositional approach to test set generation from a pre-post style formal specification, which will be introduced later in this section. To this end, we first need to precisely define the fundamental concepts known as *functional scenario form* (*FSF*) and *functional scenario* (*FS*).

4.1. Functional scenarios

For simplicity, let $S(S_{iv}, S_{ov})[S_{pre}, S_{post}]$ denote the 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-conditions of S, respectively. In S_{post}, we use $\tilde{}x$ and x to represent the initial value before the operation and the final value after the operation of external (or state) variable x, respectively. Thus, $\tilde{}x$ serves as an actual input variable of the operation while x is treated as an output variable; that is, $\tilde{}x \in S_{iv}$ and $x \in S_{ov}$. Of course, S_{iv} and S_{ov} may contain other input parameter variables and output parameter variables, respectively.

Definition 2. *Let* $S_{post} \equiv (C_1 \wedge D_1) \vee (C_2 \wedge D_2) \vee \cdots \vee (C_n \wedge D_n)$*, 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 (*functional*)* scenario f_s *of* S *is a conjunction* $S_{pre}\tilde{} \wedge C_i \wedge D_i$*, and the expression* $(S_{pre}\tilde{} \wedge C_1 \wedge D_1) \vee (S_{pre}\tilde{} \wedge C_2 \wedge D_2) \vee \cdots \vee (S_{pre}\tilde{} \wedge C_n \wedge D_n)$ *is called a* functional scenario form *(*FSF*) of* S*.*

The decorated pre-condition $S_{pre}\tilde{} = S_{pre}[\tilde{}\sigma/\sigma]$ denotes the predicate resulting from substituting the initial state $\tilde{}\sigma$ for the final state σ in pre-condition S_{pre}. We treat a conjunction $S_{pre}\tilde{} \wedge C_i \wedge D_i$ as a scenario because it defines an independent behavior: when $S_{pre}\tilde{} \wedge C_i$ is satisfied by the initial state (or intuitively by the input variables), the final state (or the output variables) is defined by the defining condition D_i. To facilitate our discussion throughout this paper, we call the conjunction $S_{pre}\tilde{} \wedge C_i$ the *test condition* of the scenario $S_{pre}\tilde{} \wedge C_i \wedge D_i$.

Note that simply treating a disjunctive clause in the disjunctive normal form of the post-condition as a functional scenario is not necessarily correct in supporting our method for checking the refinement relation between the specification and the program. For example, let $x > 0 \wedge (y = x \vee y = -x) \vee x \leq 0 \wedge y = x + 1$ be the post-condition of an operation whose pre-condition is assumed to be $true$, where x is the input and y the output. It states that when $x > 0$, y is defined either as x or as $-x$ (the specifier does not care which definition will be implemented). In this case, the programmer may refine it into the post-condition $x > 0 \wedge y = x \vee x \leq 0 \wedge y = x+1$ and then implement it, which will of course satisfy the original specification. However, if we convert the original specification into the disjunctive normal form $x > 0 \wedge y = x \vee x > 0 \wedge y = -x \vee x \leq 0 \wedge y = x+1$, and treat each of the two disjunctive clauses $x > 0 \wedge y = x$ and $x > 0 \wedge y = -x$ as an individual functional scenario, respectively, and generate test cases to cover each of these two scenarios, we may not find a satisfactory answer in the program for the scenario $x > 0 \wedge y = -x$ because no program paths implementing this scenario exist in the program. This may lead to a confusion in testing and consequently increase the cost. Another concern is that a disjunctive clause in a DNF may not properly reflect the client's perception of a desired function. For instance, neither of the two expressions $x > 0 \wedge y = x$ and $x > 0 \wedge y = -x$ is appropriate to reflect the client's desire individually, but the functional scenario $x > 0 \wedge (y = x \vee y = -x)$ obtained by their combination does. For the above reasons, our strategy of generating test cases based on functional scenarios is adopted in this paper.

Any operation specification can be transformed to an equivalent FSF and a proved algorithm for such a transformation is made available in our previous publication [7]. Below we use the operation $Swap\_Increase\_Maintain$ as an example to explain the concepts of FSF and functional scenario. The example may look a little artificial but it allows us to illustrate all the relevant aspects of the concepts.

$Swap\_Increase\_Maintain$
$(\{\tilde{}x, \tilde{}y\}, \{x, y\})$
$[x >= 0,$
$\tilde{}x < \tilde{}y \wedge y = \tilde{}x \wedge x = \tilde{}y \wedge \tilde{}x < 100 \vee$
$\tilde{}x < \tilde{}y \wedge (y > \tilde{}x \wedge x > \tilde{}y) \wedge \tilde{}x >= 100 \vee$
$\tilde{}x >= \tilde{}y \wedge (y = \tilde{}y \wedge x = \tilde{}x) \vee$
$\tilde{}x >= \tilde{}y \wedge (y + \tilde{}x = x + \tilde{}y)]$,

where $Swap\_Increase\_Maintain_{iv} = \{\tilde{}x, \tilde{}y\}$; $Swap\_Increase\_Maintain_{ov} = \{x, y\}$; $x >= 0$ is the pre-condition; the last predicate expression is the post-condition; and all the variables involved are of the type *real*. The operation exchanges, increases or "sustains" the values of x and y, depending on certain conditions satisfied by the input. An FSF of the specification is derived as follows:

(1) $\tilde{}x >= 0 \wedge \tilde{}x < \tilde{}y \wedge \tilde{}x < 100 \wedge y = \tilde{}x \wedge x = \tilde{}y \vee$
(2) $\tilde{}x >= 0 \wedge \tilde{}x < \tilde{}y \wedge \tilde{}x >= 100 \wedge (y > \tilde{}x \wedge x > \tilde{}y) \vee$
(3) $\tilde{}x >= 0 \wedge \tilde{}x >= \tilde{}y \wedge (y = \tilde{}y \wedge x = \tilde{}x \vee y + \tilde{}x = x + \tilde{}y)$

This FSF is derived from the DNF of the post-condition and includes the three functional scenarios marked by (1), (2), and (3) (after removing the last operator $\vee$ in each case), respectively. In (1), $\tilde{}x >= 0$ is the decorated pre-condition; $\tilde{}x <= \tilde{}y \wedge \tilde{}x < 100$ is the guard condition; and $y = \tilde{}x \wedge x = \tilde{}y$ is the defining condition. The other two scenarios can be interpreted similarly.

Before describing the test strategy, we need to clarify the concepts of *test case* and *test set* precisely.

Definition 3. *Let $S_{iv} = \{x_1, x_2, ..., x_r\}$ be the set of all input variables of operation S and $Type(x_i)$ denotes the type of x_i $(i = 1, ..., r)$. Then, a test case for S, denoted by T_c, is a mapping from S_{iv} to the set $Values$:*
$T_c : S_{iv} \rightarrow Values$
where $Values = Type(x_1) \cup Type(x_2) \cup \cdots \cup Type(x_r)$.

A test case is usually expressed as a set of pairs of input variables and their values, for example, $\{(x_1, 5), (x_2, 10), ..., (x_r, 20)\}$. A test set for operation S is a set of test cases, and is usually expressed as a set of sets of pairs. With the above preparation, we can now define precisely the test strategy.

Test Strategy: Let operation S have an FSF $(S_{pre}\tilde{} \wedge C_1 \wedge D_1) \vee (S_{pre}\tilde{} \wedge C_2 \wedge D_2) \vee \cdots \vee (S_{pre}\tilde{} \wedge C_n \wedge D_n)$ where $(n \geq 1)$. Let T be a test set for S. Then, T must satisfy the condition $(\forall_{i \in \{1,...,n\}} \exists_{t \in T} \cdot S_{pre}\tilde{}(t) \wedge C_i(t))$.

We call this strategy *scenario-coverage strategy* (SCS). The strategy states that for each functional scenario there exist some test cases in test set T that satisfy its test condition. If T meets this condition, it will ensure that every functional scenario defined in the specification is tested (at least once), thus ensuring that no specified functional behavior is missed in the test.

To ensure that the test strategy is fully applied, a decompositional test set generation approach is proposed, which is described in detail next.

4.2. Decompositional Approach to Test Set Generation

Using the decompositional test set generation approach, the test set generated from an operation specification is realized by generating test sets from all of its functional scenarios. The production of test set from a functional scenario is realized by generating them from its test condition, which can be divided further into test set generations from every disjunctive clause of the test condition. To clearly describe the proposed method, we first need to define a function, called *test case generator* (TCG), that yields a test set for a given logical expression such that every test case in the set plays an independent role while it satisfies the expression (e.g., assuming the expression is a disjunction, each test case in the set makes each disjunctive clause true). We use $\mathcal{G}$ to denote the TCG, which is a function from the universal set of logical expressions L_E to the universal set of test sets T_S, formally,
$\mathcal{G} : L_E \rightarrow T_S$.

By logical expressions here we mean the first order logic expressions written in SOFL. Let p be a logical expression in L_E. Then, $\mathcal{G}(p)$ represents the test set generated, which is a member of T_S and satisfies p (i.e., every test case in the test set satisfies p). The key issue now is how $\mathcal{G}$ is defined. We will define $\mathcal{G}$ to ensure that our test strategy discussed previously is supported.

An FSF of operation specification S is a disjunction of all its functional scenarios, defining a set of independent behaviors, the test set generated from the

FSF should therefore be the union of all the test sets generated from all of its functional scenarios, as formalized in **Criterion 1** below.

Criterion 1: $\mathcal{G}((S_{pre}\tilde{} \wedge C_1 \wedge D_1) \vee (S_{pre}\tilde{} \wedge C_2 \wedge D_2) \vee \cdots \vee (S_{pre}\tilde{} \wedge C_n \wedge D_n)) = \mathcal{G}(S_{pre}\tilde{} \wedge C_1 \wedge D_1) \cup \mathcal{G}(S_{pre}\tilde{} \wedge C_2 \wedge D_2) \cup \cdots \cup \mathcal{G}(S_{pre}\tilde{} \wedge C_n \wedge D_n)$.

Since the execution of a program only requires input values, test case generation from a functional scenario actually depends only on its test condition. This idea is reflected in **Criterion 2**.

Criterion 2: Let $S_{pre}\tilde{} \wedge C_i \wedge D_i$ $(i = 1, ..., n)$ be a functional scenario of operation specification S. Then, $\mathcal{G}(S_{pre}\tilde{} \wedge C_i \wedge D_i) = \mathcal{G}(S_{pre}\tilde{} \wedge C_i)$.

In general, pre-condition $S_{pre}\tilde{}$ can be in any form of predicate expression. To define $\mathcal{G}$ to deal with test case generation from the conjunction $S_{pre}\tilde{} \wedge C_i$, a systematic way is first to translate the conjunction into an equivalent disjunctive normal form (DNF) and then define $\mathcal{G}$ based on it, as reflected in **Criterion 3**.

Criterion 3: Let $P_1 \vee P_2 \vee \cdots \vee P_m$ be a DNF of the test condition $S_{pre}\tilde{} \wedge C_i$. Then, we define $\mathcal{G}(S_{pre}\tilde{} \wedge C_i) = \mathcal{G}(P_1 \vee P_2 \vee \cdots \vee P_m) = \mathcal{G}(P_1) \cup \mathcal{G}(P_2) \cup \cdots \cup \mathcal{G}(P_m)$.

A P_i $(i = 1, ..., m)$ is a conjunction of atomic predicate expressions, say $Q_i^1 \wedge Q_i^2 \wedge \cdots \wedge Q_i^w$. An atomic predicate expression, in our context, is one of the three kinds of predicate expressions: (1) a relation (e.g., $x > y$), (2) a negation of a relation, and (3) a *strict quantified predicate*. A strict quantified predicate is a quantified predicate whose body does not contain any atomic predicate unrelated to its bound variables. For example, $\forall_{x \in X} \cdot y > x$ is a strict quantified predicate, while $\forall_{x \in X} \cdot y > x \wedge q > 0$ is not, because it contains the atomic predicate $q > 0$ that is not related to the bound variable x. This criterion indicates that generating test cases from a test condition can be achieved based on its DNF. The focus now is on the issue of how to produce test cases from a clause, such as $P_i \equiv Q_i^1 \wedge Q_i^2 \wedge \cdots \wedge Q_i^w$. To discuss the generation criterion, we first need to define $\mathcal{G}$ to deal with atomic predicate expressions, which will set up a basis for discussions on test case generation from the conjunction.

Criterion 4: Let $S_{iv} = \{x_1, x_2, ..., x_r\}$ and $Q(x_1, x_2, ..., x_q)$ $(q \leq r)$ be a relation involving variables $x_1, x_2, ..., x_q$. Then, $\mathcal{G}(Q(x_1, x_2, ..., x_q)) = \{T_c \mid (\forall_{x \in \{x_1, x_2, ..., x_q\}} \cdot Q(T_c(x_1), T_c(x_2), ..., T_c(x_q))) \wedge (\forall_{x \in (S_{iv} \setminus \{x_1, x_2, ..., x_q\})} \cdot T_c(x) = anyvalue)\}$, $anyvalue$ represents any value taken from the type of variable x.

For a relation involving a subset of the input variables of operation specification S, we generate a test case in which each involved variable is given a specific value in its type and all the generated values satisfy the predicate expression $Q(x_1, x_2, ..., x_q)$, but the input variables of S that are not involved in $Q(x_1, x_2, ..., x_q)$ are all assigned $anyvalue$ from its type. For example, suppose operation specification S has two input variables x and y denoting two integers, then it is possible that $\mathcal{G}(x > 0) = \{\{(x, 2), (y, 8)\}\}$ (containing a single test case) where 8 in the pair $(y, 8)$ is $anyvalue$, but the values of variables x and y in the test set $\mathcal{G}(x < y) = \{\{(x, 2), (y, 5)\}\}$ are not $anyvalue$ but the values that satisfy the relation $x < y$.

Having the above preparation, we can now define the criterion for generating a test set for a conjunction of atomic predicate.

Criterion 5: Let $Q_i^1 \wedge Q_i^2 \wedge \cdots \wedge Q_i^w$ be a conjunction of w atomic predicates in a test condition of a functional scenario of S. Then, we have $\mathcal{G}(Q_i^1 \wedge Q_i^2 \wedge \cdots \wedge Q_i^w) = \mathcal{G}(Q_i^1) \cap \mathcal{G}(Q_i^2) \cap \cdots \cap \mathcal{G}(Q_i^w)$.

Let us take the operation $Withdraw$ as an example to show how the criteria can be applied. The operation is converted into a disjunction of the following two functional scenarios:

(1) $(exists![acc\colon accounts] \mid$
$\quad acc.card\_id = card\_id\ and$
$\quad acc.password = password)\ and$
$acc\ inset\ accounts\ and$
$acc.card\_id = card\_id\ and$
$acc.password = password\ and$
$amount <= acc.balance)\ and$
$(cash = amount\ and$
$accounts = union(diff(\tilde{}accounts, \{acc\}),$
$\{modify(acc, balance- > acc.balance - amount)\})$

(2) $(exists![acc\colon accounts] \mid$
$\quad acc.card\_id = card\_id\ and$
$\quad acc.password = password)\ and$
$acc\ inset\ accounts\ and$
$acc.card\_id = card\_id\ and$
$acc.password = password\ and$
$not\ (amount <= acc.balance))\ and$
$(\mathbf{bound}(error\_message)\ and$
$accounts = \tilde{}accounts)$

where the let expression is translated into the equivalent conjunction:

$acc\ inset\ accounts\ and\ acc.card\_id = card\_id\ and$
$acc.password = password.$

Following the test strategy SCS described above, we generate a test set as shown in Figure 5 that cover the two functional scenarios. Each test case consists of four values for the four input variables (i.e., the three parameters $card\_id$, $password$, $amount$, and one external variable $accounts$). The test case on the first row allows us to test the situation where both the inputs $card\_id$ and $password$ are correct and the input $amount$ is smaller than the customer's account balance. The test case on the second row checks the situation where both the $card\_id$ and $password$ are correct, but the $amount$ exceeds the balance. The one on the third row tests the situation where the pre-condition of the operation is not satisfied because of the inconsistency between the input $password$ and the registered $password$ in the customer's account. Since the pre-condition is treated as an assumption for the operation, the last test case will not necessarily help find errors, but it will allow the tester to understand how the implementation deals with this situation and to provide a feedback to the programmer for potential improvement.

card_id	*password*	*amount*	*accounts*
L4513	3410	50000	{(L4513, 3410, 1000000), (A5910, 3102, 150000), (X2034, 1291, 3400000)}
A5910	3102	250000	{(G8113, 3410, 5060000), (A5910, 3102, 150000), (T2014, 1291, 2900000)}
X2034	1091	10000	{(G8113, 3410, 300000), (L0310, 3102, 35000), (X2034,1291,3400000)}

Fig. 5. A test set for the operation *Withdraw*.

4.3. Test Result Analysis

One of the major advantages of formal specification-based testing is that a precise test oracle can be automatically derived from the specification and can be used to determine, both automatically and manually, whether a test has found an error or not.

Definition 4. *Let $f \equiv S_{pre}\tilde{} \wedge C \wedge D$ be a functional scenario of operation specification S and P be a program implementing S. Let t be a test case generated based on f and r be the result of executing program P using t. Then, an error in P is found using t if the following condition holds:*
$S_{pre}\tilde{}(t) \wedge C(t) \wedge \neg D(t, r)$

This test oracle states that for a test case t satisfying both the pre-condition of S and the guard condition C of functional scenario f, if the execution result r of P does not satisfy the defining condition D (together with t), we can assert that an error in P is found by test case t.

In fact, this test oracle is derived from the refinement relation between S and P, which is formally defined as follows:
$\forall \tilde{}\sigma \in \Sigma \cdot S_{pre}(\tilde{}\sigma) \Longrightarrow S_{post}(\tilde{}\sigma, \sigma)$,
where $\tilde{}\sigma$ denotes the initial state before operation S and σ $(= P(\tilde{}\sigma))$ denotes the final state after operation S that is produced by executing the corresponding program P. P is a correct refinement of specification S if and only if the final state σ produced by executing program P on the initial state $\tilde{}\sigma$ satisfying the pre-condition meets the post-condition. In other words, if there exists any final state that does not satisfies the post-condition after the execution of the program with a satisfactory initial state, the program will not be a correct refinement of the specification, implying that a bug exists in the program.

The test for the operation $Withdraw$ shown in Figure 5 can help to explain how the test oracle can be used. Assume that we obtain the actual execution result R for the test case on the first row in Figure 5 denoted by T_c^1, which consists of two values: updated $accounts$ and $cash$. We substitute them for the variables $accounts$ and $cash$ in the post-condition, respectively, and evaluate the following implication:

$Withdraw_{pre}(T_c^1) \Rightarrow Withdraw_{post}(T_c^1, R)$

If the result is true, it means that no error is found by T_c^1; otherwise, it indicates the presence of some bugs in the program. Similarly, we can also perform the test result analysis for the other two test cases, which we omit here for brevity.

5. An Example

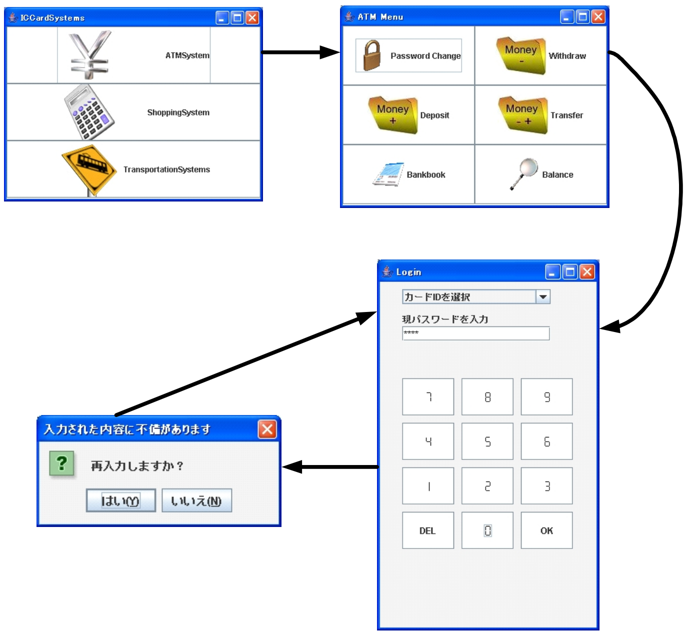
We have applied our approach to test an *IC Card software system* developed by a group of senior students at Hosei University. The system is intended to provide three services: *operations on ATM*, *purchase of railway tickets*, and *payment for shopping*. The operations on ATM includes *withdraw*, *deposit*, *show balance*, *transfer money*, *change password*, and *transactions record printing*. For purchasing railway tickets, the system provides the operations: *purchase a ticket*, *charge the card with cash*, *charge the card from the customer's bank account*, *show balance of the card* (i.e., the amount of money available in the buffer of the card), and *use the card* (for entering and existing a station). The payment for shopping is a service that allows one to pay for the goods purchased at shops using the card. It includes the following specific functions: *check card validity*, *check amount restrictions* (e.g., a card cannot be used for shopping of more than a fixed amount in Japanese yen per day at the same shop), *authenticate the card user*, and *pay for shopping*.

Since the whole example is too large to be put in a paper gracefully, we choose only the ATM sub-system as an example to show how our testing approach can be applied. As the ATM example in Section 3.3 is a simplified version of the system we use in this section, the contents of this section would be understood easily.

5.1. Application

Figure 6 shows a snapshot of the GUI scenario for an unsuccessful withdrawal from a customer's bank account. The first page of the GUI for the system shows three choices: *ATMSystem*, *ShoppingSystem*, and *TransportationSystems*. Clicking on the button named *ATMSystem*, the system transfers from the first page to the ATM GUI page given at the upper-right of the figure. Clicking on the button named *Withdraw*, the GUI changes to the page requesting a *card_id* and a *password*, which is given at the bottom-right of the figure. When a wrong password is input, the system moves to the page displaying an error message and requesting the user to re-enter the password. Clicking on the "yes" button (the left one), the system returns to the page requesting for a *card_id* and a *password*.

Using random testing based on a programmer's brief description of the interface scenarios of the system, we built a set of FSMs for the ATM system



入力された内容に不備があります

再入力しますか？

Fig. 6. A snapshot of using ATM for an unsuccessful withdrawal from the bank account

for simplicity and readability, as shown in Figure 7, each describing a single independent interface scenario, as indicated in the figure. Taking the same approach as we explained in Section 3, we construct an operation specification in pre- and post-conditions for each functional scenario. For the sake of both space and the similarity to the simplified ATM example, we omit the specifications here. We carried out testing of the program based upon the specifications to verify whether the desired behaviors are implemented correctly. The details of the testing is illustrated in Table 5.1.

Since this case study was mainly intended to test the usability of our approach (i.e., how interface scenarios can be found, how formal specifications can be constructed, and how test cases are generated) rather than conduct a rigorous

Features	**No. of test cases**	**No. of detected defects**
withdraw	8	3
deposit	7	5
show balance	6	2
transfer money	8	7
change password	9	6
transactions record printing	7	4

Table 1. Approximate data of the testing

experiment on its effectiveness, we only collected the data of defects found by our test cases rather than measured its effectiveness in terms of defect detection rate or some coverage criterion. There were total 27 defects found as the result, but the main defects that lead to major functional errors with respect to the functional requirements in the specifications are three, which are illustrated as follows:

- In the specification the cash card and password are required to provide before any service (e.g., balance inquiry, withdraw) starts, but the implementation in the program puts the selection of services before the input of card id and password. The purpose of such a requirement is to prevent people without cash card from trying to “play” with the ATM system.
- In the *Withdraw* service, if the requested amount is over the account balance, the customer should be given two choices: giving up the use of the service or re-entering the amount for withdrawal, but the implementation in the program insists only on re-entering the amount. The same error is also found in the *Transfer* service: when the amount for transferring is over the account balance, the program insists only on re-entering the amount rather than offering the choices.
- In the *Deposit* service, the amount for depositing is required not to be over 1,000,000 Japanese yen, but the program does not implement this constraint, therefore allowing any amount to be deposited in one service.

In the three errors, we found that the first one was relatively easier to find, but the second and the third ones took us more efforts to find out, because the missing functions did not cause fundamental problem in providing the services, and therefore did not easily attract the tester’s attention. The tester’s attention was actually raised by examining the corresponding defining condition in the post-condition of the operations. This result shows that formal specification does benefit testing in our approach.

5.2. Discussion

Our experience in the application has shown the usability and effectiveness of our testing approach for programs with no source code available. In particular, formalization of the informal requirements based upon the interface scenarios derived from the program has been found effective in helping the tester detect obscure errors and defects. Since all the potential interface scenarios need to be identified based upon executions of the program for writing interface-consistent formal specifications, a general technique called *testing-and-correction* must be taken. The technique suggests the following activities:

No.1 Run the program using sample test cases (selected based upon the brief description of the functions of the program provided by the programmer) and record the interface scenarios experienced if the program terminates normally.

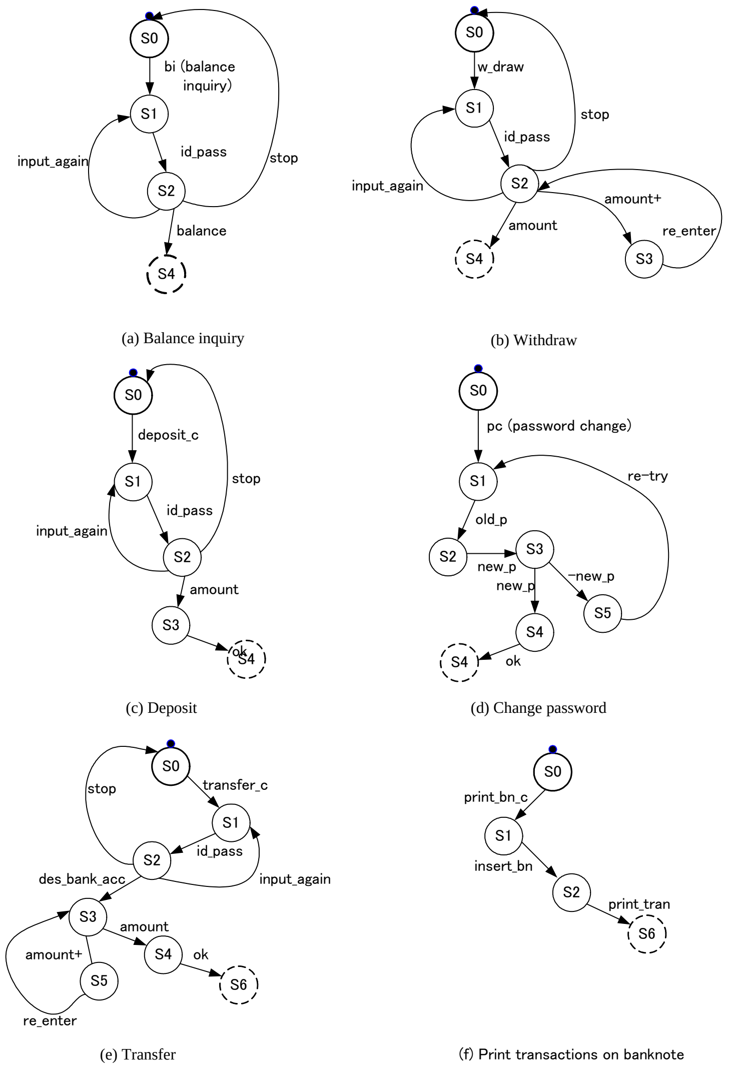


Fig. 7. FSMs for ATM interface scenarios.

No.2 If the program does not terminate normally due to some bugs, the programmer will be asked to eliminate the bugs before the next action in testing is taken.
No.3 Repeat steps 1 and 2 until all interface scenarios are found.

The process of applying the *testing-and-correction* technique helps to detect some obvious errors in the program, but it may not be effective in uncovering obscure errors, especially those violating the requirements in an ambiguous manner. Such obscure errors can be effectively revealed during the formalization of requirements and the testing process based upon the formal specifications.

To effectively use our testing approach in practice, however, the following challenges may need to be addressed.

- The program must always terminate normally or the programmer must always be available for removing bugs detected.
- The tester has sufficient ability and skill in formalizing the informal requirements into formal specifications. For this purpose, the requirements must be complete or the client must be available for consultation whenever the details of the requirements are inquired by the tester.

The first challenge is not exceptional for the application of other testing approaches, but the second one may be a major challenge in practice currently. The reason is that most software testers are not necessarily trained in formal methods and may not know how to use the technique, but this situation is changing due to three factors. The first is that many evidences have demonstrated the power of formal specification in enhancing software quality [8] and more and more companies become interested in training their employees in formal specification techniques (e.g., in Japan). The second factor is that a growing number of practitioners have become using formal specification techniques in practice, especially in the domain of dependable systems [4]; this demonstrates the feasibility and effectiveness of formal specification techniques in industry. The third is the progress made in supporting formal specification construction by software tools based on formal specification patterns [9]. With such a technique, one does not need to directly write formal specifications; instead, one only needs to express one's ideas in a structured English and the supporting tool will gradually automatically produce the formal specification. With the above progresses, it is possible for high level testing consultants or company practitioners to efficiently apply our method in practice.

6. Related Work

Our work falls into the category of model-based testing (MBT), but differs from existing approaches in the way of deriving models. In this section, we briefly overview the related work in this area and compared it to our approach.

There have been extensive research on and application of MBT over the last three decades [10], and the work can be classified into four categories: general discussion of MBT, MBT based upon graphical models, MBT based on formal specifications, and automation in MBT. El-Far and Whittaker give a general introduction to model-based testing in [11], discussing its underlying principle, process, and techniques. Apfelbaum and Doyle argue that a finite state machine MBT can reduce cost and time, and can increase software quality, but pointed out that the technique also faces challenges in acceptance [12]. Offutt *et al* describe an approach to generating test cases from UML Statecharts for components testing [13]. Hartmann *et al* extend the approach for integration testing and provide effective tool support for test automation [14]. Bernot *et al* set up a theoretical foundation and a tool support for specification-based testing, explaining how a formal specification can serve as a base for test case generation and as an oracle for test result evaluation [15]. Dick and Faivre propose to transform pre-condition into a disjunctive normal form (DNF) and then use it as the basis for test case generation [16]. Stocks and Carrington suggest that test templates be defined as the base for test case generation and a large test template is divided into smaller templates for generating more detailed test cases [17]. In order to deal with the practical challenges MBT techniques have encountered due to the inconsistency between the component interfaces in the model and in the program, Blackburn *et al* present an interface-driven MBT approach that combines requirement modeling and component interface analysis to support automated test case and test driver generation [18]. In spite of tremendous efforts by many researchers, MBT is still difficult to be used for large scale systems because of its complexity and the potential inconsistencies in both component interface and system architectures. It is also unsuitable for testing programs, such as legacy code, which does not have source code available. To tackle this problem, Bertolino *et al* describe an anti-model-based testing approach in [19]. The essential idea of this approach shares with our testing approach presented in this paper; it suggests to use some test cases to execute the program under test and observe the traces of executions, and then try to synthesize and abstract model of the system. Compared to our work in this paper, Bertolino *et al* neither advocate the use of formalization for the abstract model, nor discuss in detail about how all the necessary traces are observed and represented and how the traces are synthesized into an abstract model. Our novel contribution in this paper is to present a systematic method for identifying all interface scenarios, formalizing requirements into formal operation specifications whose interfaces are consistent with the corresponding ones of the program, and for testing programs based upon the formal specifications.

7. Conclusion and Future Work

We present a systematic approach to testing programs with no available source code. The approach includes three steps: (1) identifying important interface scenarios by executing the program with test cases, (2) formalizing the informal requirements into formal specifications, and (3) generating test cases from the formal specifications to test the program. The benefit of our testing approach is to allow the tester to thoroughly understand the desired behaviors of the system and to provide a precise model for testing it. We have also applied our approach to an IC Card system to gain experience and to learn potential challenges in practice.

In the future, we will continue to work along the direction set in this paper to make the testing approach more mature and effective. In particular, we will refine and improve the techniques for discovering interface scenarios, formalizing user requirements, and generating test cases. We also plan to conduct a rigorous experiment on the effectiveness of our approach in industrial environment. To enhance the efficiency of the testing activities, we will also work on the tool support.

Acknowledgments. This research is supported in part by NII Collaborative Research Program, 973 program Grant No.2010CB328102, and NSFC under Grant Nos. 61133001, 60910004. We would like to thank all the students involved in this project for their contributions to the application of the testing approach and cooperations in data collection.

References

1. B. Beizer, *Black-Box Testing*, John Wiley and Sons, Inc., 1995.
2. S. Liu, J. A. McDermid, and Y. Chen, "A Rigorous Method for Inspection of Model-Based Formal Specifications", *IEEE Transactions on Reliability*, vol. 59, no. 4, pp. 667–684, December 2010.
3. M. Li and S. Liu, "Automated Functional Scenarios-Based Formal Specification Animation", in *Proceedings of 19th Asia-Pacific Software Engineering Conference (APSEC 2012)*. December 4-7 2012, p. to appear, IEEE CS Press.
4. J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal Methods: Practice and Experience", *ACM Computing Surveys*, vol. 41, no. 4, pp. 1–39, 2009.
5. S. Liu, *Formal Engineering for Industrial Software Development Using the SOFL Method*, Springer-Verlag, ISBN 3-540-20602-7, 2004.
6. The VDM-SL Tool Group, *Users Manual for the IFAD VDM-SL Tools*, The Institute of Applied Computer Science, February 1994.
7. S. Liu et al, "An Automated Approach to Specification-Based Program Inspection", in *Proceedings of 7th International Conference on Formal Engineering Methods (ICFEM2005)*, Manchester, UK, 1-4 Nov. 2005, pp. 421–434, LNCS 3785, Springer-Verlag.
8. D. Craigen, S. Gerhart, and T. Ralston, *Industrial Applications of Formal Methods to Model, Design and Analyze Computer Systems: an International Survey*, Noyes Data Corporation, USA, 1995.
9. X. Wang, S. Liu, and H. Miao, "A Pattern-Based Approach to Formal Specification Construction", in *Proceedings of 2011 International Conference on Advanced Software Engineering and Its Application*, Jeju, Korea, December 13-15 2011, CCIS 257, pp. 159–168, Springer-Verlag.
10. S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-Based Testing in Practice", in *Proceedings of Proceedings of the 21st International Conference on Software Engineering*. 1999, pp. 285–294, IEEE Computer Society Press.

11. I. K. El-Far and J. A. Whittaker, *Model-based Software Testing*, Wiley, 2001.
12. L. Apfelbaum and J. Doyle, "Model-Based Testing", in *Proceedings of Software Quality Week Conference*. May 1997, LNCS, Springer-Verlag.
13. J. Offutt and A. Abdurazik, "Generating Test Cases from UML Specifications", in *Proceedings of Second International Conference on UML'99*, Robert France and Bernhard Rumpe, Eds., Fort Collins, CO, USA, October 28-30 1999, pp. 416–429, Springer.
14. J. Hartmann, C. Imoberdorf, and M. Meisinger, "UML-Based Integration Testing", in *Proceedings of 2000 ACM SIGSOFT Symposium on Software Testing and Analysis*, Portland, Oregon, USA, 2000, pp. 60–70, ACM Press.
15. Gilles Bernot, Marie Claude Gaudel, and Bruno Marre, "Software testing based on formal specifications: a theory and a tool", *Software Engineering Journal*, pp. 387–405, November 1991.
16. 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*, Odense, Denmark, 1993, pp. 268–284, LMCS 670, Springer-Verlag.
17. P. Stocks and D. Carrington, "A Framework for Specification-Based Testing", *IEEE Transactions on Software Engineering*, vol. 22, no. 11, pp. 777–793, November 1996.
18. M. R. Blackburn, R. D. Busser, and A. M. Nauman, "Interface-Driven, Model-Based Test Automation", in *Proceedings of 2005 International Conference on Software Testing, Analysis and Review*. Nov. 14-18 2002, pp. 27–30, Software Quality Engineering.
19. A. Bertolino, A. Polini, P. Inverardi, and H. Muccini, "Towards Anti-Model-Based Testing", in *Proceedings of 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (Extended Abstract)*, June 28 - July 1 2004, pp. 124–125.

Shaoying LIU is Professor of Software Engineering at Hosei University, Japan. He received the Ph.D degree in Computer Science from the University of Manchester, U.K in 1992. His research interests include Formal Methods and Formal Engineering Methods for Software Development, Specification Verification and Validation, Specification-based Program Inspection, Automatic Specification-based Testing, and Intelligent Software Engineering Environments. He has published a book titled "Formal Engineering for Industrial Software Development Using the SOFL Method" with Springer-Verlag, four edited conference proceedings, and over 120 academic papers in refereed journals and international conferences. He has recently served as General Co-Chair of ICFEM 2012 and PC member for numerous international conferences. He is on the editorial board for the Journal of Software Testing, Verification and Reliability (STVR) and ISRN Software Engineering. He is a Fellow of British Computer Society, a Senior Member of IEEE Computer Society, and a member of Japan Society for Software Science and Technology.

Wuwei SHEN received the PhD degree in computer science from the Department of Electrical Engineering and Computer Science, the University of Michigan, Ann Arbor in 2001. He is currently an associate professor at Western Michigan University. His research interests include object-oriented software model design and analysis, software model consistency checking, model-based software testing, software error detection, and the management of software artifacts.

Shin NAKAJIMA received Ms degree in 1981 and Ph.D degree in 2000 both from The University of Tokyo. He is currently professor at National Institute of Informatics, Tokyo, Japan. His current interests include formal methods, automated verification, and software modeling. Dr. Nakajima is a member of ACM, JSSST, and IPSJ.

Received: March 1, 2012; Accepted: December 5, 2012.