Acceptance Testing based on Relationships among Use Cases

Susumu Kariyuki,
Hironori Washizaki,
Yoshiaki Fukazawa
Waseda University
Tokyo, Japan
washizaki@waseda.jp

Atsuto Kubo
Aoyama Media Laboratory
Tokyo, Japan
a.kubo@fuka.info.waseda.ac.jp

Mikio Suzuki
TIS Inc.
Tokyo, Japan
mickey_suzuki@ybb.ne.jp
mikio.suzuki@gmail.com

Abstract

In software development using use cases, such as use-case-driven object-oriented development, test scenarios for the acceptance test can be built from use cases. However, the manual listing of execution flows of complex use cases sometimes results in incomplete coverage of the possible execution flows, particularly if the relationships between use cases are complicated. Moreover, the lack of a widely accepted coverage definition for the acceptance test results in the ambiguous judgment of acceptance test completion. We propose a definition of acceptance test coverage using use cases and an automated generation procedure for test scenarios and skeleton codes under a specified condition for coverage while automatically identifying the execution flows of use cases. Our technique can reduce the incomplete coverage of execution flows of use cases and create objective standards for judging acceptance test completion. Moreover, we expect an improvement of the efficiency of acceptance tests because of the automated generation procedure for test scenarios and skeleton codes.

1. Introduction

An acceptance test is carried out by users who take the initiative in confirming whether the developed system precisely performs the requirements of the users.[2)] As a method of representing functional requirements, use cases have been used.[3)] Use cases are descriptions of external functions of a system from the viewpoint of the users and external systems (both are called actors). When functional requirements are identified using use cases, the execution flows of the use cases are manually identified and test scenarios are generated on the basis of the identified execution flows. When the number of use cases is large, such as several tens or several hundred, manual identification of the execution flows may become difficult and incomplete coverage of the execution flows may occur. The likelihood of this particularly increases when the relationship between use cases is complicated, leading to a decrease in the reliability of the acceptance test.

Test coverage has been used as a measure for judging the completion of an acceptance test. Test coverage refers to the ratio of the number of test items that were actually tested to the total number of test items that should have been tested.[4)] Examples used to define test coverage include statement coverage and branch coverage in a white box test. However, a widely accepted definition for test coverage has not yet been established for acceptance tests targeting the test scenarios identified from use cases; the judgment of acceptance test completion largely depends on individuals. As a result, the coverage of an acceptance test differs each time, leading to a decrease in the reliability of the acceptance test.

In this study, we define the coverage of the acceptance test targeting the test scenarios identified on the basis of use case descriptions. Using this definition, we propose a technique of automatically generating an acceptance test environment with a particular test framework format targeting the functional test using the use case descriptions, a domain model, and the test coverage, as input. Our proposed technique can automatically identify the test scenarios using use cases; incomplete coverage of the test scenarios targeted in the acceptance test can be prevented, and we can remove the dependence of the judgment of acceptance test completion on individuals. In addition, we can expect an improvement in the efficiency of the acceptance test by automatically generating its test scenarios and skeleton codes.

2. Acceptance test using use cases and its problems

2.1 Domain model and relationships among use cases

Use cases are descriptions of the external functions of a system from the viewpoint of actors. Use cases do not deal with the internal structure of a system; rather, they are used to organize the requirements of the actors in the system. A use case description provides details of a use case in texts.[3)] An action described in the flow of a use case description is called a use case step. In the following explanation, a use case step is simply called a step. In this study, a sequence of steps that may be executed by a combination of steps is defined as an execution flow. **Figure 1** shows an example of a use case diagram for a hotel reservation system developed using unified modeling language (UML) 2.0.[5)] **Figures 2-6** show use case descriptions of “Reserve facility,” “Reserve room,” “Log in,” “Intercept,” and “Encrypt,” respectively.

Use case descriptions can be reused and steps can be added by using the relationship between use cases (such as include, generalization, and extend) defined in UML 2.0 without changing the existing use cases.

Include: The include relationship is a relationship in which a certain use case includes the steps of another use case. In Fig. 1, the use case “Log in” includes the use case “Encrypt.” This relationship is represented by the text in Fig. 4, i.e., “Include: The step ‘The system codes the input information and transmits it’ in the flow ‘Log in’ executes the use case ‘Encrypt.’”

Generalization: The generalization relationship is a relationship of “is-a” between use cases. In Fig. 1, the use case “Reserve room” is a use case “Reserve facility.” This relationship is represented by the text in Fig. 3, i.e., “Use case: Specializing the use case ‘Reserve facility.’”

Extend: The extend relationship is a relationship in which steps of another use case are added to an existing use case. The extended use case has extension points, at which additional steps can be inserted. In the extended use case, the point of insertion is specified by the extension point. In Fig. 1, the use case “Log in” extends the use case “Reserve facility.” As an extension point, “Start process” is specified. Examples of the application of the extend relationship include misuse cases and security use cases.[6)] Misuse cases are the use cases described from the viewpoint of an opposing actor (attacker) of the system under design. Security use cases are the use cases that mitigate the effects of the misuse cases. According to ref. 6), threatening a use case by a misuse case is defined as “threaten,” and mitigating the threat of the misuse case by a security use case is defined as “mitigate.” Also, it is stated that threaten can be defined as a special type of extend. In Fig. 1, the use case “Intercept” threatens the use case “Log in,” which is represented by a <<threaten>> arrow. Similarly, the use case “Encrypt” mitigates the threat of the use case “Intercept,” which is represented by a <<mitigate>> arrow.

A domain model is a model that represents an essential and target relationship between concepts in a target system for the purpose of precisely understanding the problem domain that is targeted by the system developer.[7)] **Figure 7** shows an example of a domain model corresponding to the hotel reservation system shown in Fig. 1. In the figure, the main concepts related to the hotel reservation system, such as “Room” and “Reservation,” are shown.

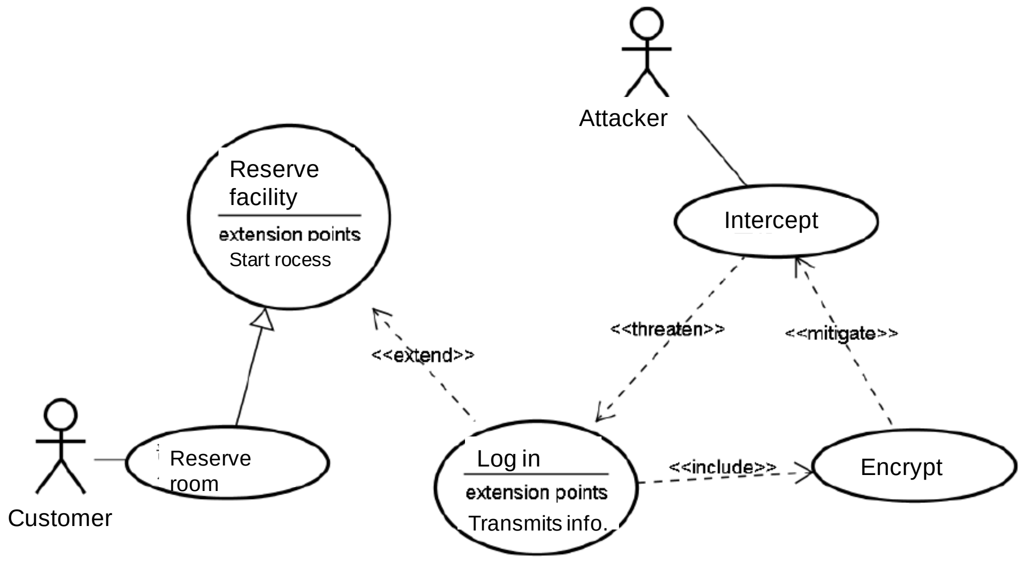


Figure 1. Example of use case diagram

Use case: Reserve facility

Main flow name: Reserve facility

1. The system displays available facilities.
2. The customer selects a facility.
3. The system displays the total fee for the selected facility.
4. The system reduces the total number of available facilities on the database.
5. The system makes a new reservation for the selected facility.
6. The system displays a reservation confirmation number.
7. The system terminates the use case.

Alternative flow:

Alternative flow name: Duplicate application

After the step "The system reduces the total number of available facilities on the database," the flow branches to "duplicate application."

A1. 1. If the customer indicates that the new reservation is a duplicate one, the system terminates the use case.

Precondition: When the same reservation (same customer, e-mail address, start and end days) exists

Extension point: "Start process"

The step "The system displays available facilities" in the flow "Reserve facility"

Figure 2. Use case description of "Reserve facility"

Use case: Reserve room

Use case: Specializing the use case “Reserve facility”

Main flow name: Reserve facility

1. The system displays available facilities.
2. The customer chooses to make a reservation for a room.
3. The customer selects the type of room.
4. The customer confirms the fee for the room.
5. The system displays the total fee for the selected facility.
6. The system reduces the total number of available facilities on the database.
7. The system makes a new reservation for the selected facility.
8. The system displays a reservation confirmation number.
9. The system terminates the use case.

Alternative flow:

Alternative flow name: Duplicate facility

After the step “The system reduces the total number of available facilities on the database,” the flow branches to “duplicate facility.”

A1. 1. If the customer indicates that the new reservation is a duplicate one, the system terminates the use case.

Precondition: When the same reservation (same customer, e-mail address, start and end days) exists

Figure 3. Use case description of “Reserve room”

Use case: Log in

Main flow name: Log in

1. The system requests the customer to input ID and password.
2. The customer inputs ID and password.
3. The system encrypts the input information and transmits it.
4. The system authenticates the ID and password.
5. The system executes a base use case.

Alternative flow: Alternative flow name: Authentication failure

After the step “The system authenticates the ID and password,” the flow branches to “authentication failure.”

A1. 1. The system terminates the base use case.

Precondition: When the input ID is not registered or the input ID does not match the registered password

Include:

The step “The system encrypts the input information and transmits it” in the flow “Log in” executes the use case “Encrypt.”

Extend: Prior to “Start process”

Extension point: “Transmits information”

The step “The system encrypts the input information and transmits it” in the flow “Log in”

Figure 4. Use case description of “Log in”

Use case: Intercept

Main flow name: Intercept

1. An attacker attempts to intercept the transmitted information.

Extend: After “transmitting information”

Figure 5. Use case description of “Intercept”

Use case: Encrypt

Main flow name: Encrypt

1. The system encrypts the information to be transmitted.

Figure 6. Use case description of “Encrypt”

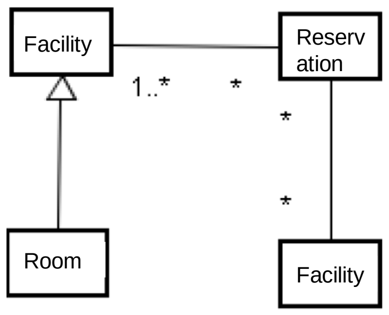


Figure 7. Example of domain model corresponding to Fig. 1

2.2 Acceptance test

Use cases represent the requirements of the users. Therefore, it is possible to test whether the system precisely implements the requirements of the users by testing the system using the test scenarios prepared using use cases. One such test is an acceptance test. An acceptance test is carried out by users who take the initiative in confirming whether the software under development really implements the requirements of the users. In this study, we focus on a functional test, one type of acceptance test, to test whether the functions required by the users are implemented.

2.3 Framework for integrated test

The framework for integrated test (FIT)[8)] was developed by Ward Cunningham and is a test framework used for the acceptance test. FIT has the following characteristics.

- Automatic testing is possible.
- Users unfamiliar with programming can generate test scenarios because test scenarios are generated using an HTML table format.
- FIT is compatible with many languages including Java and Ruby.

FIT is composed of the following elements.

- Runner: a test execution program
- Fixture: defining the correspondence between a test description file and a test target program
- Test description file: a file describing a test scenario
- Test result file: a file including a test result

Figure 8 shows the outline of the test using FIT.

(1) Users develop a test description file in HTML format. Concretely, a user describes the test contents to check the program functions required by the users using an HTML table format in a test description file. Here, “CheckAuthentication” in Fig. 8 is used as an example. The user specifies the fixture “checking authentication” on the first row. Also, the user indicates that three items, i.e., userName, password, valid(), are dealt with as the test items on the second row. On the third or later rows, test scenarios are described. On the third row, “Jack” and “waseda” are assigned to userName and password, respectively. The third column indicates whether the result of executing the valid() method is true.

(2) The developer develops a product program so that the contents of the table in the test description file are satisfied. The developer also develops fixtures to connect the table contents with the product program. The fixture “checking authentication” in Fig. 8 has variables such as userName and password. The developer writes codes to test the required functions using userName, password, and the valid() method.

(3) The user executes the acceptance test by inputting a test description file to FIT. The user confirms the result of the acceptance test by browsing the test result file that is output from a runner.

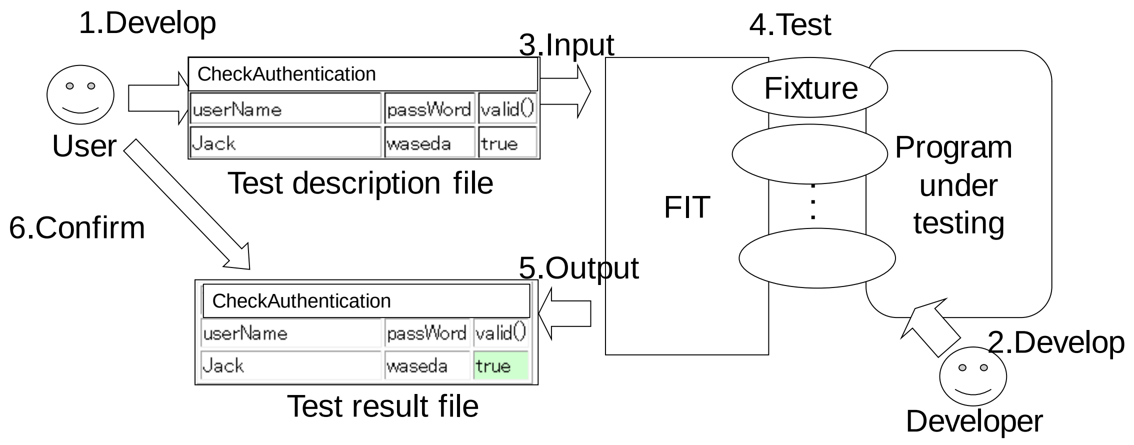


Figure 8. Outline of test using FIT

2.4 Problems associated with acceptance test using use cases

References 9) and 10) explain the conventional techniques of generating test scenarios by identifying execution flows in use cases. In these references, the execution flows of the use cases are manually identified, and test scenarios are generated on the basis of these execution flows. These techniques have the following two problems.

Problem 1: A widely accepted definition for the coverage of the acceptance test using test scenarios identified from use cases has not yet been established. Therefore, in the development of test scenarios covering all the use cases, one person may develop test scenarios including all the steps at least once; another may develop test scenarios including all the branches at least once. This indicates that the judgment of acceptance test completion may depend on the individual, leading to a decrease in the reliability of the acceptance test.

Problem 2: A complicated relationship between use cases may lead to the incomplete coverage of execution flows. In large-scale development, the number of use cases is large, such as several tens or several hundred; moreover, the relationship between use cases tends to become complicated. In addition, the requirement for security has been intensifying, and misuse cases and security use cases, in addition to general use cases, should be taken into consideration. In this case, the incomplete coverage of execution flows tends to occur because the execution flows involving misuse cases and security use cases are more complicated that those involving only general use cases.

3. Definition of coverage on the basis of execution flows of use cases

We propose solutions to the two problems explained in section 2.4.

Solution for problem 1: First, we define the coverage for the acceptance test targeting test scenarios identified from use case descriptions. Following this definition, the dependence of the judgment of acceptance test completion on the individual can be avoided.

Solution for problem 2: We automatically identify the execution flows of use cases using a system for generating the test scenarios and skeleton codes for the acceptance test environment of a particular test framework format**.** In this study, we adopted FIT in consideration of its compatibility with various languages and its familiarity. When execution flows of the use cases are automatically identified, the incomplete coverage and errors of the test scenarios can be avoided.

The solutions to problems 1 and 2 are discussed in sections 3 and 4, respectively.

3.1 Identification of all execution flows in use cases

First, we generate a graph of flows of use cases and define the test coverage using the diagram. The procedure for generating the graph flows is explained below.

Assuming that the sets of all the steps and the transitions between steps are *S* and *E*, respectively, then use case *uc* is expressed as

$$S = \{s_1, s_2, \ldots\}$$
$$E \subseteq S \times S$$
$$uc = (S, E)$$

Here, we define *n* as the number of steps that are defined. It is assumed that |*X*| and X_i are the number of elements in a set *X* and the set of edges with step s_i as a destination of the transition, respectively. When the number of elements in X_i is two or more, X_i is the set of branches from s_i, and the set *B* of branches is expressed as

$$X_i = \{(s_i, s_j) \mid s_j \in (S - \{s_i\}) \wedge (s_i, s_j) \in E\}$$
$$B = \{\bigcup_{i=1}^{n} X_i \mid |X_i| \geq 2\}$$

The set of execution flows generated by the following algorithm, i.e., the set of all execution flows, is denoted as *F*.

Function name: generate_allFlows

Outline: Obtains the set of all execution flows in a certain use case

Input: Use case

Output: The set of all execution flows in the input use case

```
$generate\_allFlows(uc:$ Use case $)$ {
    $s_1$ := The first step in uc;
    $F := \phi;$
    $Oneflow := \phi;$
    $traverse(s_1, Oneflow, F);$
    $return\ F;$
}
```

Function name: traverse

Outline: Obtains the set of execution flows that are executable using the input steps

Input: A step and a description of the execution flows in a use case, and the set of all execution flows in a use case

Output: None

```
$traverse(s:$ Step $, Oneflow:$ Exec. flow$, All$ : Set of exec. flows$)$ {
    $nextSteps := \{s_i \mid (s, s_i) \in E\};$
    $if\ (nextSteps \neq \phi)\{$
        $All := All \bigcup \{Oneflow\};$
    $\}\ else\ if\ \{$
        $for\ each\ (s_j \in nextSteps)\ \{$
            $Oneflow := Oneflow \bigcup \{(s, s_j)\};$
            $traverse(s_j, Oneflow, All);$
        }
    }
}
```

When only the use case description shown in Fig. 2 is considered and input to the function “generate_allFlows,” then *S*, *E*, and *F* are obtained as follows. Here, each step is described using the step number instead of its content.

$$S = (s_1,\ s_2,\ s_3,\ s_4,\ s_5,\ s_6,\ s_7,\ s_{A1.1})$$
$$E = \{(s_1,\ s_2),\ (s_2,\ s_3),\ (s_3,\ s_4),\ (s_4,\ s_5),\ (s_5,\ s_6),\ (s_6,\ s_7),\ (s_4,\ s_{A1.1})\}$$
$$F = \{\ \{(s_1,\ s_2),\ (s_2,\ s_3),\ (s_3,\ s_4),\ (s_4,\ s_5),\ (s_5,\ s_6),\ (s_6,\ s_7)\},\ \{(s_1,\ s_2),\ (s_2,\ s_3),\ (s_3,\ s_4),\ (s_4,\ s_{A1.1})\}\ \}$$

Figure 9 shows the procedure for identifying the change in execution flows depending on the relationship between use cases. The use cases shown in Fig. 1 are used as an example. Diagrams of the execution flows of the use cases are generated from the use case descriptions that are input.

(1) If there is a generalized use case, its execution flow is overwritten with the execution flow of the specialized use case in the unit of step. **Figure 10** shows a diagram after the change by generalization for this example, where the gray area shows the changes after generalization. In this case, the step “The customer selects a facility” in the use case “Reserve facility” is replaced with the steps “The customer chooses to make a reservation for a room,” “The customer selects the type of room,” and “The customer confirms the fee for the room” in the use case “Reserve room.”

(2) If there is an including use case, the step referring to the included use case is overwritten with the execution flow of the included use case. **Figure 11** shows a diagram after the change by inclusion, where the gray area shows the changes after inclusion. In this case, the step “The system codes the input information and transmits it” in the use case “Log in” is replaced with the step “The system codes the information to be transmitted” in the use case “Encrypt,” to which the use case “Log in” refers. This means that the likelihood of the malicious use of the tapped information is mitigated by executing this step.

(3) If there is an extending use case, its execution flow is added to the step specified as the extension point. **Figure 12** shows a diagram after the change by extension, where the gray area shows the changes after extension. The flow “Intercept” is added after the step “The system codes the input information and transmits it” in the use case “Log in,” which is specified as the extension point. This indicates a threat that an attacker attempts to tap the information transmitted during the Log in of a customer. Similarly, the flow “Log in” is added before the step “The system displays available facilities” in the use case “Reserve facility,” which is specified as the extension point.

The threat relationship can be interpreted as a special type of the extend relationship that is defined in UML 2.0.6) Therefore, the threat relationship is treated as an extend relationship when specifying the execution flows. In addition, the mitigate relationship indicates that the effect of a threat is mitigated and does not affect the execution flows; therefore, the mitigate relationship is considered nonexistent when specifying the execution flows.

```
Generates a graph of only the execution flows in a target UC

if (there is a generalization relationship){

  The execution flow in the generalized UC is overwritten with that in the target UC in the unit of step;

}

if (there is an include relationship){

  The step referring to the included UC is overwritten with the execution flow of the included UC;

}

if (there is an extend relationship){

  The execution flow of the extending UC is added at the extension point;

}
```

Figure 9. Identification procedure of execution flow

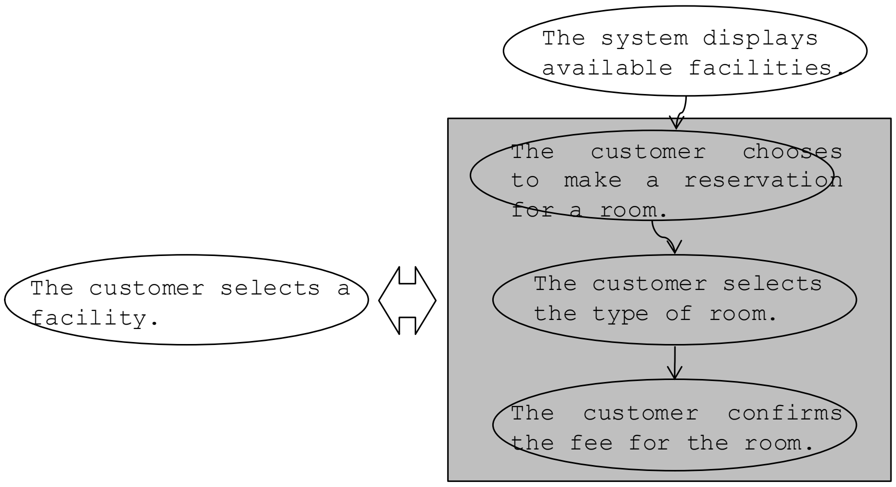

The system displays available facilities.
The customer selects a facility.
The customer chooses to make a reservation for a room.
The customer selects the type of room.
The customer confirms the fee for the room.

Figure 10. Change of flow by generalization

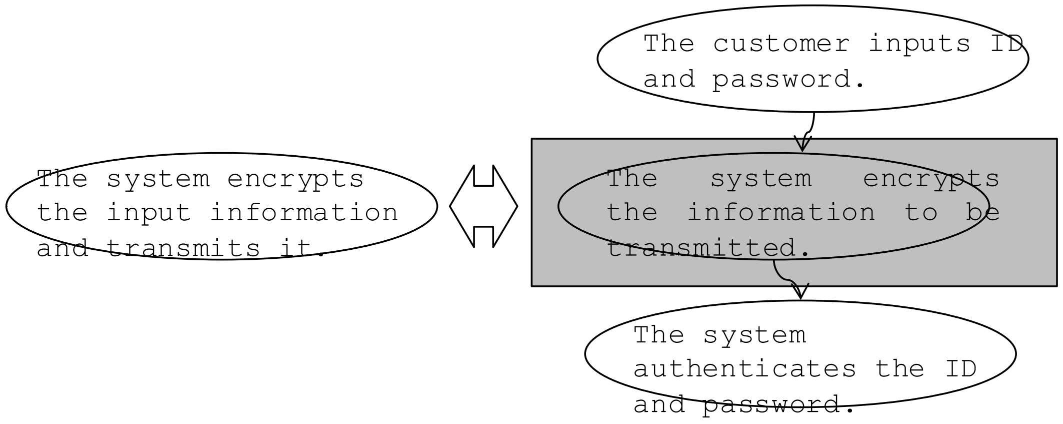

The customer inputs ID and password.
The system encrypts the input information and transmits it.
The system encrypts the information to be transmitted.
The system authenticates the ID and password.

Figure 11. Change of flow by include

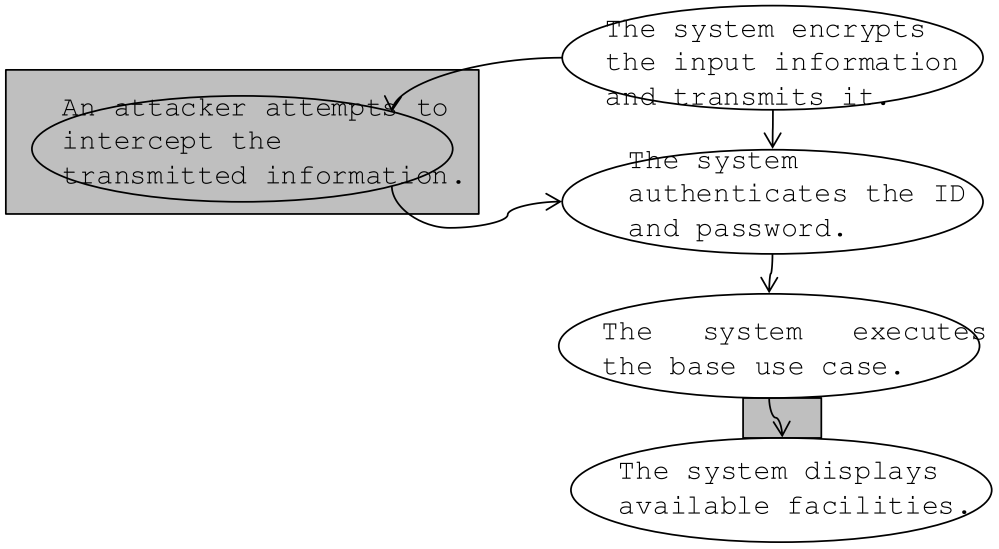

The system encrypts the input information and transmits it.
An attacker attempts to intercept the transmitted information.
The system authenticates the ID and password.
The system executes the base use case.
The system displays available facilities.

Figure 12. Change of flow by extend

3.2 Coverage of acceptance test using use cases

We define the coverage of the acceptance test on the basis of test scenarios identified from use cases as follows, using the diagrams of execution flows created in section 3.1. Focusing on the similarity between the flow of the use case description and a program flow, we propose three coverages, step coverage, branch coverage, and all-execution-flows coverage, referring to the definition of the test coverage of a program flow. A loop is treated as a branch, similarly to in the control flow of a program. Note that only one iteration of a loop is treated and subsequent iterations are not taken into consideration.

Definition 1: Step coverage, C'_0

C'_0 denotes the percentage of actually executed steps among all the use case steps. Assuming that the set of steps executed in a test case t included in the set of test cases T is S_t, C'_0 is defined as

$$C'_0 = \frac{|S_t|}{|S|} \times 100$$

Definition 2: Branch coverage, *C'*$_1$

C'$_1$ denotes the percentage of actually executed branches among all the branches. Assuming that the set of branches executed in a test case *t* included in the set of test cases *T* is B_t, *C'*$_1$ is defined as

$$C'_1 = \frac{|B_t|}{|B|} \times 100$$

Definition 3: All-execution-flows coverage, *C'*$_\infty$

C'$_\infty$ denotes the percentage of actually executed combinations of flows among all the combinations of flows. Assuming that the set of execution flows executed in a test case *t* included in the set of test cases *T* is F_t, *C'*$_\infty$ is defined as

$$C'_\infty = \frac{|F_t|}{|F|} \times 100$$

C'$_1$ for the use case shown in **Fig. 13** is calculated as an example [extracted from ref. 3) after partial modification]. **Figure 14** shows a graph flows. Assuming that the set of branches actually executed is $\{c_2\}$, *C'*$_1$ is calculated as

$$\begin{aligned} C'_1 &= \frac{|\{c_2\}|}{|\{c_2, c_4\}|} \times 100 \\ &= 1/2 \times 100 \\ &= 50\% \end{aligned}$$

Here, *C'*$_0$ and *C'*$_\infty$ are 80 and 50%, respectively.

Use case: Keeps the client waiting for access to a resource

Main flow name: Keeps the client waiting for access to a resource

1. The resource lock temporary disrupts the process of a client.
2. The client waits for the resumption.
3. The process of the client resumes.

Exceptional flow:

A1. Expiration of latency time

In step 2 of the main flow, when the latency time expires

1. The system informs the client by a signal that the requested access is not permitted.
2. The system terminates the use case.

Figure 13. Use case description of "Keeps the client waiting for access to resource"

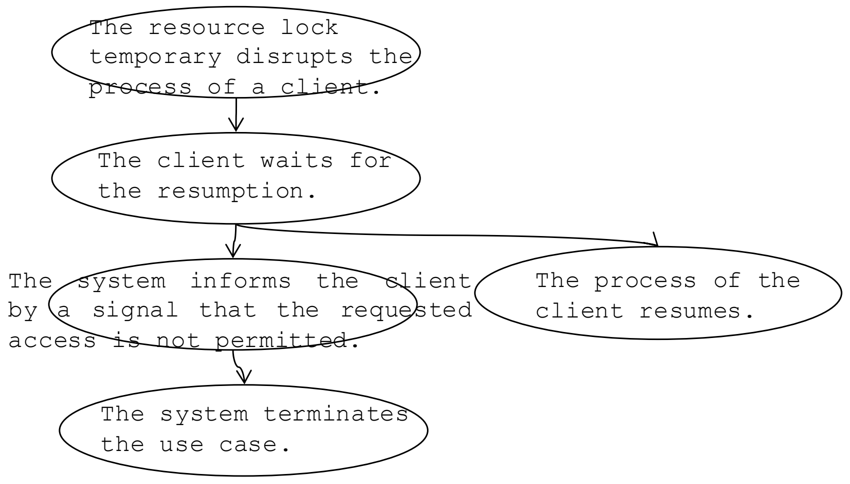

The resource lock temporary disrupts the process of a client.
The client waits for the resumption.
The system informs the client by a signal that the requested access is not permitted.
The process of the client resumes.
The system terminates the use case.

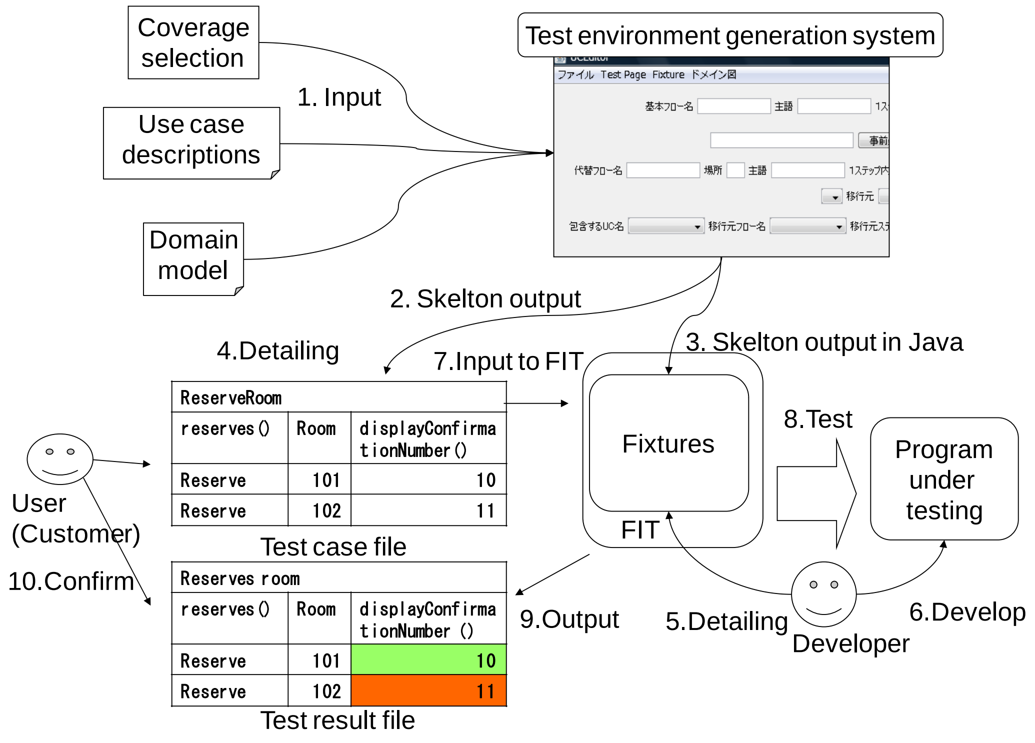
Figure 14. Execution flow of use case of "Keeps the client waiting for access to resource"

4. Test environment generation system

The manual identification of execution flows may result in incomplete coverage and errors. We propose a system for generating the test scenarios and skeleton codes for an acceptance test environment with a particular test framework format (hereafter, proposed system). Incomplete coverage and errors can be eliminated using the proposed system because the execution flows of the use cases are automatically identified. In this study, we adopted test framework FIT in consideration of its compatibility with various languages and its familiarity.

4.1 Structure and testing method of the proposed system

The manual identification of execution flows of use cases may result in incomplete coverage and errors, particularly when the relationship between use cases is complicated. As a solution to these problems, we propose a system for generating test scenarios and skeleton codes for an acceptance test environment with a particular test framework format through the automatic identification of the execution flows using the use case description, domain model, and coverage as the input. **Figure 15** shows an outline of the proposed system. The processes involved in the acceptance test using the proposed system are divided into the following three steps: (1) a phase to input data to the proposed system (input phase), (2) a phase to detail the output from the proposed system (detailing phase), and (3) an acceptance test phase involving FIT.



ReserveRoom		
reserves()	Room	displayConfirmationNumber()
Reserve	101	10
Reserve	102	11

Reserves room		
reserves()	Room	displayConfirmationNumber ()
Reserve	101	10
Reserve	102	11

Figure 15. Outline of proposed system

(1) Input phase

Three types of data, the selection of coverage, use case descriptions, and the domain model, are input to the proposed system. By inputting the domain model to the proposed system, the elements used as input values of the test can be extracted. For example, when a test scenario that a “customer” Saitoh reserves “room number” 3022 in the use case “Reserve room,” the elements “customer” and “room number” can be extracted. Upon receiving the input, the proposed system outputs the two data, i.e., a skeleton test scenario (HTML table format) and skeleton fixture (execution program to test the test scenario).

(2) Detailing phase

In the test scenario output from the proposed system, the input items and steps in a domain model are described. In the test scenarios, input items, such as “customer” and “room number” in a domain model, are described; however, concrete values, such as Saitoh for “customer” and 3022 for “room number” are not described. Similarly, operations having the same names as those of the steps are defined in the fixture; however, function calls of the functions that should be tested are not described. Therefore, the concrete values of the test items should be added to the test scenario, and the required function calls should be added in the fixture. In the detailing phase, the following two files are detailed so that they can be directly used in the test.

- Skeleton test scenario: This is modified by users.
- Skeleton fixture: This is modified by the staff in charge of the test.

(3) Acceptance test phase involving FIT

After manually detailing the test scenario and fixture, the acceptance test by FIT is carried out.

4.2 Generation of skeleton test scenarios using a graph of execution flow

In this section, the following two topics are explained.

- Generation of skeleton test scenarios obtained in the input phase
- Detailing of the test scenarios carried out in the detailing phase

The following two topics are explained in section 4.3.

- Generation of skeleton fixture obtained in the input phase
- Detailing of the fixture carried out in the detailing phase

Using the graph of execution flows obtained in section 3.1, skeleton test scenarios are generated. Then, we select a test scenario output using the graph of execution flows on the basis of the selection of the three types of coverage defined in section 3.2. The test scenario selection algorithm is explained next.

Function name: generate_testScenarios

Outline: Obtains a graph flow of a use case based on a particular coverage

Input: Coverage and use case

Output: The set of execution flows of a use case based on the input coverage

```
generate_testScenarios(criteria : Coverage criteria , uc : Use case ) {
    Flows := generate_allFlows(uc) ;
    if(criteria = All-execution -flows coverage ){
        return Flows;
    }
    coverageFlows := $\phi$;
    for each ($f \in$ Flows) {
        coverageFlows := coverageFlows $\bigcup$ {f};
        Edges := getEdges(coverageFlows) ;
        if (criteria = Branch coverage ) {
            if(E = Edges){
                return coverageFlows;
            }
        } else if (criteria = Step coverage ) {
            if(S = getSteps(Edges)){
                return coverageFlows;
            }
        }
    }
}
```

Function name: getEdges

Outline: Extracts the set of transitions between steps from the set of diagrams of execution flows. For example, if we input {{(s_1, s_2), (s_2, s_3)},{(s_1, s_A)}}, we obtain {(s_1, s_2), (s_2, s_3), (s_1, s_A)}.

Input: The set of diagrams of execution flows

Output: The set of transitions between steps shown in the input diagrams of execution flows

```
getEdges(Flows : Set of execution flow graphs ) {
    Edges := $\phi$;
    for each ( f $\in$ Flows) {
        for each ( e $\in$ f) {
            Edges := Edge $\bigcup$ {e};
        }
    }
    return Edges;
}
```

Function name: getSteps

Outline: Extracts the steps from the set of transitions between steps. For example, if we input {(s_1, s_2), (s_2, s_3),(s_1, s_A)}, we obtain {s_1, s_2, s_3, s_A}.

Input: The set of transitions between steps

Output: The set of steps shown in the set of transitions between steps

```
getSteps(Edges : Set of transitions ) {
    Steps := $\phi$;
    for each ( e | e $\in$ Edges, e = ($s_i$ , $s_j$) ) {
        Steps := Steps $\bigcup$ {$s_i$};
        Steps := Steps $\bigcup$ {$s_j$};
    }
    return Steps;
}
```

For example, the execution flows generated by generate_testScenarios are the same when either step coverage, branch coverage, or all-execution-flows coverage is specified for the use case shown in Fig. 2, and we obtain

F = {{(s_1, s_2), (s_2, s_3), (s_3, s_4), (s_4, s_5), (s_5, s_6), (s_6, s_7)}, {(s_1, s_2), (s_2, s_3), (s_3, s_4), (s_4, $s_{A1.1}$)}}.[*1]

[*1]When step coverage or branch coverage is specified for generate_testScenarios without considering the relationship between use cases for the use case shown in Fig. 38, we obtain

$\{\{(s_1, s_2), (s_2, s_{A1.1}), (s_{A1.1}, s_{A1.2}), (s_{A1.2}, s_{A1.3}), (s_{A1.3}, s_3), (s_3, s_4)\}\}$ for step coverage and $\{\{(s_1, s_2), (s_2, s_3), (s_3, s_4)\}, \{(s_1, s_2), (s_2, s_{A1.1}), (s_{A1.1}, s_{A1.2}), (s_{A1.2}, s_{A1.3}), (s_{A1.3}, s_3), (s_3, s_4)\}\}$ for branch coverage. The diagram is smaller for step coverage because all the steps are included in branch 1.

From the above algorithm, test scenarios are selected and output. **Table 1** summarizes an example of the generated test scenario.

- In the row of test items (definition of variables used as input and output items for FIT), steps are described in the order of execution. For example, as shown in Table 1, steps such as “The system displays available facilities,” “The customer chooses to make a reservation for a room,” … “The system terminates the use case” are described in the row of test items (the second row in Table 1).

- When class names included in the domain model are described in the steps, the class names designated by the user of the proposed system are described in the column prior to that of the step in the row of test items. For example, when “customer,” “room,” and “reservation” are included among the class names of the domain model, these three items are the candidate test items for the step “The customer chooses to make a reservation for a room.” If the user of the proposed system selects only “room,” then “room” and “The customer chooses to make a reservation for a room” are added to the row of test items.

- The number of output test scenarios changes with the type of coverage selected. For example, if step coverage is selected, the minimum number of test scenarios that are required to test each step at least once is output. Namely, the following relationship holds for the number of test scenarios generated:

(Number of test scenarios when step coverage is selected) ≤ (Number of test scenarios when branch coverage is selected) ≤ (Number of test scenarios when all-execution-flows coverage is selected).

- In the third or later rows, “pass” and “null” are described when the columns on the second row are the steps and elements of the domain model, respectively. This is because the output of the proposed system simply consists of skeleton test scenarios. For example, “pass” is written in the row immediately below the row “The system displays available facilities,” and “null” is written in the row immediately below the row “room.” In the detailing phase, “pass” and “null” should be modified.

Table 1 shows a part of the test scenario output when all-execution-flows coverage is selected. **Table 2** shows an example of the test scenario obtained by manually detailing the test scenario in Tables 1. The domain model used as the input is shown in Fig. 7.

Table 1. Generated test scenario 1 corresponding to use case “Reserve room” before detailing

ReserveRoom				
systemDisplaysAvailable Facilities()	Room	`customerChooses ToMakeAReservation ForARoom()`	…	`systemTerminates TheUseCase()`
Pass	null	pass	…	pass
Pass	null	pass	…	pass

Table 2. Example of detailed test scenario 1 corresponding to use case "Reserve room"

ReserveRoom				
systemDisplaysAvailable Facilities()	Room	`customerChooses ToMakeAReservation ForARoom()`	…	`systemTerminates TheUseCase()`
List	101	Reserve	…	exit
List	201	Reserve	…	exit

4.3 Generation of fixtures

In this section, the generation of fixtures in the input phase and the detailing of fixtures carried out in the detailing phase are explained.

Figure 16 shows an example of a fixture generated by the proposed system. For use case A,

- only a fixture with the name AFixture, or
- a fixture with the name AFixture and the fixture ConcreteAFixture, which inherits AFixture,

is generated through the selection of the user. ConcreteAFixture is implemented as an example of test codes and the developer carries out the test using ConcreteAFixture. The reason for allowing the user to select one of the two above options is to prevent the overwriting of the fixture details that are added by the developer when a use case is changed and the fixture is regenerated. For a step defined in use case A, AFixture, a parent class corresponding to use case A, owns the operation with the same name as the step. Similarly, if the use case description of use case A includes class names of the domain model, AFixture owns these classes as attributes. **Figure 17** shows a code fragment of ReserveRoomFixture shown in Fig. 16.

In the use case "Reserve room," there are steps such as "The system displays available facilities," "The customer chooses to make a reservation for a room," … "The system terminates the use case," and "If the customer indicates that the new reservation is a duplicate one, the system terminates the use case." Therefore, ReserveRoomFixture has the operations with the same names listed above. Similarly, because there are "customer," "room," "reservation," and "facility" in the domain model, AFixture has them as attributes. Developers have to manually code detail of the fixture output from the proposed system.

The column "The system displays available facilities" in Table 2 corresponds to the method "systemDisplaysAvailableFacilities" in **Fig. 16**; similarly, the column "room" in Table 2 corresponds to the variable "room" in Fig. 16. The third row in Table 2 indicates that the system tests the following: (1) whether we obtain "List" when the method "systemDisplaysAvailableFacilities" is executed; (2) whether we obtain "Reserve" when the method "customerChoosesToMakeAReservationForARoom" is executed by substituting 101 for "room"; and (3) whether we obtain "exit" when the method "systemTerminatesTheUseCase" is executed.

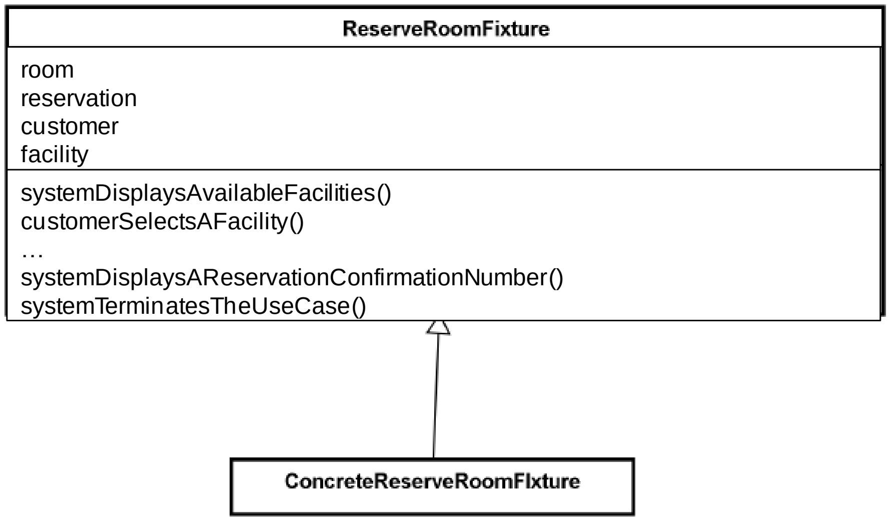

ReserveRoomFixture
room
reservation
customer
facility
systemDisplaysAvailableFacilities()
customerSelectsAFacility()
…
systemDisplaysAReservationConfirmationNumber()
systemTerminatesTheUseCase()
ConcreteReserveRoomFIxture

Figure 16. Example of generated fixture before detailing

```
public class ReserveRoomFixture extends ColumnFixture {

    public Room room;

    …

    public String systemDisplaysAvailableFacilities() {

        return “pass”;

    }

    …
```

Figure 17. Part of fixture corresponding to use case “Reserve room” before detailing

4.4 Generation of fixtures on the basis of relationship between use cases

The relationship between fixtures corresponding to the use cases is defined by an inter-type declaration (a mechanism describing the method fields that should belong to another class[13)]) based on aspect-oriented programming (a technique for modularizing cross-sectional interest[12)]). Owing to the definition of the relationship between use cases using aspects, we only need to revise the aspects, instead of the fixtures, when the relationship between use cases changes, leading to an increase in the reusability of the fixtures. In this study, aspects are generated using AspectJ.

Include: If there is an include relationship between two use cases, an aspect is generated so that the fixture of the including use case has operations with the same name as those of the fixture of the included use case by referring to a fixture corresponding to the included use case.

Generalization: If there is a generalization relationship between two use cases, an aspect is generated so that the corresponding fixtures have an inheritance relationship.

Extend: If there is an extend relationship between two use cases, an aspect is generated so that the fixture of the extended use case has operations with the same name as those of the fixture of the extending use case by referring to a fixture corresponding to the extending use case.

Figure 18 shows the fixtures generated for the use cases shown in Fig. 1 using the proposed system on the basis of the above rules.

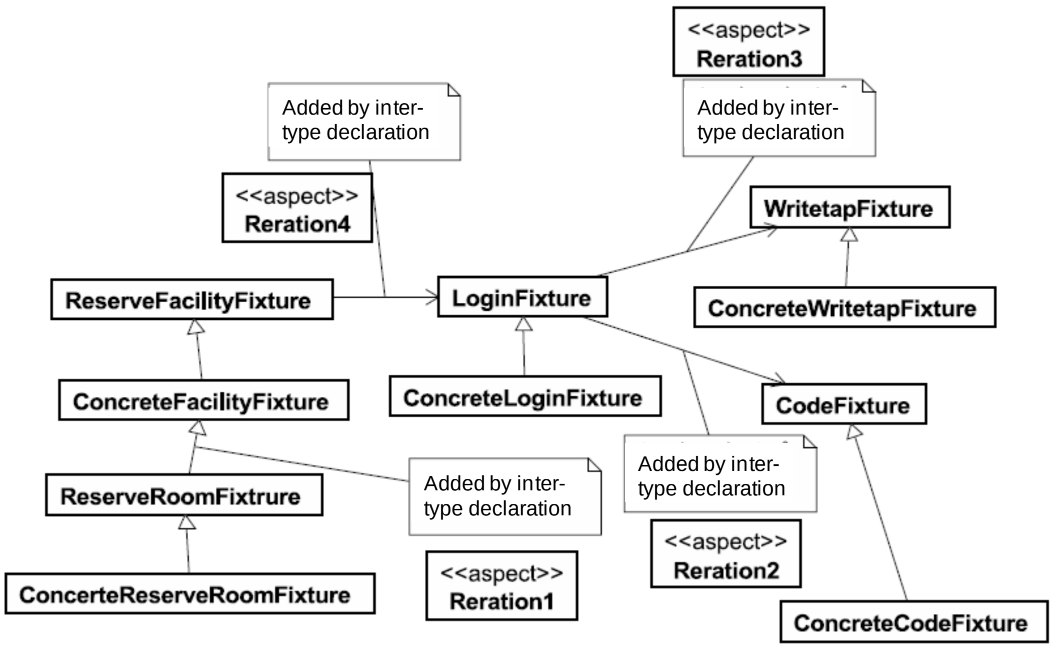


Figure 18. Class diagram of generated fixtures

4.5 Limitations of the proposed technique and system

The test scenarios prepared by the proposed technique alone are insufficient as the scenarios to be used in the acceptance test. The proposed technique focuses on the coverage of steps. Nonfunctional requirements, such as usability, are not described in the flow and such factors should be tested by other techniques.

Because the proposed technique requires the manual detailing of the output test scenarios and test fixtures, errors may occur during these processes. In the conventional technique, identification of the execution flows and implementation of the test scenarios are also carried out manually, whereas in the proposed system, identification of the execution flows is automated and only the implementation of the test scenarios is carried out manually. Namely, the proposed technique involves fewer manual processes, leading to a decreased likelihood of errors. However, the proposed technique may not be able to detect errors that depend on the data.

The proposed technique automatically generates the test scenarios of a functional test on the basis of specific coverage criteria, leading to an increase in the reliability of the functional test.

The limitations of the proposed system in terms of implementation are as follows. These problems do not depend on the framework of the proposed technique.

(A) A special editor is used for use case description.

(B) The generated fixtures are in Java format.

(C) Because the domain model is input using the Jude application programming interface (API),[11)] which is a modeling tool, the domain model is assumed to be written in Jude. However, the framework of the proposed system does not depend on a particular modeling tool. In practice, the use of the Jude API is not a major problem because Jude Professional is compatible with the import and export of the extensive markup language metadata interchange (XMI) format.

5. Experiment

A comparative experiment was carried out using the conventional[9)] and proposed techniques. The subjects were six students with approximately two years of experience with UML. The subjects were provided with the use case diagram (**Fig. 19**) in ref. 14) and the use case descriptions, and instructed to identify all the execution flows in the use case "Manage a history of clothes sales" and list the test scenarios using branch coverage. The number of use cases was ten, the number of use case descriptions was ten, the number of relationships between use cases was ten (nine include relationships and one extend relationship), the total number of steps in the ten use cases was 47, the number of branches was 12, and the total number of flows was 32. In the experiment, the numbers of errors and the incomplete coverage of the execution flows were compared between the two techniques. The number of use cases for which the proposed technique and system are effective is arbitrary; however, we consider that the proposed technique is more effective for large systems with a large number of use cases and a complicated relationship between them. The complexity of the relationship between the use cases adopted in the experiment is considered to be sufficient for comparing the effectiveness between the conventional[9)] and proposed techniques. The purposes of the experiment are as follows.

- To demonstrate that incomplete coverage and errors may occur during the manual identification of execution flows when there is a complicated relationship between use cases.

- To determine that incomplete coverage and error can be eliminated by the automatic identification of execution flows of use cases by the proposed system.

Subjects A, B, and C first identified and listed the execution flows using the conventional technique first. Then, they inputted the use case descriptions into the proposed system, regardless of the output from the conventional system, and obtained the automatically generated test scenarios. In contrast, subjects D, E, and F first inputted the use case descriptions into the proposed system and obtained the automatically generated test scenarios. After that, they identified the execution flows in the use case descriptions by the conventional technique and listed them, regardless of the output from the proposed system. **Table 3** shows the number of errors in the execution flows listed by the subjects.

When the conventional technique was used first, the number of errors was two for subject A and one for subjects B, C, E, and F. The number of errors for subject D was zero both when the conventional technique was used first and when the proposed technique was used first. Subject A forgot to consider the change as a result of the extend relationship in the use case "Schedule dates of repeated consultation with customer" and transferred to an alternative flow at an incorrect step in the use case "Design clothes." Similarly, subjects B, E, and F transferred to an alternative flow at an incorrect step in the use case "Design clothes." For subject C, the extend step in the use case "Schedule dates of repeated consultation with customer" was in correct.

These errors may decrease the branch coverage and all-execution-flows coverage. In the proposed technique, execution flows are automatically identified; thus, errors and incomplete coverage can be prevented. We experimentally confirmed that incomplete coverage and errors may occur during the manual identification of execution flows. The effects of such errors were not significant in this experiment owing to the small number of use cases; however, the effects may be critical when the number of use cases is large, such as several tens or several hundred. From the results of this experiment, the proposed technique is considered to be effective because incomplete coverage and errors can be prevented and the coverage of the acceptance test is improved. **Figure 20** shows the time required to identify the execution flows by the conventional technique. The minimum and maximum, mean, and median times were 30, 72, 46.3, and 42 min, respectively. These times can be reduced using the proposed technique.

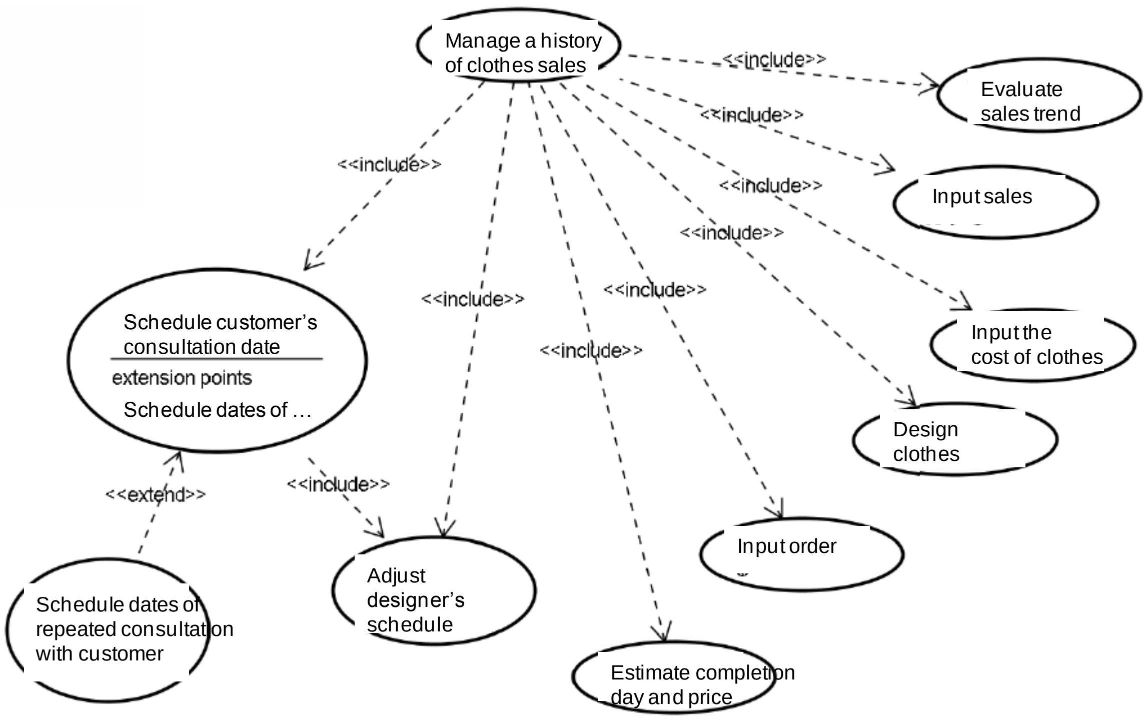

Manage a history of clothes sales
<<include>>
Evaluate sales trend
<<include>>
Input sales
<<include>>
<<include>>
Input the cost of clothes
<<include>>
Design clothes
<<include>>
<<include>>
Input order
<<include>>
Estimate completion day and price
Schedule customer's consultation date
extension points
Schedule dates of ...
<<extend>>
<<include>>
Schedule dates of repeated consultation with customer
Adjust designer's schedule

Figure 19. Use case diagram used in the experiment

Table 3. Number of errors in execution flows listed by subjects (CON: Conventional technique, PRO: Proposed technique)

Subject	A	B	C	D	E	F
Technique used first	CON	CON	CON	PRO	PRO	PRO
Conventional technique	2	1	1	0	1	1
Proposed technique	0	0	0	0	0	0

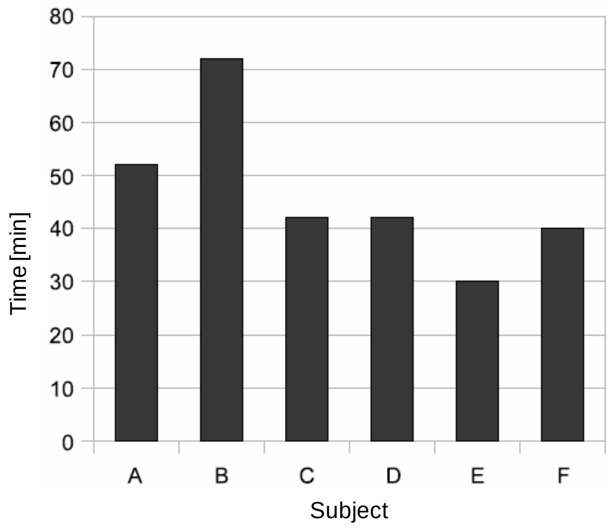


Figure 20. Time required for identification by the conventional technique

6. Related work

Zielczynski discussed the traceability from use cases to test cases.9) In Zielczynski's technique, all the test scenarios are manually identified using the use cases and appropriate test scenarios are selected. Incomplete coverage and errors may occur during the manual identification. Also, the relationship between use cases is not taken into consideration. Using the proposed technique, incomplete coverage can be prevented because execution flows are automatically identified. Thus, comprehensive test scenarios considering the relationship between use cases can be generated.

Stephane *et al*. described the generation of test scenarios at the system level on the basis of text-format use cases.15) In their technique, the test scenarios used to check the functions of the entire system are generated from the use cases to realize the early verification of the use cases. However, the purpose of their technique is to verify the use cases, and executable test scenarios are not generated by their technique. In contrast, our proposed technique generates both test scenarios and test programs, although they are skeletons. The relationships between use cases considered in the technique of Stephane *et al*. and the proposed technique are different. In their technique, include relationships, preconditions, and postconditions are considered; whereas, in the proposed technique, the include, generalization, and extend relationships are considered when generating the test scenarios. In addition, the definition of the diagram of use cases differs between the technique of Stephane *et al*. and the proposed technique. Steps are defined as nodes in the proposed system. In Stephane *et al*.'s technique, steps are defined as edges, and the states of the system before and after a step are defined as nodes. Because the definition of the diagram of use cases differs between the two techniques, the definition of the coverage also differs. In Stephane *et al*.'s technique, state coverage, step coverage, and all-execution-flows coverage are defined, whereas in the proposed technique, step coverage, branch coverage, and all-execution-flows coverage are defined. From the viewpoint of testing, the proposed technique, which defines branch coverage, is superior because bugs are likely to be detected at branches.4) Owing to the definition of branch coverage, errors in the description of branch conditions and bugs due to transition to unexpected steps or flows can be detected.

Amyot *et al*. proposed the use-case-map (UCM;17) a technique for modeling scenarios)-driven testing of Web applications.16) Their technique identifies execution flows, which are targets of the acceptance test, using UCM to improve the accuracy of the identification of test scenarios. However, the relationship between different use cases is not taken into consideration, leading to difficulty in generating comprehensive test scenarios. In contrast, the proposed technique can generate comprehensive test scenarios considering the relationship between use cases.

Nebut *et al*. explained the use-case-driven automatic generation of test scenarios.[18)] A formal limitation is applied to the preconditions and postconditions in the use cases to automatically generate test scenarios. Although test scenarios can be generated automatically using a formal description, the cost associated with the formal description is high. In addition, comprehensive test scenarios using the preconditions and postconditions are generated by Nebut *et al*.'s technique. On the other hand, the proposed technique adopts an informal description and considers the relationship between different use cases, but does not consider the preconditions and postconditions.

7. Conclusion and future work

A widely accepted definition for the coverage of the acceptance test using test scenarios identified from use cases has not yet been established. Thus, the judgment of acceptance test completion largely depends on individuals, leading to a decrease in the reliability of the acceptance test. The reliability also decreases when the relationship between use cases is complicated because incomplete coverage and errors may occur during the manual identification of execution flows. In this study, we defined three coverages for the acceptance test based on the test scenarios identified from use cases. We also proposed a technique of automatically generating skeleton test scenarios and skeleton test programs using FIT that satisfy the specified coverage using use case descriptions and a domain model as the input. An experiment using a small number of use cases was carried out using the proposed and conventional techniques. We confirmed that the incomplete coverage can be prevented using the proposed technique. Because the proposed technique automatically identifies the execution flows using the definition of the coverage in the acceptance test, acceptance test completion can be determined without depending on individual judgments. In addition, the proposed technique can automatically generate test scenarios that satisfy the coverage required in the test, leading to an increase in the efficiency of the acceptance test.

In the future, experiments using more samples, compatibility with languages other than Java, the generation of test cases in addition to execution flows, and XMI input to improve the compatibility with multiple modeling tools, should be considered to improve the performance of the proposed technique.

References

1) Doug Rosenberg, “Guide for Use-Case-Driven Development,” Shoeisha, 2007.

2) Cem Kaner, Hung Quoc Nguyen, Jack Falk, “Basics for Software Test – Becoming a Professional in Testing,” Nikkei Business Publications, Inc., 2001.

3) Alister Cockburn, “Writing Effective Use Cases,” Shoeisha, 2001.

4) Boris Beizer, “Software Testing Techniques,” Nikkei Business Publications, Inc., 1994.

5) Object Management Group, Unified Modeling Language 2.0, http://www.uml.org/.

6) Sindre Guttorm, ”Eliciting security requirements with misuse cases. ” Requirements Engineering, Vol. 10, No. 1, pp. 34 - 44, Jan. 2005.

7) Martin Fowler, “UML Distilled, 3rd Edition,” Shoeisha, 2005.

8) Rick Mugridge, Ward Cunningham, ”Fit for Developing Software: Framework for Integrated Tests,” Prentice Hall, 2005.

9) Peter Zielczynski , ”Traceability from Use Cases to Test Cases,” developerWorks, http://www.ibm.com/developerworks/rational/library/04/r-3217/3217 rm14.pdf, 2006.

10) Jim Heumann, ”Generating Test Cases From Use Cases,” Rational edge, 2001, http://www.ibm.com/developerworks/rational/library/content/RationalEdge/jun01/GeneratingTestCasesFromUseCasesJune01.pdf

11) JUDE Professional, http://jude.change-vision.com/jude-web/index.html

12) Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jffrey Palm, and William G. Griswold, ”An Overview of AspectJ ”, Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001), LNCS, Vol.2071, pp. 327-354, 2001

13) Yoshihide Nagase, Masahiro Amano, Hironori Washizaki, Michiaki Tatsubori, “Introduction to Aspect-Oriented Programming by AspectJ,” Softbank Publishing, 2004 (in Japanese).

14) Daryl Kulak, Eamonn Guiney, “Use Cases: Requirement in Context,” Pearson Education, 2002.

15) Stephane S Some , Xu Cheng, ”An approach for supporting system-level test scenarios generation from textual use cases”, Symposium on Applied Computing: Proceedings of the 2008 ACM symposium on Applied computing, pp.724 - 729, 2008.

16) Daniel Amyot, Jean-Francois, Roy Michael Weiss, ”UCM-Driven Testing of Web Applications”, Proceeding of 12th SDL Forum (SDL 2005), LNCS 3530, Springer, pp.247-264, 2005.

17) Ray J.A. Buhr, Ron S. Casselman, “Use Case Maps for Object-Oriented Systems,” Toppan Printing Co., Ltd., 1998.

18) Cle’mentine Nebut, Franck Fl eurey, Yves Le Traon, ”Automatic Test Generation: A Use Case Driven Approach,” IEEE Transactions on Software Engineering, Vol.32, No.3, pp.140-155, 2006.