A Framework for Automatic Functional Testing Based on Formal Specifications

Shaoying Liu* and Shin Nakajima+

***Department of Computer Science**

Hosei University, Japan

+NII, **Japan**

Emails: sliu@hosei.ac.jp, nkjm@nii.ac.jp

Overview

- Motivation
- Framework for automatic specification-based testing
- Conclusion and future research directions

1. Motivation

(1) Understanding necessary activities and potential technical challenges for automatic specification-based testing.

(2) Establishing a foundation for building automatic specification-based testing tools.

(3) Exploring new and/or better techniques for automatic-based testing (e.g., functional scenario-based testing and integration of the Hoare logic and test case generation for debugging)

Goals of automatic testing

1. Every function defined in the specification is tested (at least once) (User's view).

2. Every representative program path is traversed or some required coverage criteria are satisfied. (Program's view)

3. By testing every function defined in the specification, all of the bugs in the program are detected. (ideal goal for both the user and the programmer)

2. Framework for automatic specification-based testing

The framework shows 10 major activities necessary for automatic specification-based testing:

(1) Deriving test conditions from the specification
(2) Generating test cases based on test conditions
(3) Translating abstract test cases to concrete ones
(4) Generating test environment
(5) Running the program with test cases and managing test result data
(6) Translating test result from concrete form to abstract form
(7) Analyzing test results for bug discovery
(8) Debugging
(9) Eliminating bugs
(10) Managing and reusing test data and related documents

(1) Deriving test conditions from the specification

The objective of test case generation is to exercise every functional scenario defined in the specification. Each functional scenario describes an independent function in terms of input-output relation, and is expressed as a conjunction of a test condition and a defining condition. A specification is equivalent to a functional scenario form, which is a conjunction of all functional scenarios defined in the specification.

Let S(Siv,Sov)[Spre,Spost] denote an operation specification.

Definition 1. (FSF)

Let $S_{post} \equiv C_1 \wedge D_1 \vee C_2 \wedge D_2 \vee \cdots \vee C_n \wedge D_n$,

C_i: guard condition

D_i: defining condition. $i = 1,\ldots,n$.

Then, a functional scenario form (FSF) of S is:

$(\sim S_{pre} \wedge C_1 \wedge D_1) \vee (\sim S_{pre} \wedge C_2 \wedge D_2) \vee \cdots \vee (\sim S_{pre} \wedge C_n \wedge D_n)$

where

$f_i = \sim S_{pre} \wedge C_i \wedge D_i$ is called a functional scenario) and

$\sim S_{pre} \wedge C_i$ is called a test condition

Scenario-based testing: a strategy for ``divide and conquer''

Specification

```
process A(x: int) y: int
pre   x > 0
post x > 10 and y = x + 1 or
       x <= 10 and y = x – 1
end_process
```

Program

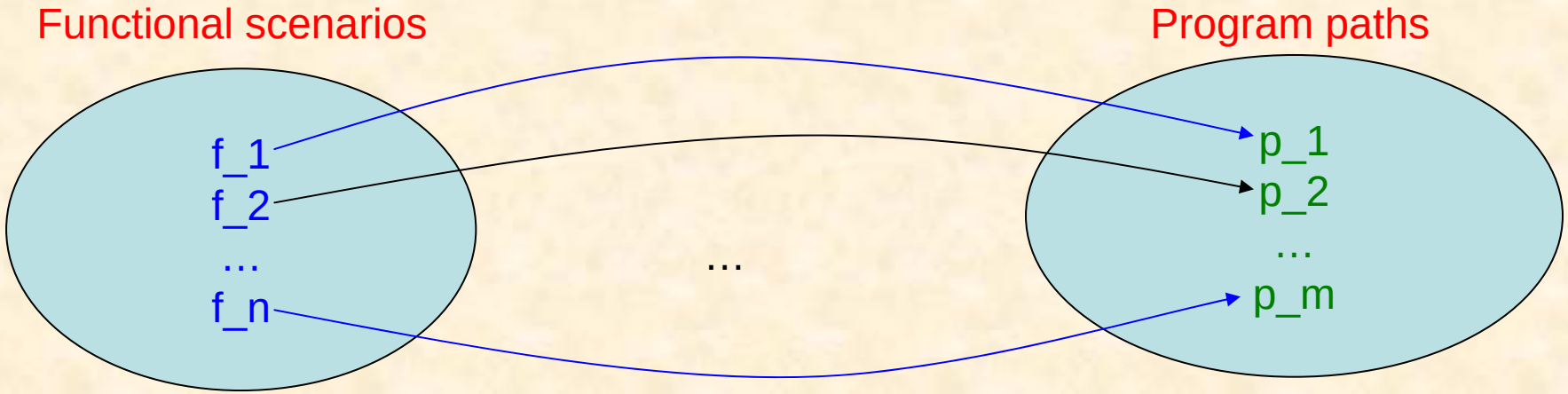
```
int A(int x) {
If (x > 0) {
 if (x > 10)  y = x + 1;
 else  y = x – 1;
 return y; }
else System.out.println("the
       pre is violated")  }
```

Satisfy?

Functional scenario:

~Apre ∧Ci ∧Di

(i=1,…,n)



(2) Generating test cases based on test conditions

Let G denote a test set generator (TSG), which is a function from the universal set of logical expressions L_E to the universal set of test sets T_S, formally,

$$G: L_E \rightarrow T_S$$

A decompostional approach to test case generation (only two rules are shown here):

(1) $G((\sim S_{pre} \wedge C_1 \wedge D_1) \vee (\sim S_{pre} \wedge C_2 \wedge D_2) \vee \cdots \vee (\sim S_{pre} \wedge C_n \wedge D_n)) =$

$G(\sim S_{pre} \wedge C_1 \wedge D_1) \cup G(\sim S_{pre} \wedge C_2 \wedge D_2) \cup \cdots \cup G(\sim S_{pre} \wedge C_n \wedge D_n)$

A test set generated for a specification is equal to the union of all test sets generated for all the functional scenarios of the specification that take both the pre- and postconditions into account.

(2) $G(\sim S_{pre} \wedge C_1 \wedge D_1$

$G(\sim S_{pre} \wedge C_1)$

Test set generation from a functional scenario is made by generating the test set from its test condition.

(3) Translating abstract test cases to concrete ones

Test cases generated from a specification usually conforms to the specification language syntax and therefore may not be directly used by the program under test. A translation from the abstract test cases to concrete ones that can be used by the program is necessary.

Considering the fact that there are many possibilities of using concrete data structures to implement abstract data types in programs, automation of such a translation can be a challenge due to complex data structures.

(4) Generating test environment

To run the program with the generated test cases, a test environment, called driving module, must be automatically created. The driving module is intended to insert probs in appropriate places of the subprogram under test and call the subprogram with the test cases.

(5) Running the program with test cases and managing test result data

The driving module is activated so that the subprogram under test is executed with the supplied test cases. The test result must be collected for test result analysis in the next step. The challenge lies in the collection of all the produced or updated “output’’ values due to the scale or the complexity of the data structures.

(6) Translating test result from concrete form to abstract form

The actual test result, which is a set of ``output'' values, must be translated into expressions in the abstract form that conforms to the specification language syntax for test result analysis in the next step. The reason is that the "evaluator'' for the formal specification language can only deal with expressions in the abstract form.

(7) Analyzing test results for bug discovery

In this step, a test oracle is automatically derived from the specification, which is condition given below, and the test oracle is evaluated to determine whether any bug exists in the program.

Definition 3.1: Let **T** be a test set. If the condition

$$\exists\, t \in T \cdot S_{pre}(t) \wedge \neg\, S_{post}(t, P(t))$$

holds, it indicates the existence of a bug in program **P**.

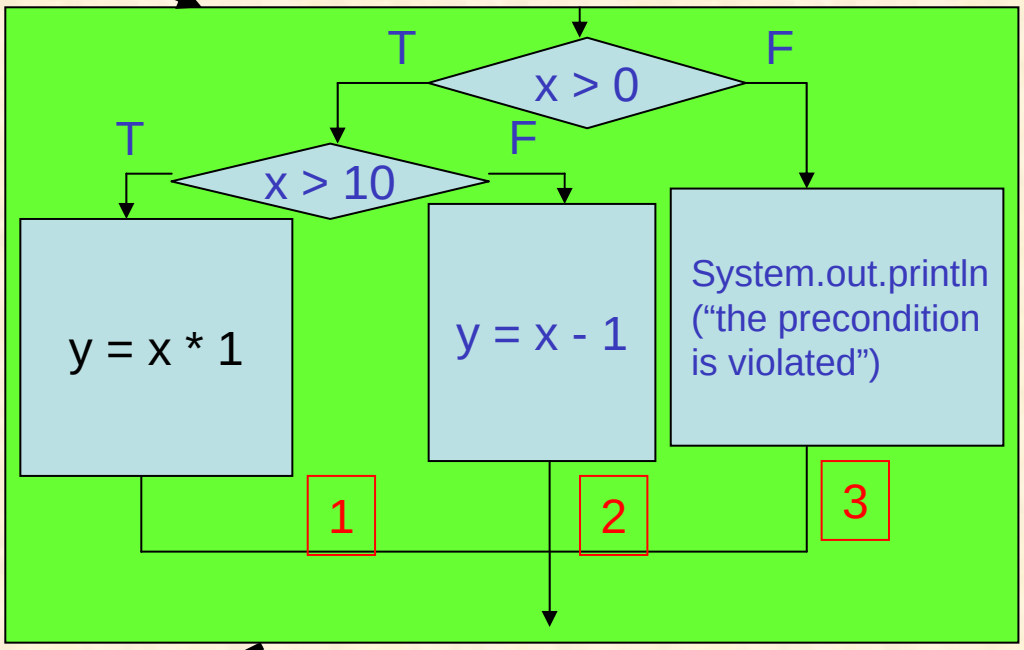
Test case generation

Specification

A(x: int) y: int
pre x > 0
post x > 10 ∧ y = x + 1 ∨
x <= 10 ∧ y = x – 1

Functional scenarios:
(1) x > 0 ∧ x > 10 ∧ y = x + 1
(2) x > 0 ∧ x <= 10 ∧ y = x – 1
(3) x <= 0 (optional)

Program



Test result analysis

x	y	Apre	Apost	Apre ∧ ¬ Apost
15	15	true	false	true
5	4	true	true	false

(8) Debugging

Debugging is often perceived as an activity independent of testing, but if the objective of automatic testing is to detect all bugs, automatic debugging should be considered as part of the testing process.

Automatic debugging has long been a challenge and there is still no effective technique for handling this problem, but combining the Hoare logic and automatic test case generation may lead to a possible solution.

(9) Eliminating bugs

Automatic elimination of bugs is not usually counted as part of the automatic testing process, but if we wish to achieve the effect of removing all bugs by only pressing one button, elimination of bugs must be taken into account. This task is another great challenge, and perhaps no satisfactory solution can be worked out in the near future.

(10) Managing and reusing test data and related documents

To support all the test activities, test data and the related information must be stored and managed properly.

The information can be classified into three categories: (1)documents, (2) test data, and (3) bug data.

The documents include the specification from which test cases are generated and the program under testing. The test data include test cases in the abstract form, test cases in the concrete form, test results in the concrete form, test results in the abstract form, and traversed program paths. The bug data include the bug descriptions, the program statements or conditions containing bugs revealed, and the program paths containing bugs.

3. Conclusion and future research directions

3.1 Conclusion

- We describe a framework that covers all of the major activities necessary for automatic specification-based testing.
- The framework establishes a foundation for building automatic specification-based testing tools.
- The framework presents some preliminary ideas for a new approach to some test activities such as test case generation and debugging.

3.2 Future research directions

- A theoretical foundation for automatic specification-based testing.
- More effective and efficient test case generation methods.
- Effective techniques for automatic debugging.
- Development of an automatic specification-based testing environment.

Thank you !