Integrating Specification Animation with Specification-Based Program Testing and Inspection for Software Quality Assurance

Shaoying Liu
Department of Computer Science
Faculty of Computer and Information Sciences
Hosei University, Japan
Email: sliu@hosei.ac.jp
HP: http://cis.k.hosei.ac.jp/~sliu/

This work is supported by JSPS KAKENHI Grant Number 26240008.
Overview

1. Challenges to Software Quality Assurance
2. Our Solution
3. Specification Animation
4. Specification-Based Program Testing and Inspection
5. Open Problems
6. Conclusions
7. Future Work
1. Challenges to Software Quality Assurance

- The scale and complexity of software development projects
  - The scale of documentation
  - The complexity of documentation
  - The complexity of situations (e.g., requirements changing, people moving, client complaining, manager worrying, and developer fighting)
The constrained development environment

- Budget
- Schedule
- Requirements for reliability
- Developer’s skills and experience
- Communications
Deficiencies of techniques available for use

- **Formal proof of correctness**: ideal but tedious, ineffective (for faulty programs), requiring skills (loop invariants), error-prone, and time consuming.

- **Model checking**: needs appropriate abstraction of a real system to a FSM model and faces the state explosion problem (two state space explosions for software: initial state space and program state space).

- **Testing**: can tell the existence of bugs, but cannot tell their absence in general. Nevertheless, it is a common practice in industry.

- **Review and inspection**: easy to carry out, but heavily depend on human judgment, ability, and experience.
Harsh reality

Manager:
Why is the project over budget and behind schedule?

Client:
Why does the software system behave differently from my requirements?

Developer:
Why are there so many bugs remaining in the program?
Why is my own program difficult to understand even by myself?
2. Our Solution

Specification

Animation

Software
defects

Preparation

Preparation

Specification-Based
Testing

Specification-Based
Inspection

Mutual aid
3. Specification Animation

Specification animation is a technique for dynamic and visualized demonstration of the system behaviors defined in the specification.

Three expected effects: improving understanding of requirements or designs, strengthening communication, and verifying/validating specifications.
SOFL: Structured Object-oriented Formal Language

The structure of a SOFL specification:

CDFDs + modules + classes

class S1;
const; type; var; inv;
method Init;
method P1;
method P2;
method P3;
end-class;

module SYSTEM;
const; type; var; inv;
process Init;
process A1;
process A2;
end-module;

class S2;
const; type; var; inv;
method Init;
method Q1;
method Q2;
method Q3;
end-class;

module A2-decom;
const; type; var; inv;
process Init;
process B1;
process B2;
process B3;
end-module;
A simplified ATM specification in SOFL:

- **Receive_Command**
  - balance
  - w_draw

- **Check_Password**
  - card_id
  - pass
  - sel
  - account1
  - account2

- **Withdraw**
  - amount
  - e_msg
  - cash

- **Show_Balance**
  - balance
module SYSTEM_ATM;

type
    Account = composed of
        account_no: nat
        password: nat
        balance: real
    end

var
    account_file: set of Account;

inv
    forall[x: account_file] | x.balance >= 0;

behav CDFD_No1;
...

process Withdraw(amount: real, account1: Account)
    e_msg: string | cash: real
ext wr account_file: set of Account
pre account1 inset account_file
post if amount <= account1.balance
    then
        cash = amount and
        let Newacc =
            modify(account1, balance -> account1.balance – amount)
        in
            account_file = union(diff(~account_file, {account1}), {Newacc})
    else
        e_meg = "The amount is over the limit. Reenter your amount."
end_process;
Basic idea of SOFL specification animation for verification and validation

{withdraw_comm}[Receive_Command, Check_Password, Withdraw]{cash}
{withdraw_comm}[Receive_Command, Check_Password, Withdraw]{err2}
{withdraw_comm}[Receive_Command, Check_Password]{err1}
{withdraw_comm}[Receive_Command, Check_Password, Show_Balance]{balance}
{balance_comm}[Receive_Command, Check_Password, Withdraw]{cash}
{balance_comm}[Receive_Command, Check_Password, Withdraw]{err2}
{balance_comm}[Receive_Command, Check_Password]{err1}
{balance_comm}[Receive_Command, Check_Password, Show_Balance]{balance}
Testing-Based Animation Approach

Steps of Animation:

Step 1: Deriving system functional scenarios

Step 2: Generating test cases

Step 3: Carrying out animation for each scenario using the test cases.
Animation of a single scenario

\{withdraw\_comm\}[Receive\_Command11, Check\_Password11, Withdraw11]\{cash\}

<table>
<thead>
<tr>
<th>Process</th>
<th>Input Variables</th>
<th>Input Data</th>
<th>Output Variables</th>
<th>Output Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Received_Command11</td>
<td>{withdraw_comm}</td>
<td>{“withdraw”}</td>
<td>{sel}</td>
<td>{true}</td>
</tr>
<tr>
<td>Check_Password11</td>
<td>{sel, id, pass, ~Account_file}</td>
<td>{true, 0001, 1111, (0001, “Jack”, 1111, 15000)}</td>
<td>{acc1}</td>
<td>{(0001, “Jack”, 1111, 15000)}</td>
</tr>
<tr>
<td>Withdraw11</td>
<td>{acc1, amount}</td>
<td>{(0001, “Jack”, 1111, 15000), 5000}</td>
<td>{cash, Account_file}</td>
<td>{5000, (0001, “Jack”, 1111, 10000)}</td>
</tr>
</tbody>
</table>
Animation of a single scenario

{withdraw_comm}\[Receive_Command11, Check_Password11, Withdraw11\}{cash}

<table>
<thead>
<tr>
<th>Process</th>
<th>Input Variables</th>
<th>Input Data</th>
<th>Output Variables</th>
<th>Output Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Received_Command11</td>
<td>{withdraw_comm}</td>
<td>{“withdraw”}</td>
<td>{sel}</td>
<td>{true}</td>
</tr>
<tr>
<td>Check_Password11</td>
<td>{sel, id, pass, account_file}</td>
<td>{true, 0001, 1111, (0001, “Jack”, 1111, 15000)}</td>
<td>{acc1}</td>
<td>{(0001, “Jack”, 1111, 15000)}</td>
</tr>
<tr>
<td>Withdraw11</td>
<td>{acc1, amount}</td>
<td>{(0001, “Jack”, 1111, 15000), 5000}</td>
<td>{cash, Account_file}</td>
<td>{5000, (0001, “Jack”, 1111, 10000)}</td>
</tr>
</tbody>
</table>
Animation of a single scenario

\{\text{withdraw\_comm}\}[\text{Receive\_Command11, Check\_Password11, Withdraw11}\}{\text{cash}}

<table>
<thead>
<tr>
<th>Process</th>
<th>Input Variables</th>
<th>Input Data</th>
<th>Output Variables</th>
<th>Output Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Received_Command11</td>
<td>{withdraw_comm}</td>
<td>{“withdraw”}</td>
<td>{sel}</td>
<td>{true}</td>
</tr>
<tr>
<td>Check_Password11</td>
<td>{sel, id, pass, \sim Account_file}</td>
<td>{true, 0001, 1111, (0001, “Jack”, 1111, 15000)}</td>
<td>{acc1}</td>
<td>{(0001, “Jack”, 1111, 15000)}</td>
</tr>
<tr>
<td>Withdraw11</td>
<td>{acc1, amount}</td>
<td>{(0001, “Jack”, 1111, 15000), 5000}</td>
<td>{cash, Account_file}</td>
<td>{5000, (0001, “Jack”, 1111, 10000)}</td>
</tr>
</tbody>
</table>
Animation of a single scenario

{withdraw_comm}[Receive_Command11, Check_Password11, Withdraw11][cash]
Test case generation for processes (operations)

A test case is composed of a test datum and the corresponding expected result.

\[ S(S_{iv}, S_{ov})[S_{pre}, S_{post}] \]
A specific method for test case generation

Functional Scenario-Based Test Case Generation:

a strategy for “divide and conquer”
Overall idea:

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

Functional scenario:
A_{pre} \land G_{i} \land D_{i}
(i=1,\ldots,n)

A set of functional scenarios

Derivation
Definition (FSF): Let

\[ S_{\text{post}} \equiv (G_1 \land D_1) \lor (G_2 \land D_2) \lor \cdots \lor (G_n \land D_n), \]

where \( G_i \) is a **guard condition** and \( D_i \) is a **defining condition**, \( i = 1, \ldots, n \).

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

\[ (S_{\text{pre}} \land G_1 \land D_1) \lor (S_{\text{pre}} \land G_2 \land D_2) \lor \cdots \lor (S_{\text{pre}} \land G_n \land D_n) \]

where \( f_i = S_{\text{pre}} \land G_i \land D_i \) is called a **functional scenario** (for generating test cases).
Test case generation criterion:

Let operation $S$ have an FSF:

$$(S_{\text{pre}} \land G_1 \land D_1) \lor (S_{\text{pre}} \land G_2 \land D_2) \lor \cdots \lor (S_{\text{pre}} \land G_n \land D_n), \text{ where } (n \geq 1).$$

Let $T$ be a test set for $S$. Then, $T$ must satisfy the condition

$$(\forall i \in \{1, \ldots, n\} \exists t \in T \cdot S_{\text{pre}}(t) \land G_i(t) \land D_i(t))) \text{ and } \exists t \in T \cdot \neg S_{\text{pre}}(t)$$

where $\neg S_{\text{pre}}(t)$ describes an exceptional situation.
Example

A process specification in SOFL:

```plaintext
process ChildTicketDiscount(a: int, np: int) ap: int
pre a > 0 and np > 1
post (a > 12 => ap = np) and (a <= 12 => ap = np - np * 0.5)
end_process

where a = age, ap = actual price, np = normal price
```
Two functional scenarios and one exception can be derived from this formal specification:

(1) \(a > 0 \text{ and } np > 1 \text{ and } a > 12 \text{ and } ap = np\)

(2) \(a > 0 \text{ and } np > 1 \text{ and } a \leq 12 \text{ and } ap = np - np \times 0.5\)

(3) \(a \leq 0 \text{ or } np \leq 1 \text{ and anything}\)

where anything means that anything can happen when the pre-condition is violated.
Test case generation

Test cases satisfying functional scenarios:

\[
t1 = \{(a, 15), (np, 100), (ap, 100)\}
\]
\[
t2 = \{(a, 10), (np, 100), (ap, 50)\}
\]

Test case violating the pre-condition (exceptional test case):

\[
t3 = \{(a, 0), (np, 200), (ap, 100)\}
\]
Test case generation within a system functional scenario

{withdraw_comm}[Receive_Command11, Check_Password11, Withdraw11]{cash}

<table>
<thead>
<tr>
<th>Process</th>
<th>Input Variables</th>
<th>Input Data</th>
<th>Output Variables</th>
<th>Output Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Received_Command11</td>
<td>{withdraw_comm}</td>
<td>{“withdraw”}</td>
<td>{sel}</td>
<td>{true}</td>
</tr>
<tr>
<td>Check_Password11</td>
<td>{sel, id, pass, ~Account_file}</td>
<td>{true, 0001, 1111, (0001, “Jack”, 1111, 15000)}</td>
<td>{acc1}</td>
<td>{(0001, “Jack”, 1111, 15000)}</td>
</tr>
<tr>
<td>Withdraw11</td>
<td>{acc1, amount}</td>
<td>{(0001, “Jack”, 1111, 15000), 5000}</td>
<td>{cash, Account_file}</td>
<td>{5000, (0001, “Jack”, 1111, 10000)}</td>
</tr>
</tbody>
</table>
Goal of testing: $S \subseteq P$

$P$ is a refinement of $S$
Steps of Specification-Based Testing

Three steps:

No. 1 Generate test cases based on the specification (reuse the test cases generated for specification animation)

No. 2 Run the program with the test cases.

No. 3 Analyze test results to determine whether the program contains bugs.
Test Strategy

① Ensure that all of the representative program paths are traversed.

② Ensure that all of the traversed program paths are correct.
Ideal Effect of the Testing

Press a Button

Method(int x, int y, int z) {
  int w;  
  if (x < y) {
    w = y / x;  
    while (w < z) {
      ...
    }
  } else {
    ...
  }
}

Adequate test cases

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>case1</td>
<td>3</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>case2</td>
<td>0</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>case3</td>
<td>9</td>
<td>3</td>
<td>35</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Techniques for implementing the test strategy

① Effective methods for test case generation based on formal specifications.

② Combination of functional scenario-based testing and inspection.

③ Combination of functional scenario-based testing and Hoare logic
Effective methods for test case generation based on formal specifications.

A) Functional scenario-based test case generation method

B) “Vibration” test case generation method
Scenario-based testing: a strategy for "divide and conquer"

Specification (in SOFL)

process A(x: int) y: int
pre x > 0
post (x > 10 => y = x + 1) and
(x <= 10 => 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")
}

Functional scenario:
A_pre \land G_i \land D_i
(i=1,...,n)

Functional scenarios
f_1
f_2
...
...

Program paths
p_1
p_2
...
p_m
**Specification:**

process A(x: int) y: int

pre  x > 0

post (x > 10 => y = x + 1) and
     (x <= 10 => y = x - 1)

end_process

**Program:**

`statement`

```
C1

C2

C3

C4

C5

C6

C7
```

**Derivation**

Functional scenarios:

- f_1
- f_2
- ...
- f_n
Definition (FSF): Let

\[
S_{post} \equiv (G_1 \land D_1) \lor (G_2 \land D_2) \lor \cdots \lor (G_n \land D_n),
\]

where \( G_i \) is a guard condition and \( D_i \) is a defining condition, \( i = 1, \ldots, n \).

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

\[
(S_{pre} \land G_1 \land D_1) \lor (S_{pre} \land G_2 \land D_2) \lor \cdots \lor (S_{pre} \land G_n \land D_n)
\]

where

\( f_i = S_{pre} \land G_i \land D_i \) is called a functional scenario

\( S_{pre} \land G_i \) is called a test condition
Test case generation criterion:

Let operation $S$ have an FSF:

$$(S_{\text{pre}} \land G_1 \land D_1) \lor (S_{\text{pre}} \land G_2 \land D_2) \lor \cdots \lor (S_{\text{pre}} \land G_n \land D_n), \text{ where } (n \geq 1).$$

Let $T$ be a test set for $S$. Then, $T$ must satisfy the condition

$$(\forall i \in \{1, \ldots, n\} \exists t \in T \cdot S_{\text{pre}}(t) \land G_i(t))$$

and

$$\exists t \in T \cdot \neg S_{\text{pre}}(t)$$

where $\neg S_{\text{pre}}(t)$ describes an exceptional situation.
Test oracle for test result analysis in the scenario-based testing

**Definition:** Let $\text{Spre} \land \text{G} \land \text{D}$ be a functional scenario and $\text{T}$ be a test set generated from its test condition $\text{Spre} \land \text{G}$. If the condition

$$\exists t \in \text{T} \cdot \text{Spre}(t) \land \text{G}(t) \land \neg \text{D}(t, \text{P}(t))$$

holds, it indicates that a bug in program $\text{P}$ is found by $t$ (also by $\text{T}$).
A “Vibration” method for test set generation

Let $E_1(x_1, x_2, ..., x_n) \mathbin{R} E_2(x_1, x_2, ..., x_n)$ denote that expressions $E_1$ and $E_2$ have relation $R$, where $x_1, x_2, ..., x_n$ are all input variables involved in these expressions.

**Question:** how can test cases be generated based on the relation so that they can quickly cover all of the paths implementing the functional scenario involving the relation in the specification?
V-Method:

We first produce values for $x_1, x_2, \ldots, x_n$ such that the relation $E_1(x_1, x_2, \ldots, x_n) \ R \ E_2(x_1, x_2, \ldots, x_n)$ holds with an initial "distance" between $E_1$ and $E_2$, and then repeatedly create more values for the variables such that the relation still holds but the "distance" between $E_1$ and $E_2$ "vibrates" (changes repeatedly) between the initial "distance" and the maximum "distance".
Example: \( E1 > E2 \)
(2) Combination of functional scenario-based testing and inspection.

Step 1: Generate a test case.
Step 2: Execute the program to obtain a traversed path.
Step 3: Inspect the traversed path based on the corresponding functional scenario in the specification.
process ChildTicketDiscount(a: int, np: int) ap: int
pre a > 0 and np > 1
post (a > 12 => ap = np) and
    (a <= 12 => ap = np – np * 0.5)
end_process

Two functional scenarios and one exception:
(1) a > 0 and np > 1 and a > 12 and ap = np
(2) a >0 and np > 1 and a <= 12 and
    ap = np – np * 0.5
(3) a <= 0 or np <= 1 and anything (exception)
int ChildTicketDiscount(int a, int np) {
    (1) If (a > 0 && np > 1) {
        (2) if (a > 12)
            (3) ap := np;
        (4) else ap := np ** 2 – np – np * 0.5;
        (5) return ap;
    (6) else System.out.println(```the
                   precondition is violated.```
Test case and test result

test case: $a = 5$, $np = 2$

test condition: $a > 0$ and $np > 1$ and $a \leq 12$

functional scenario: $a > 0$ and $np > 1$ and

$a \leq 12$ and $ap = np - np \times 0.5$

traversed program path:

$[(1)(2)'(4)(5)]$

That is:

(1) $a > 0$ && $np > 1$

(2’) $a \leq 12$

(4) $ap := np \times 2 - np - np \times 0.5$

(5) return $ap$
Checklist derived from the functional scenario:

(1) Is the pre-condition $a > 0$ and $np > 1$ implemented correctly?
(2) Is the guard condition $a \leq 12$ implemented correctly?
(3) Is the defining condition $ap = np - np \times 0.5$ implemented correctly?

By trying to answer the above questions, the traversed path can be inspected.
(3) Combination of functional scenario-based testing and Hoare logic:

process `A(a: int) b: int`
pre `Pre_A`
post `Post_A`

From `Pre_A` and `Post_A` generate a test case (`a = 2`);
Execute `program_A` to obtain a traversed path `Path1`

Prove `Pre_A => Pre_Path1`

{`Pre_A`}
{`Pre_Path1`}
`Path1`
{`Post_A`}

Determine the correctness of `Path1`
(either by automatic testing or formal proof)
Relevant axioms derived from Hoare logic:

(1) \( \{Q(E/x)\} \ x := E \ {Q} \) (axiom for assignment)

(2) \( \{Q\} \ S \ {Q} \) where \( S \) is one of the non-changing segments, such as the following two:
   
   “return” statement,
   printing statement.

(3) \( \{S \land Q\} \ S \ {Q} \) where \( S \) is a decision, condition, or predicate expression, which is used in an if-then-else statement or a while-loop.
Example

test case: \( a = 5, \ np = 2 \)

test condition: \( a > 0 \) and \( np > 1 \) and \( a \leq 12 \)

functional scenario: \( a > 0 \) and \( np > 1 \) and
\[ a \leq 12 \] and \( ap = np - np \times 0.5 \)

traversed program path:
\[(1)(2)'(4)(5)\]

output \( ap = 1 \)

test result evaluation:
\( a > 0 \) and \( np > 1 \) and \( a \leq 12 \) and not
\[ ap = np - np \times 0.5 \quad (false) \]

No bug is found in this test, although a bug exists on the path.
Step 1: Form the path triple:

\{a > 0 \text{ and } np > 1\}

\[a > 0 \land n_f > 1, \ a \leq 12,\]

\[ap := np \times 2 - np - np \times 0.5,\]

\[\text{return } ap\]

\{a \leq 12 \text{ and } ap = np - np \times 0.5\}
Step 2: Derive the asserted path by applying the **axiom for assignment or non-change segments**:

\[
\{ a > 0 \text{ and } \text{np} > 1 \} \\
\{ a > 0 \text{ and } \text{np} > 1 \text{ and } \} \\
a <= 12 \text{ and } \text{np} ** 2 - \text{np} - \text{np} * 0.5 = \text{np} - \text{np} * 0.5 \\
a > 0 \text{ && np} > 1 \\
\{ a <= 12 \text{ and } \text{np} ** 2 - \text{np} - \text{np} * 0.5 = \text{np} - \text{np} * 0.5 \} \\
a <= 12 \\
\{ a <= 12 \text{ and } \text{np} ** 2 - \text{np} - \text{np} * 0.5 = \text{np} - \text{np} * 0.5 \} \\
ap := \text{np} ** 2 - \text{np} - \text{np} * 0.5 \\
\{ a <= 12 \text{ and } ap = \text{np} - \text{np} * 0.5 \} \\
\text{return ap} \\
\{ a <= 12 \text{ and } ap = \text{np} - \text{np} * 0.5 \} \\
\]

Derived pre-condition
Step 3: Verify the validity of the implication:

\[
a > 0 \text{ and } np > 1 \implies
\]
\[
a > 0 \text{ and } np > 1 \text{ and }
a \leq 12 \text{ and }
np \times 2 - np - np \times 0.5 = np - np \times 0.5
\]

**Methods for verification:**

(1) Automatic testing (effective when the implication does not hold, but may not be effective to give a conclusion when the implication holds)

(2) Formal proof (effective when the implication holds, but full automation may be impossible)
Example of verification by testing

Let $a = 1$

np = 4.

Then, the implication becomes

$$(a > 0 \text{ and } np > 1)[1/a, 4/np] \Rightarrow (a > 0 \text{ and } np > 1 \text{ and } a \leq 12 \text{ and } np \ast^2 2 - np - np \ast 0.5 = np - np \ast 0.5)[1/a, 4/np]$$

Result: \((true \Rightarrow false) \iff \text{fase}\)
5. Open Problems

(1) There is a lack of a theory and method for generating adequate test cases only based on specifications to cover all of the representative paths for any given program (necessary to consider both the program and specification structures, but how?)

(2) How to avoid human impact on the effectiveness of program inspection (automatic inspection?)

(3) How to deal with the program path explosion problem? (when the program contains many nested conditional or iterative constructs)
6. Conclusions

(1) Specification animation can prevent errors and help set up a foundation for implementation and specification-based testing and inspection.

(2) Specification-based testing can be used to check automatically whether a program is consistent with its specification, but it needs review/inspection to enhance its effectiveness in reliability assurance.

(3) Integration of specification animation, testing, and inspection can help reduce time and cost in verification and validation.
7. Future Work

(1) Address the open problems mentioned previously.

(2) Explore techniques for full automation of the integrated method for verification and validation.

(3) Conduct experiments to evaluate the performance of the integrated method.
Thank You!