

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/>

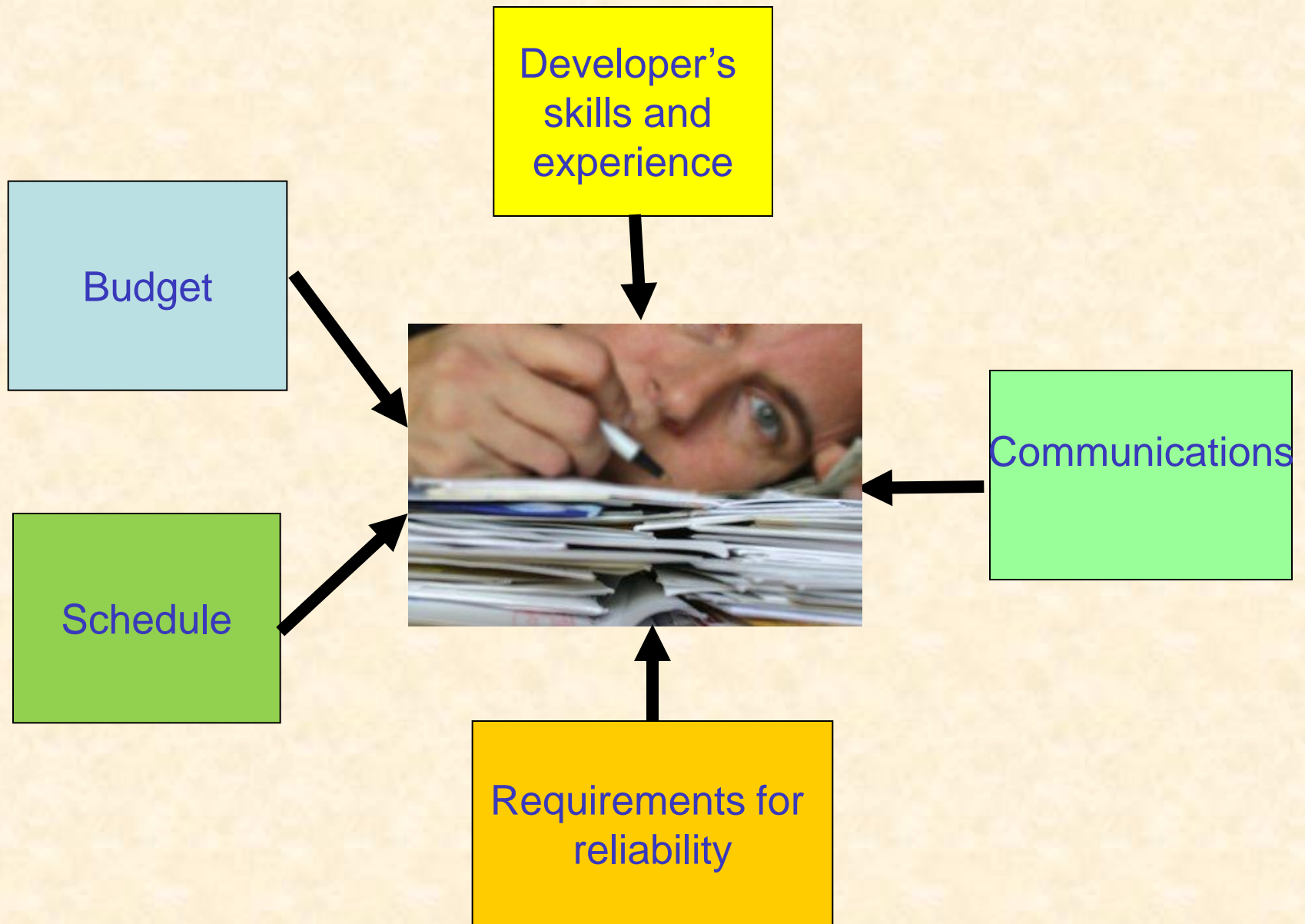
Overview

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



➤ 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 behaves differently from my requirements?

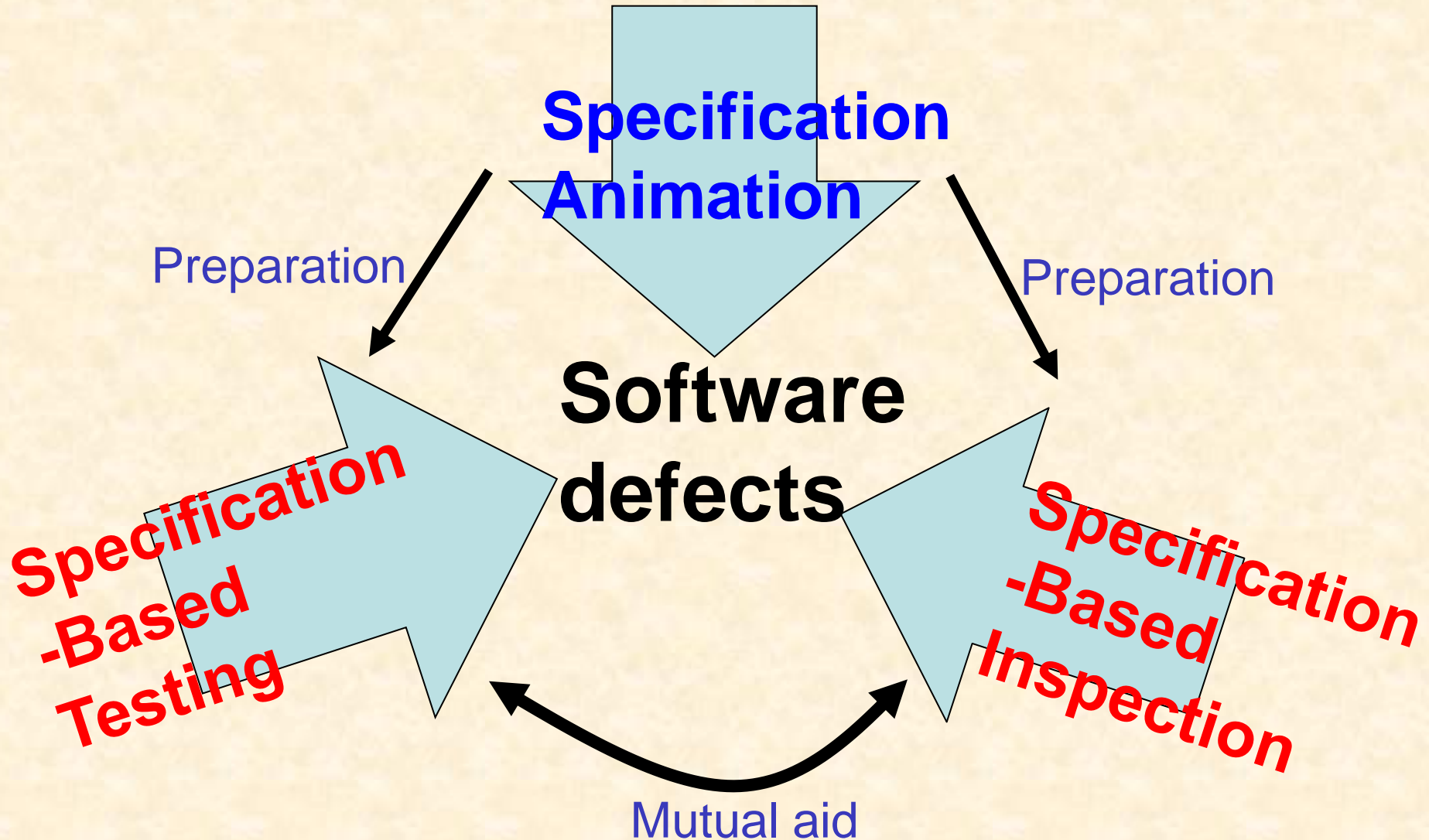


Developer:

Why are there so many bugs remaining in the program?

Why is my own program difficult to understand even by myself?

2. Our Solution



3. Specification Animation

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

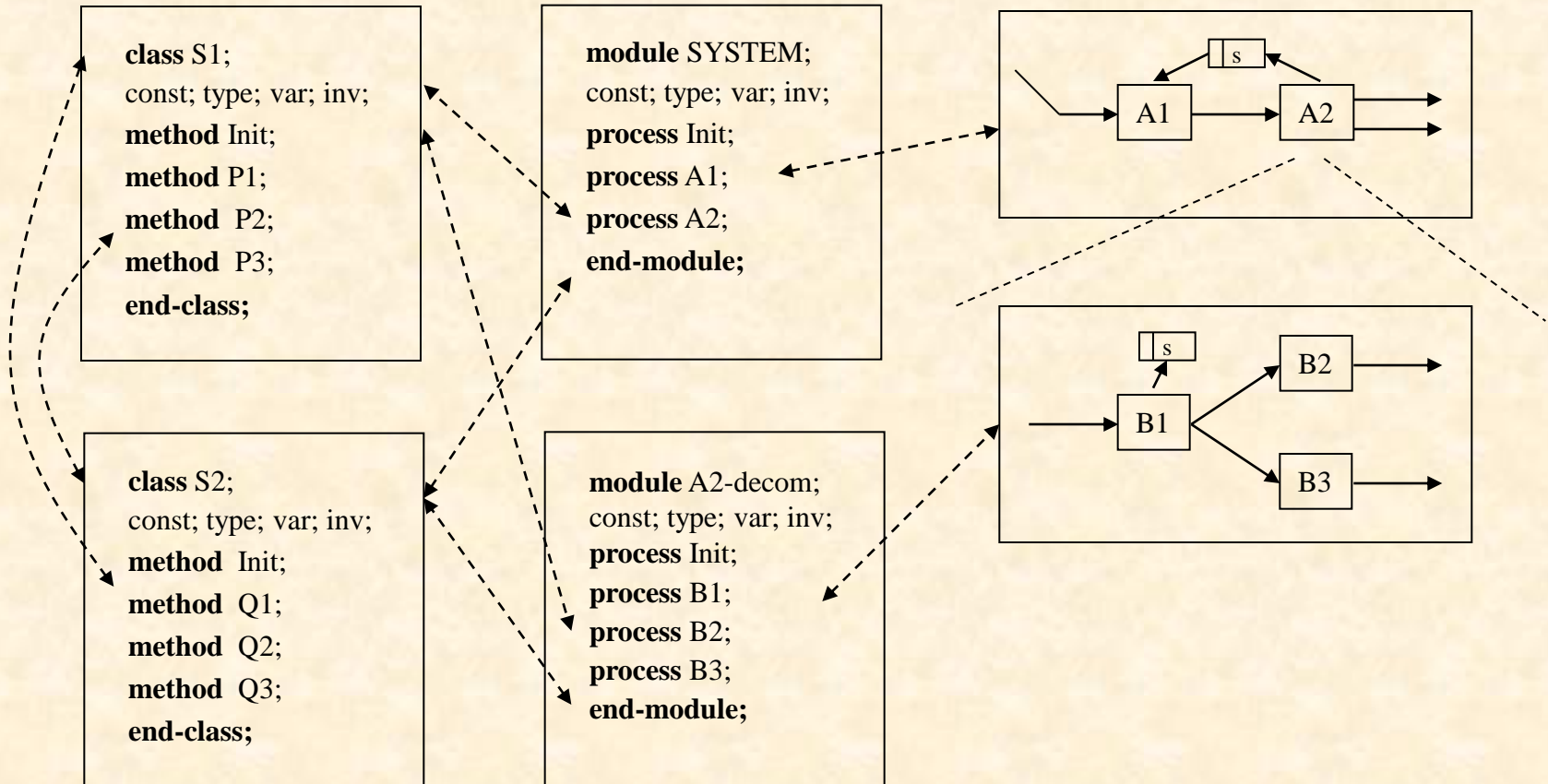
Three expected effects: improving understanding of requirements or designs, strengthening communication, and verifying/validating specifications.



SOFL: Structured Object-oriented Formal Language

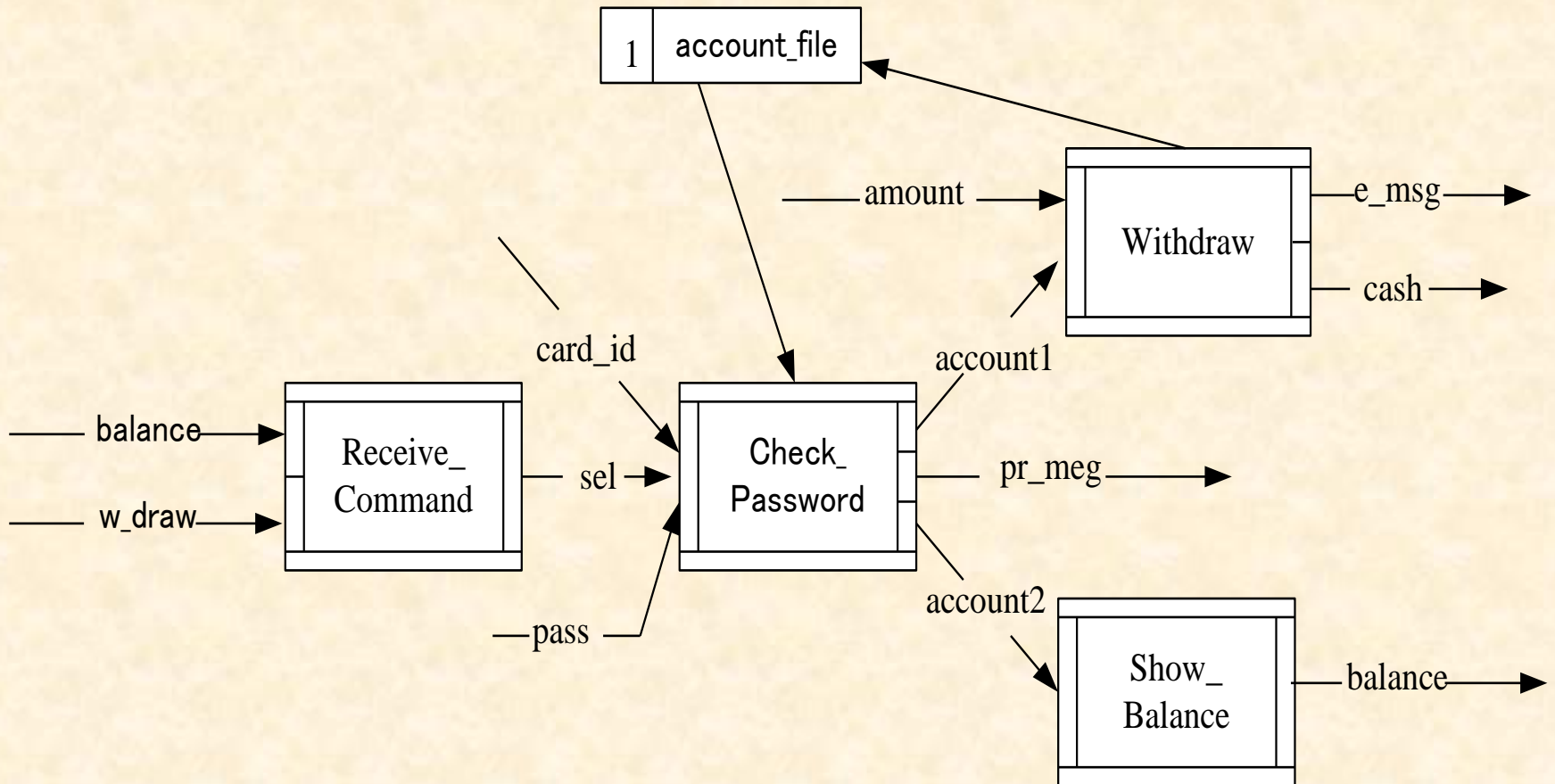
The structure of a SOFL specification:

CDFDs + modules + classes



Example:

A simplified ATM specification in SOFL:



```
module SYSTEM_ATM;
```

```
  type
```

```
    Account = composed of
```

```
      account_no: nat
```

```
      password: nat
```

```
      balance: real
```

```
    end
```

```
  var
```

```
    account_file: set of Account;
```

```
  inv
```

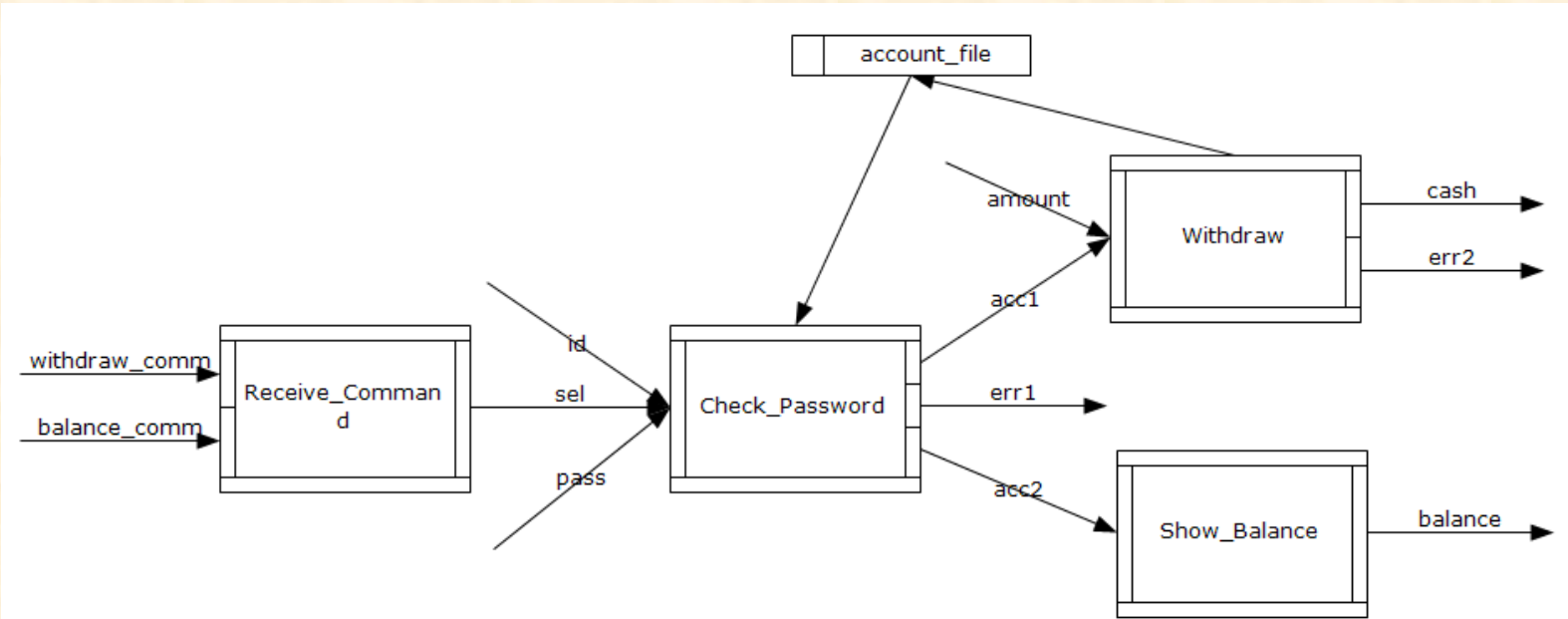
```
    forall[x: account_file] | x.balance >= 0;
```

```
  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."
comment
...
end_process;
end_module
```

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

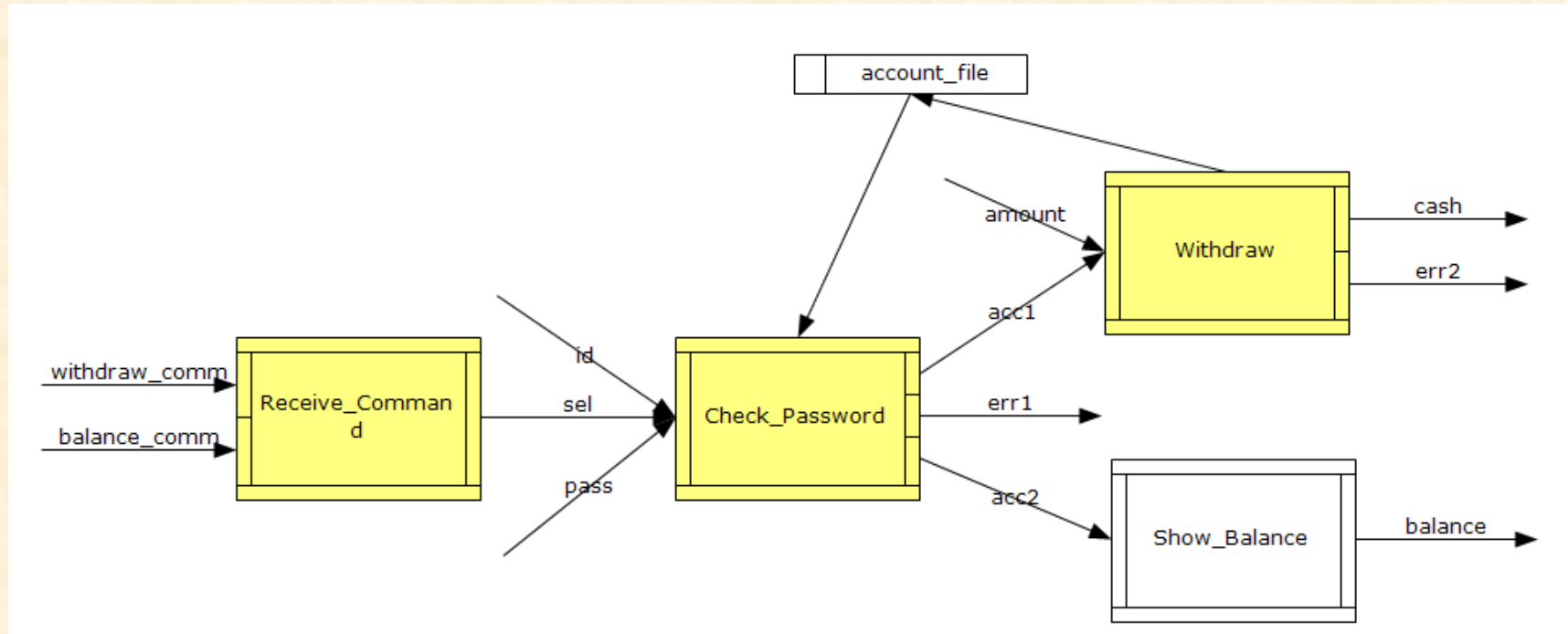
Steps of Animation:

Step1: Deriving system functional scenarios

Step2: Generating test cases

Step3: Carrying out animation for each scenario using the test cases.

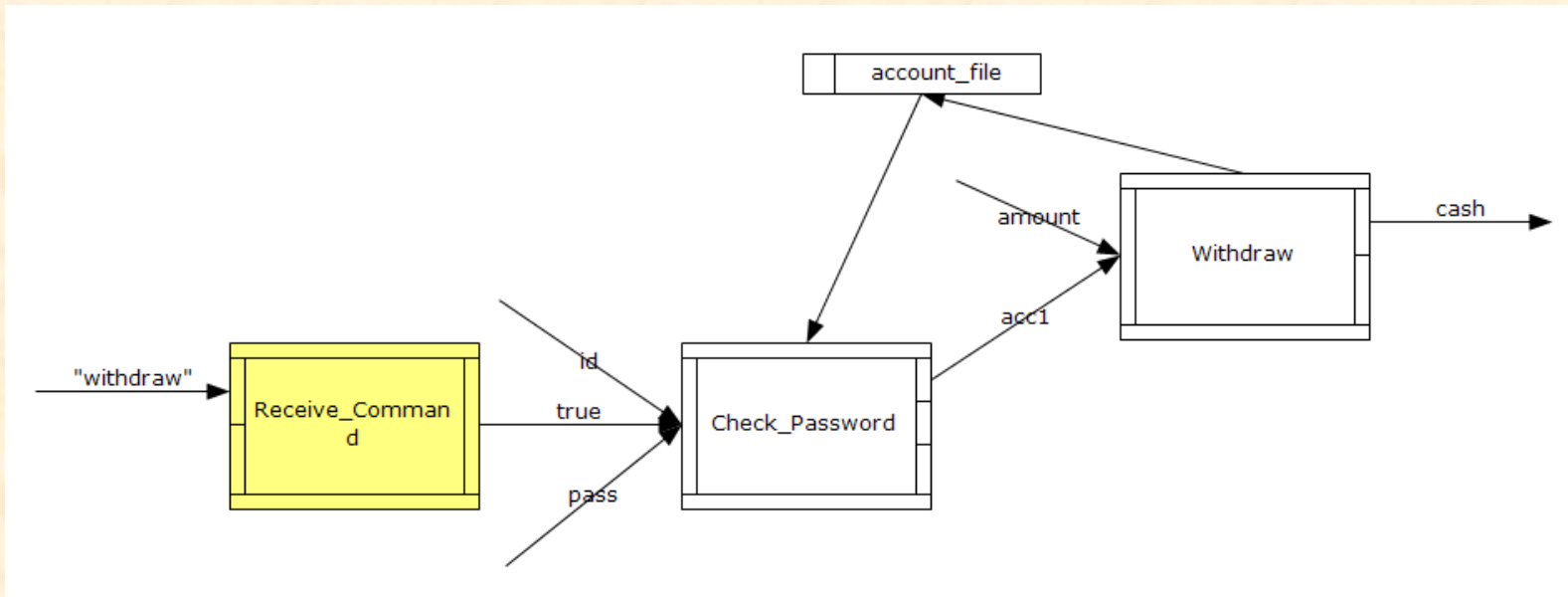
Animation of a single scenario



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

Process	Input Variables	Input Data	Output Variables	Output Data
Received_Command ₁₁	{withdraw_comm}	{"withdraw"}	{sel}	{true}
Check_Password ₁₁	{sel, id, pass, ~Account_file}	{true, 0001, 1111, (0001, "Jack", 1111, 15000)}	{acc1}	{(0001, "Jack", 1111, 15000)}
Withdraw ₁₁	{acc1, amount}	{(0001, "Jack", 1111, 15000), 5000}	{cash, Account_file}	{5000, (0001, "Jack", 1111, 10000)}

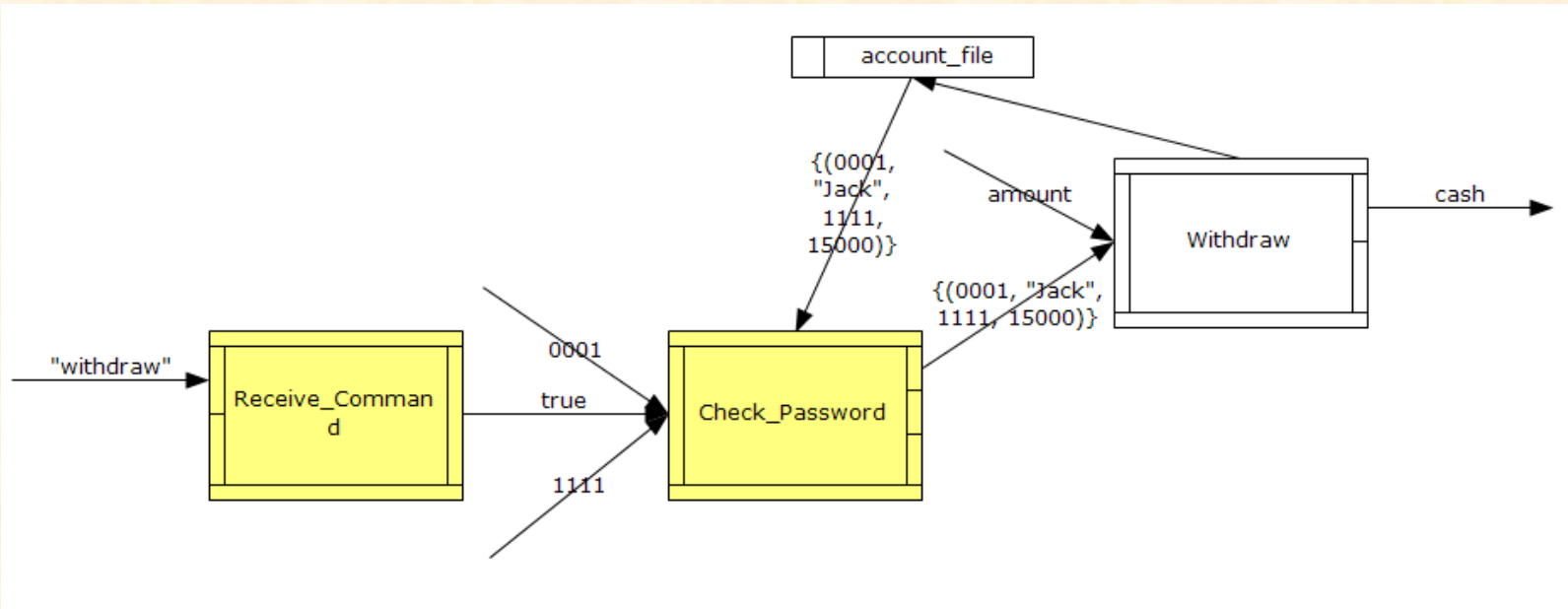
Animation of a single scenario



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

Process	Input Variables	Input Data	Output Variables	Output Data
Received_Command ₁₁	{withdraw_comm}	{"withdraw"}	{sel}	{true}
Check_Password ₁₁	{sel, id, pass, ~Account_file}	{true, 0001, 1111, (0001, "Jack", 1111, 15000)}	{acc1}	{(0001, "Jack", 1111, 15000)}
Withdraw ₁₁	{acc1, amount}	{(0001, "Jack", 1111, 15000), 5000}	{cash, Account_file}	{5000, (0001, "Jack", 1111, 10000)}

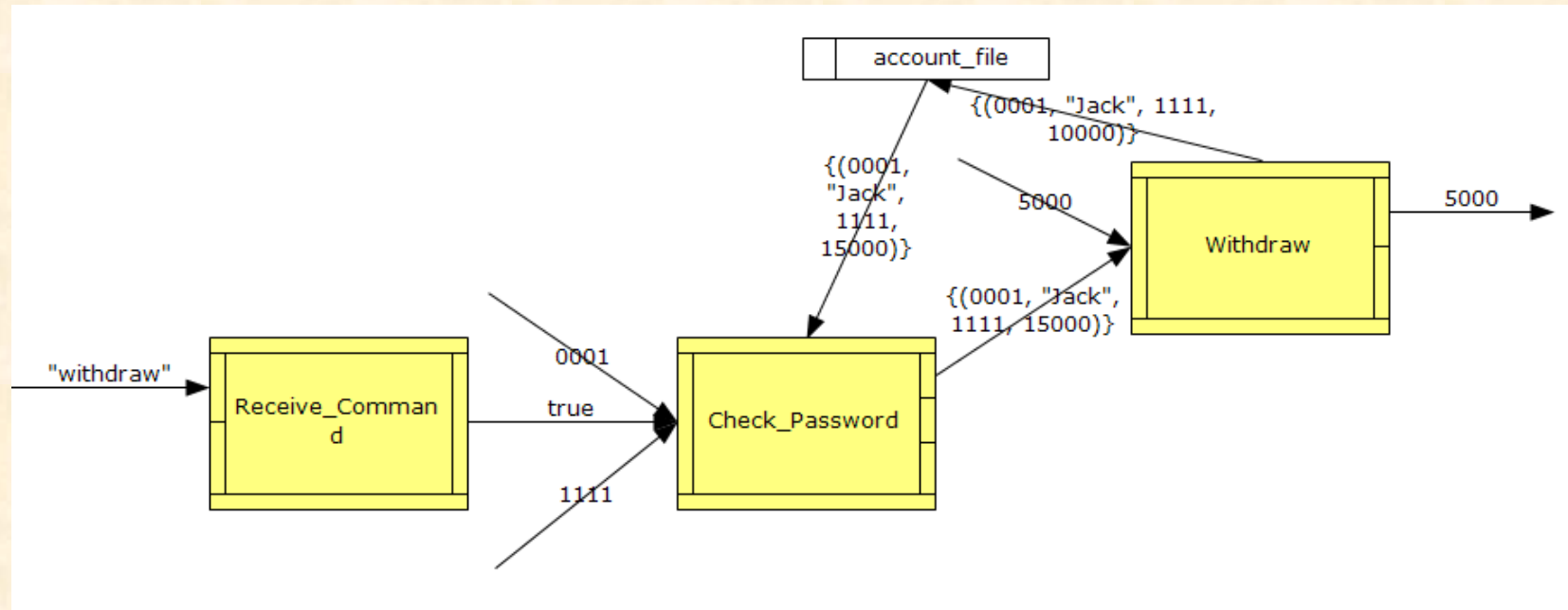
Animation of a single scenario



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

Process	Input Variables	Input Data	Output Variables	Output Data
Received_Command ₁₁	{withdraw_comm}	{"withdraw"}	{sel}	{true}
Check_Password ₁₁	{sel, id, pass, ~Account_file}	{true, 0001, 1111, (0001, "Jack", 1111, 15000)}	{acc1}	{(0001, "Jack", 1111, 15000)}
Withdraw ₁₁	{acc1, amount}	{(0001, "Jack", 1111, 15000), 5000}	{cash, Account_file}	{5000, (0001, "Jack", 1111, 10000)}

Animation of a single scenario



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

Process	Input Variables	Input Data	Output Variables	Output Data
Received_Command ₁₁	{withdraw_comm}	{"withdraw"}	{sel}	{true}
Check_Password ₁₁	{sel, id, pass, ~Account_file}	{true, 0001, 1111, (0001, "Jack", 1111, 15000)}	{acc1}	{(0001, "Jack", 1111, 15000)}
Withdraw ₁₁	{acc1, amount}	{(0001, "Jack", 1111, 15000), 5000}	{cash, Account_file}	{5000, (0001, "Jack", 1111, 10000)}

Test case generation for processes (operations)

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



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:

$A_{pre} \wedge G_i \wedge D_i$

$(i=1, \dots, n)$

A set of functional scenarios

Derivation

f_1
f_2
...
f_n

Definition (FSF): Let

$$S_{\text{post}} \equiv (G_1 \wedge D_1) \vee (G_2 \wedge D_2) \vee \dots \vee (G_n \wedge D_n),$$

where G_i is a **guard condition** and

D_i is a **defining condition**, $i = 1, \dots, n$.

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

$$(S_{\text{pre}} \wedge G_1 \wedge D_1) \vee (S_{\text{pre}} \wedge G_2 \wedge D_2) \vee \dots \vee (S_{\text{pre}} \wedge G_n \wedge D_n)$$

where

$f_i = S_{\text{pre}} \wedge G_i \wedge D_i$ is called a

functional scenario (for generating test cases)

Test case generation criterion:

Let operation **S** have an FSF :

$(S_{pre} \wedge G_1 \wedge D_1) \vee (S_{pre} \wedge G_2 \wedge D_2) \vee \dots \vee (S_{pre} \wedge G_n \wedge D_n)$, where $(n \geq 1)$.

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

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

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

Example

A process specification in SOFL:

```
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) $a > 0$ and $np > 1$ and $a > 12$ and $ap = np$

(2) $a > 0$ and $np > 1$ and $a \leq 12$ and
 $ap = np - np * 0.5$

(3) $a \leq 0$ or $np \leq 1$ and anything

where **anything** means that anything can happen when the pre-condition is violated.

Test case generation

Test cases satisfying functional scenarios:

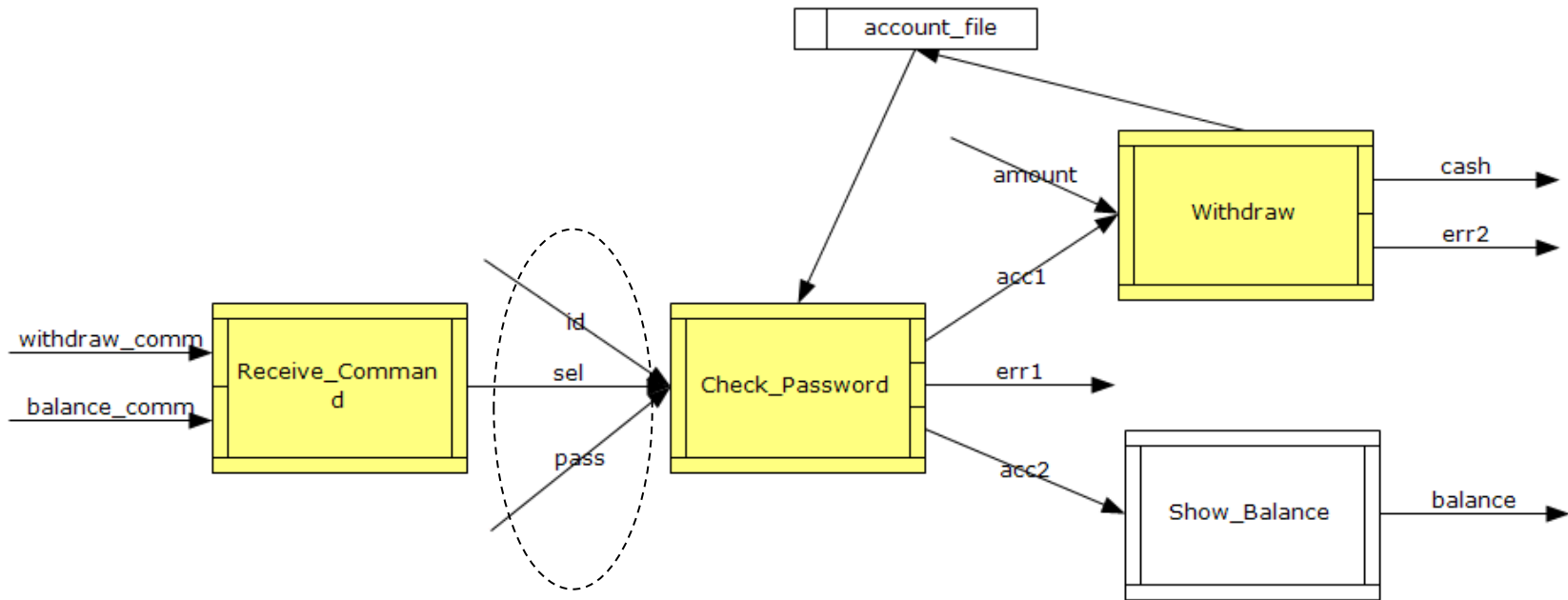
$t1 = \{(a, 15), (np, 100), (ap, 100)\}$

$t2 = \{(a, 10), (np, 100), (ap, 50)\}$

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

$t3 = \{(a, 0), (np, 200), (ap, 100)\}$

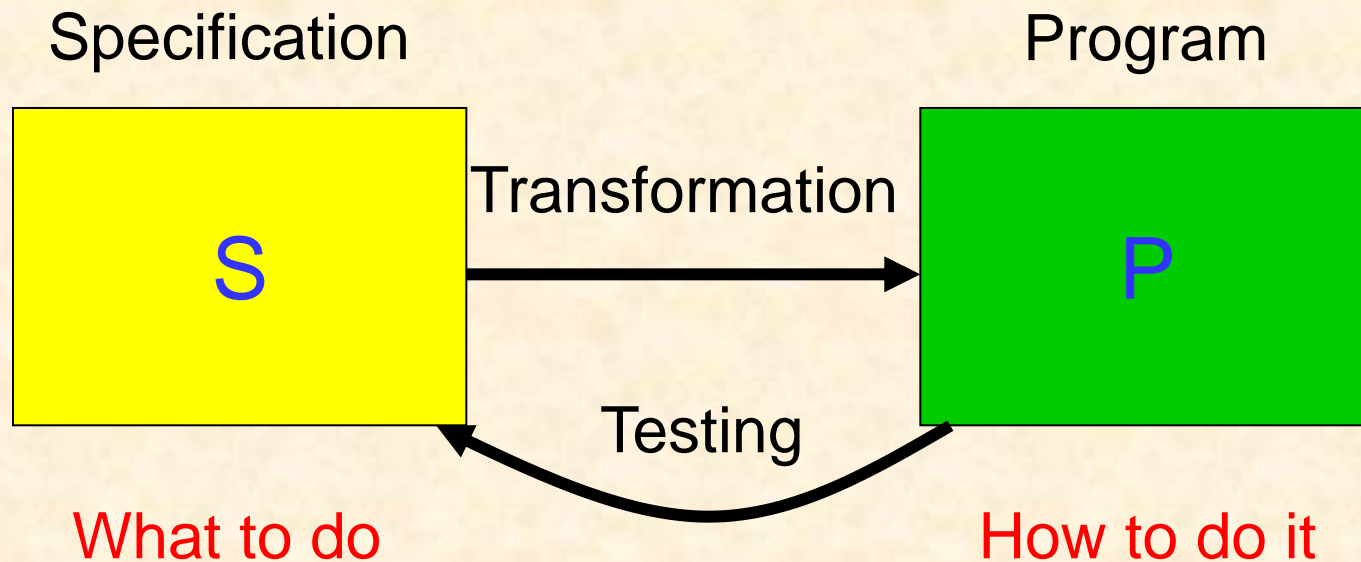
Test case generation within a system functional scenario



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

Process	Input Variables	Input Data	Output Variables	Output Data
Received_Command ₁₁	{withdraw_comm}	{"withdraw"}	{sel}	{true}
Check_Password ₁₁	{sel, id, pass, ~Account_file}	{true, 0001, 1111, (0001, "Jack", 1111, 15000)}	{acc1}	{(0001, "Jack", 1111, 15000)}
Withdraw ₁₁	{acc1, amount}	{(0001, "Jack", 1111, 15000), 5000}	{cash, Account_file}	{5000, (0001, "Jack", 1111, 10000)}

4. Specification-Based Program Testing and Inspection



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.

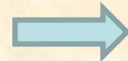
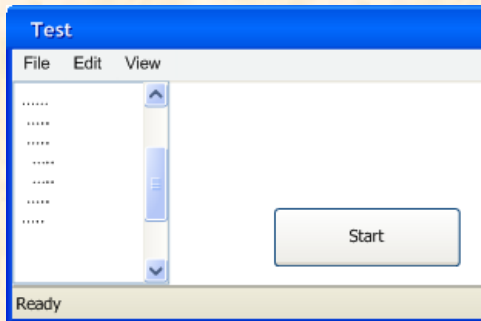
No. 3 Analyze test results to determine whether the program contains bugs.

Test Strategy

- ① Ensure that all of the representative program paths are traversed.
- ② Ensure that all of the traversed program paths are correct.

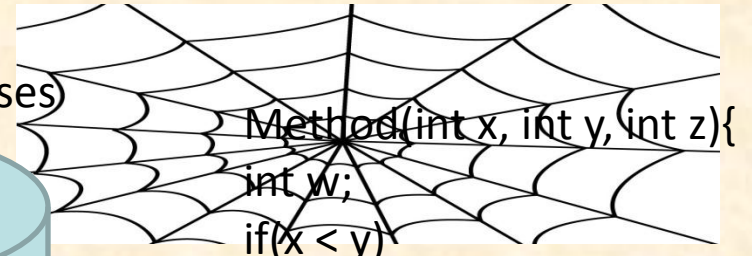
Ideal Effect of the Testing

Press a Button






Adequate test cases)

	x	y	z
case1	3	5	2
case2	0	4	9
case3	9	3	35
.....			
.....			



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



Next

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

① 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
```

Functional scenario:

$A_{pre} \wedge G_i \wedge D_i$

$(i=1, \dots, n)$

Program

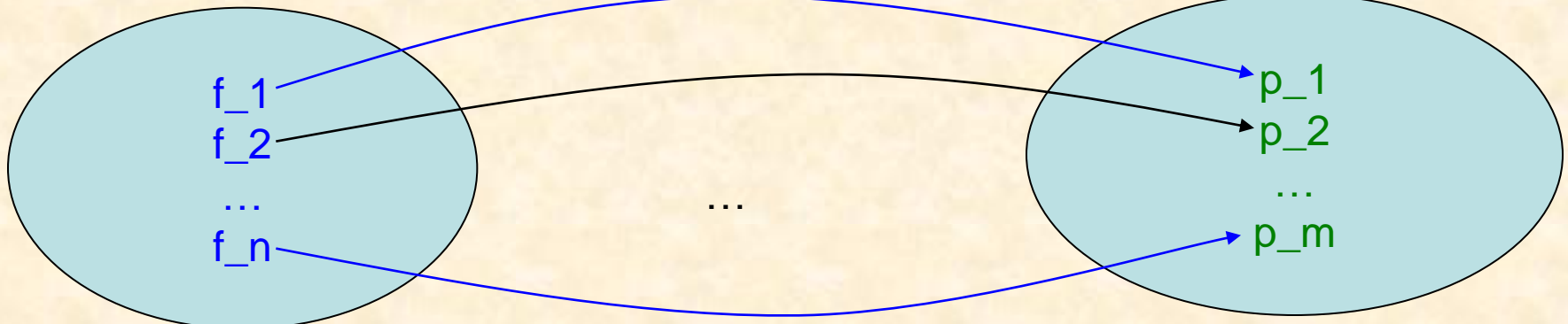
```
int A(int x) {
  If (x > 0) {
    if (x > 10) y := x * 1;
    else y := x - 1;
    return y; }
  else System.out.println("the
    pre is violated") }
```

Satisfy?

Functional scenarios

M

Program paths



Specification:

```
process A(x: int) y: int
```

```
pre  x > 0
```

```
post (x > 10 => y = x + 1) and
```

```
     (x <= 10 => y = x - 1)
```

```
end_process
```

Derivation

Functional scenarios

f_1

f_2

...

f_n

Program:

statement

C1

C2

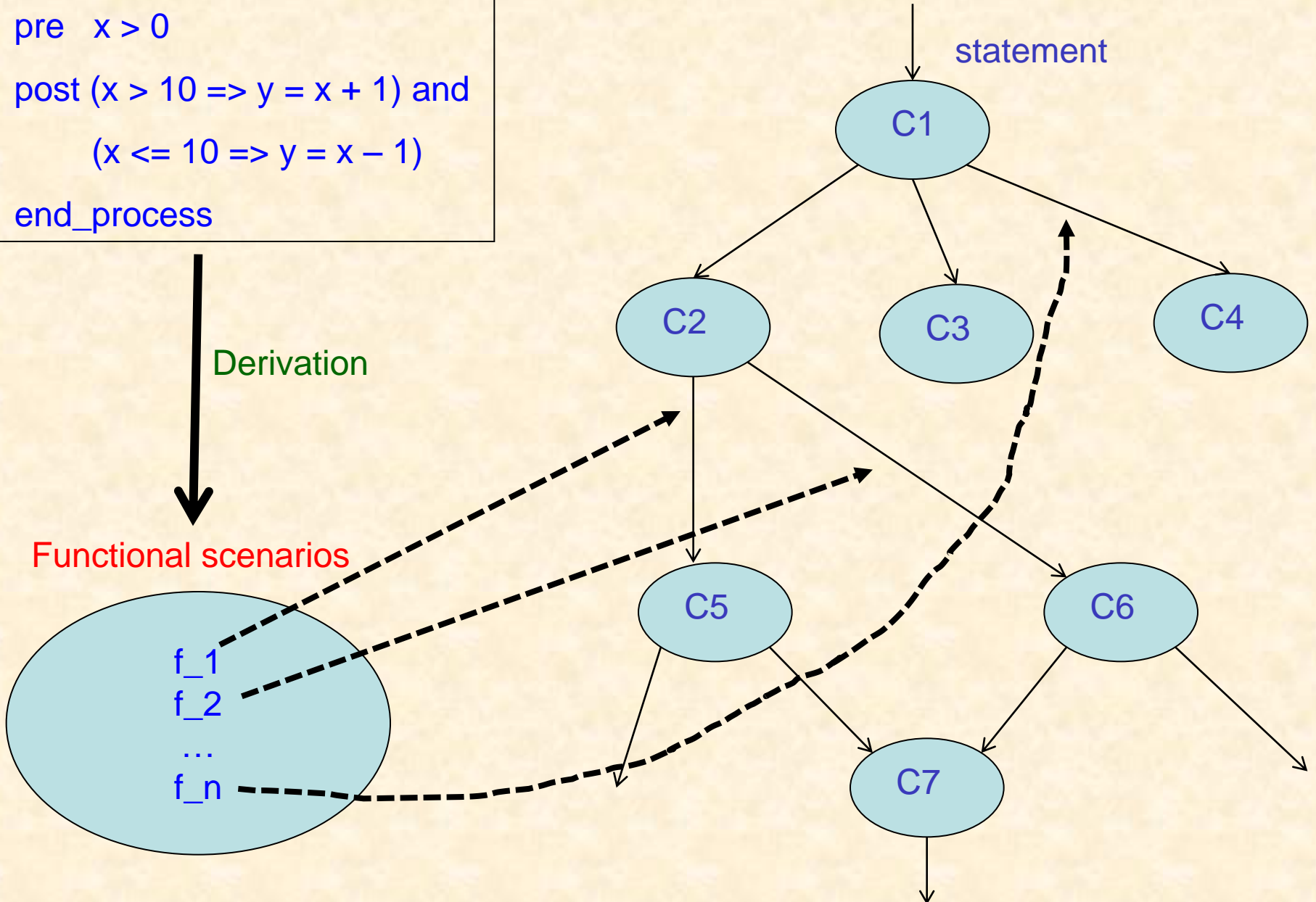
C3

C4

C5

C6

C7



Definition (FSF): Let

$$S_{\text{post}} \equiv (G_1 \wedge D_1) \vee (G_2 \wedge D_2) \vee \dots \vee (G_n \wedge D_n),$$

where G_i is a **guard condition** and

D_i is a **defining condition**, $i = 1, \dots, n$.

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

$$(S_{\text{pre}} \wedge G_1 \wedge D_1) \vee (S_{\text{pre}} \wedge G_2 \wedge D_2) \vee \dots \vee (S_{\text{pre}} \wedge G_n \wedge D_n)$$

where

$f_i = S_{\text{pre}} \wedge G_i \wedge D_i$ is called a **functional scenario**

$S_{\text{pre}} \wedge G_i$ is called a **test condition**

Test case generation criterion:

Let operation **S** have an FSF :

$(S_{pre} \wedge G_1 \wedge D_1) \vee (S_{pre} \wedge G_2 \wedge D_2) \vee \dots \vee (S_{pre} \wedge G_n \wedge D_n)$, where $(n \geq 1)$.

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

$(\forall i \in \{1, \dots, n\} \exists t \in T \cdot S_{pre}(t) \wedge G_i(t))$ and
 $\exists t \in T \cdot \neg S_{pre}(t)$

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

Test oracle for test result analysis in the scenario-based testing

Definition: Let $\mathbf{S}_{pre} \wedge \mathbf{G} \wedge \mathbf{D}$ be a functional scenario and \mathbf{T} be a test set generated from its test condition $\mathbf{S}_{pre} \wedge \mathbf{G}$. If the condition

$$\exists t \in \mathbf{T} \cdot \mathbf{S}_{pre}(t) \wedge \mathbf{G}(t) \wedge \neg \mathbf{D}(t, \mathbf{P}(t))$$

holds, it indicates that a bug in program \mathbf{P} is found by \mathbf{t} (also by \mathbf{T}).

A “Vibration” method for test set generation

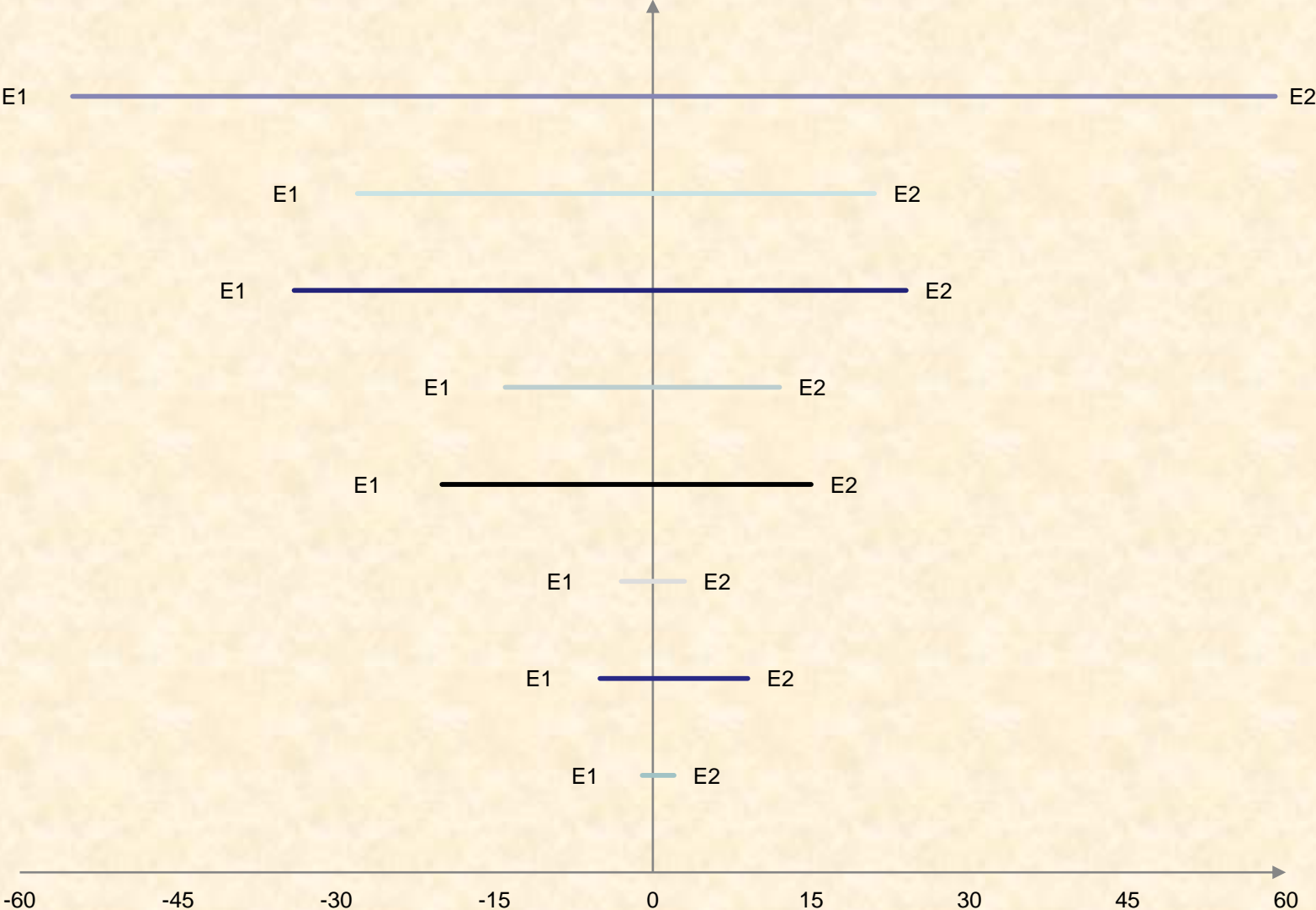
Let $E_1(x_1, x_2, \dots, x_n) R E_2(x_1, x_2, \dots, x_n)$ denote that expressions E_1 and E_2 have relation R , where x_1, x_2, \dots, x_n are all input variables involved in these expressions.

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, \dots, x_n such that the relation $E_1(x_1, x_2, \dots, x_n) R E_2(x_1, x_2, \dots, x_n)$ 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.

Example

```
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 $a \leq 12$ and

$ap = np - np * 0.5$

(3) $a \leq 0$ or $np \leq 1$ and anything (exception)

Implementation of the specification

```
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 $ap = np - np * 0.5$

traversed program path:

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

That is:

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

(2)' $a \leq 12$

(4) $ap := np ** 2 - np - np * 0.5$

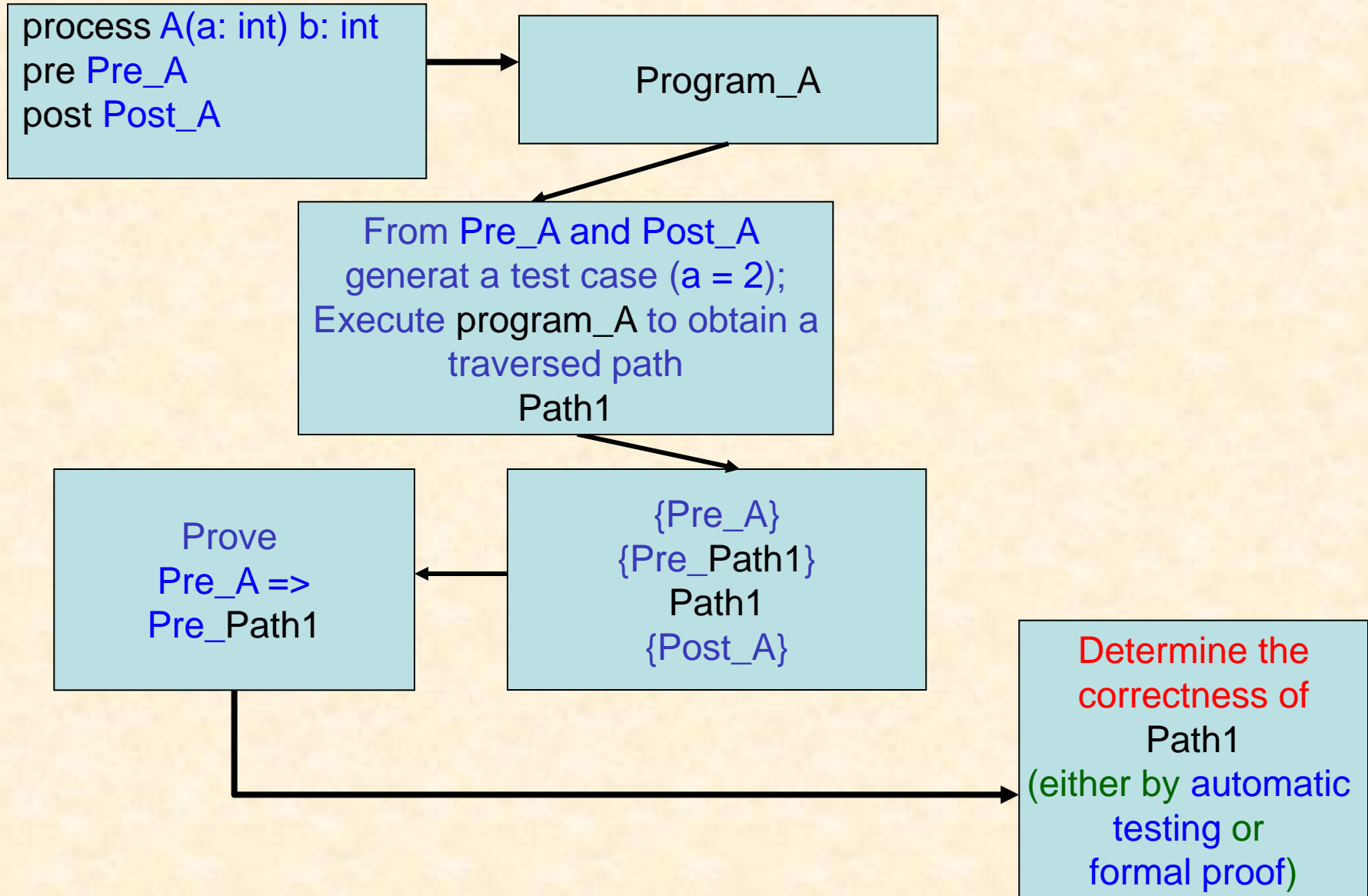
(5) return ap

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 $ap = np - np * 0.5$ 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:



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) $\{S \wedge Q\} S \{Q\}$ 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 $ap = np - np * 0.5$

traversed program path:

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

output $ap = 1$

test result evaluation:

$a > 0$ and $np > 1$ and $a \leq 12$ and not

$ap = np - np * 0.5$ (false)

No bug is found in this test, although a bug exists on the path.

Step1: Form the path triple:

{a > 0 and np > 1}

[a > 0 && n_f > 1, a <= 12,

ap := np ** 2 - np - np * 0.5,

return ap]

{a <= 12 and ap = np - np * 0.5}

Step 2: Derive the asserted path by applying
the axiom for assignment or non-
change segments:

{a > 0 and np > 1}

{a > 0 and np > 1 and

a <= 12 and np ** 2 - np - np * 0.5 = np - np * 0.5}

Derived pre-condition

a > 0 && np > 1

{a <= 12 and np** 2 - np - np * 0.5 = np - np * 0.5}

a <= 12

{a <= 12 and np** 2 - np - np * 0.5 = np - np * 0.5}

ap := np ** 2 - np - np * 0.5

{a <= 12 and ap = np - np * 0.5}

return ap

{a <= 12 and ap = np - np * 0.5}

Step 3: Verify the validity of the implication:

$a > 0$ and $np > 1 \Rightarrow$

$a > 0$ and $np > 1$ and

$a \leq 12$ and

$np ** 2 - np - np * 0.5 = np - np * 0.5$

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

$(a > 0 \text{ and } np > 1)[1/a, 4/np] \Rightarrow$

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

$a \leq 12 \text{ and}$

$np ** 2 - np - np * 0.5 = np - np * 0.5)[1/a, 4/np]$

Result: $(\text{true} \Rightarrow \text{false}) \Leftrightarrow \text{false}$

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 !