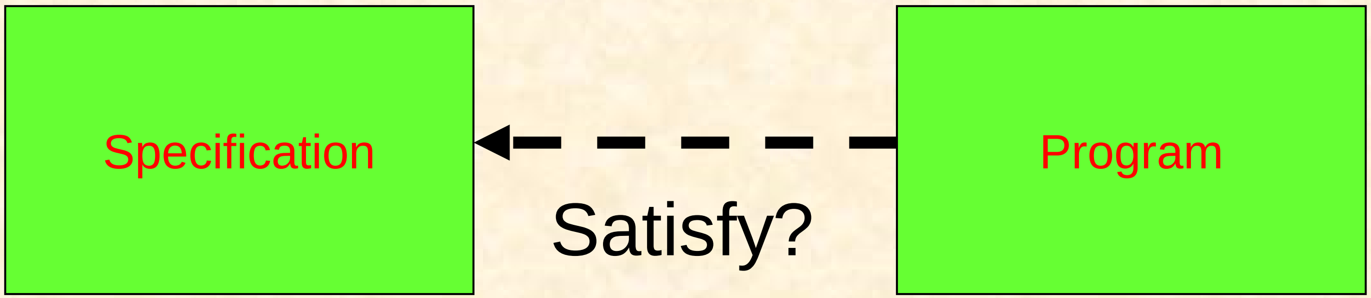
IV. Specification-Based Inspection and Testing

In this part, we will learn two practical techniques for program verification.

(1)Specification-based program inspection

(2)Specification-based program testing

IV.1 Specification-Based Program Inspection



Inspection

(analyze program based on a checklist to find bugs)

Why inspection before testing?

(1) A test may not be carried out effectively due to either crash in execution or non-termination of the program.

(2) Not necessarily all the program paths can be tested.

(3) Even if every path is tested, there is no guarantee that every functional scenario defined in the specification is correctly implemented.

Scenario-based inspection: 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

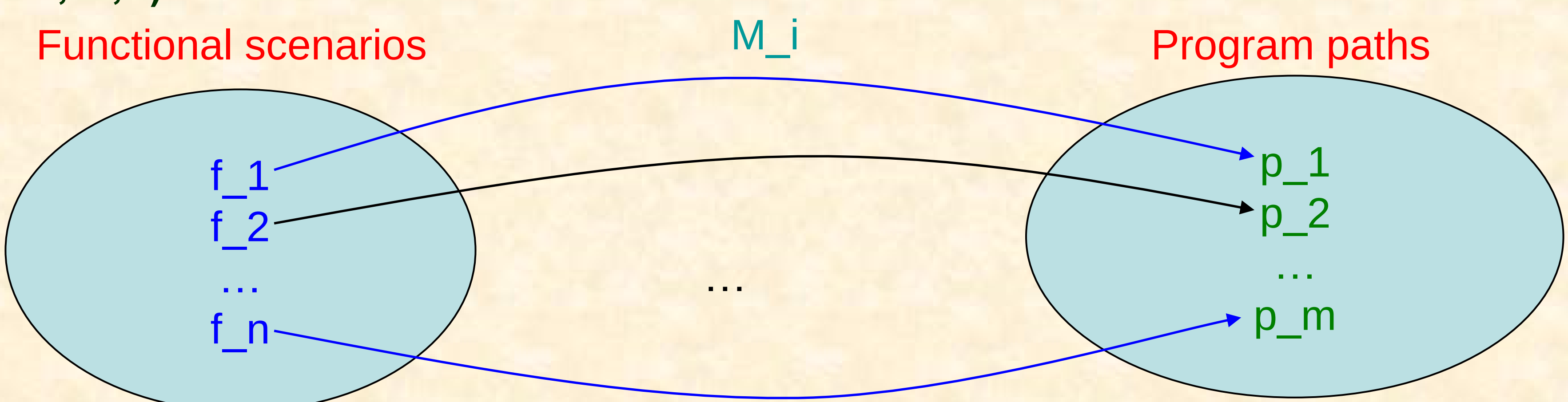
```
int A(int x) {
If (x > 0) {
  if (x > 10)  y = x + 1;
  else  y = x – 1;
   return y; }
 else System.out.println("the
       pre is violated")  }
```

Satisfy?

Functional scenario:

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

(i=1,…,n)



The principle of scenario-based inspection:

(1) Check whether every functional scenario defined in the specification is correctly implemented by a set of program paths in the program.

(2) Every program path contributes to the implementation of some scenario.

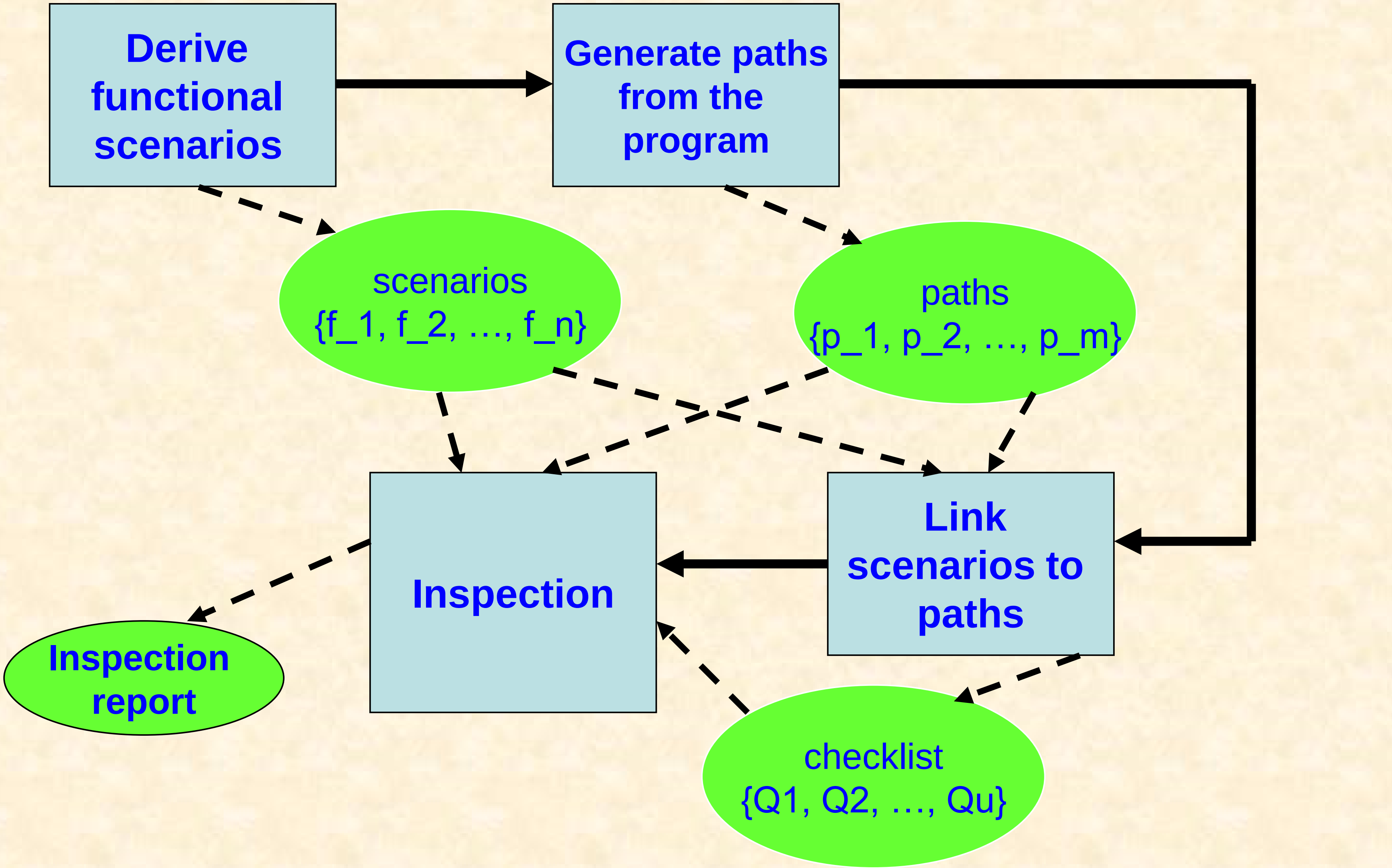
Formally, the principle is described as follows:

$M: \; S \rightarrow power(P)$

$(\forall_{f \in S} \exists_{q \in power(P)} \cdot M(f) = q \wedge q <> \{\,\}) \wedge$

$(\forall_{p \in P} \exists_{f \in S} \cdot p \in M(f))$

A Process for Scenario-Based Inspection



(1) Derivation of functional scenarios from a specification

Definition 1. Let OP be an operation,
pre_OP denote its pre-condition, and
post_OP = G_1 and D_1 or
G_2 and D_2 or···or
G_n and D_n
be it post-condition,
where G_i ($i \in \{1,...,n\}$) is a guard condition and D_i is a defining condition. Then, a functional scenario f_s of OP is a conjunction pre_OP and G_i and D_i, and such a form of pre-post conditions is called functional scenario form or FSF for short. That is,

(pre_OP and G_1 and D_1) or (pre_OP and G_2 and D_2) or … or (pre_OP and G_n and D_n)

For example,

```
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
```

Guard condition

Defining condition

The steps for deriving scenarios

No.1 Transform post_OP into a disjunctive normal form.

No.2 Transform the disjunctive normal form into a functional scenarios form.

No.3 Obtain the set of functional scenarios from the functional scenario form and the pre_OP.

For example, suppose

```
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
```

No.1 Transform post-condition into a disjunctive normal form:

x > 10 and y = x + 1 or

x <= 10 and y = x – 1

No.2 Transform the disjunctive normal form into the functional scenario form:

x > 0 and x > 10 and y = x + 1 or
x > 0 and x <= 10 and y = x – 1 or
not x > 0

No. 3 Obtain the following functional scenarios:

f_1: x > 0 and x > 10 and y = x + 1
f_2: x > 0 and x <= 10 and y = x – 1
f_3: not x > 0

More complicated example

Formal specification

```
process M(x, y: int) z: int
ext wr w: real
pre x <> y
post
x > 0 and
z = y / x and
w > ~w**2 and
x >= y or
x > 0 and
z = x * y and
x < y and
w = z * ~w or
x = 0 and
z = y and
w = ~w or
x < 0 and
z = x + y + ~w and
w < ~w
```

FSF of the specification

pre_M and C1 and D1
or
pre_M and C1 and D1
or
…
or
pre_M and C1 and D1

C1: guard condition

D1: defining condition

~pre_M: pre-condition

Example

Formal specification

M(x, y: int)z: int
ext wr w: real
pre x <> y
post
x > 0 and
z = y / x and
w > w~2 and**
x >= y or
x > 0 and
z = x * y and
x < y and
w = z * w~ or
x = 0 and
z = y and
w = w~ or
x < 0 and
z = x + y + w~ and
w < w~

Formal specification

~pre_M G1

D1

x <> y and x > 0 and x >= y and
z = y / x and w > ~w2**
or
X <> y and x > 0 and x < y and
Z = x * y and w = z * ~w
or
x <> y and x = 0 and
z = y and w = ~w
or
x <> y and x < 0 and
z = x + y + ~w and
w < ~w

(2) The Generation of execution paths from a program

For example,

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

Generation of paths

```
1
x > 0;
x > 10;
y = x + 1;
return y;
```

```
2
x > 0;
x <= 10;
y = x – 1;
return y;
```

```
3
x <= 0;
System.err.println("
the pre is violated");
```

(3) Linking functional scenarios to their execution paths

Two strategies:

- Forward linking: from scenarios to paths.
- Backward linking: from paths to scenarios.

Techniques for the linking

- Identifying paths by testing, provided that the program can terminate normally.

- Identifying paths by comparing the logical expression of the functional scenario to the statements and conditions in the paths.

Functional scenarios in specification

f_1: x > 0 and x > 10 and y = x + 1

f_2: x > 0 and x <= 10 and y = x – 1

f_3: x <= 0

Execution paths

1
x > 0;
x > 10;
y = x + 1;
return y;

2
x > 0;
x <= 10;
y = x – 1;
return y;

3
x <= 0;
System.err.println(“
the pre is violated”);

(4) Analyzing paths (two techniques)

- **Static checking based on a checklist.**

  Example questions on the checklist:

  (1) Is the guard condition in the scenario implemented accurately in the paths?

  (2) Is every defining condition in the scenario implemented correctly in the paths?

  (3) Is every input, output, and external variable used in the scenario implemented properly in terms of its name, type, and use in the paths?

- **Walkthrough with test cases.**

x = 15

f_1: x > 0 and x > 10 and y = x + 1

f_2: x > 0 and x <= 10 and y = x – 1

f_3: x <= 0

1

x = 15

```
x > 0;
x > 10;
y = x + 1;
return y;
```

2

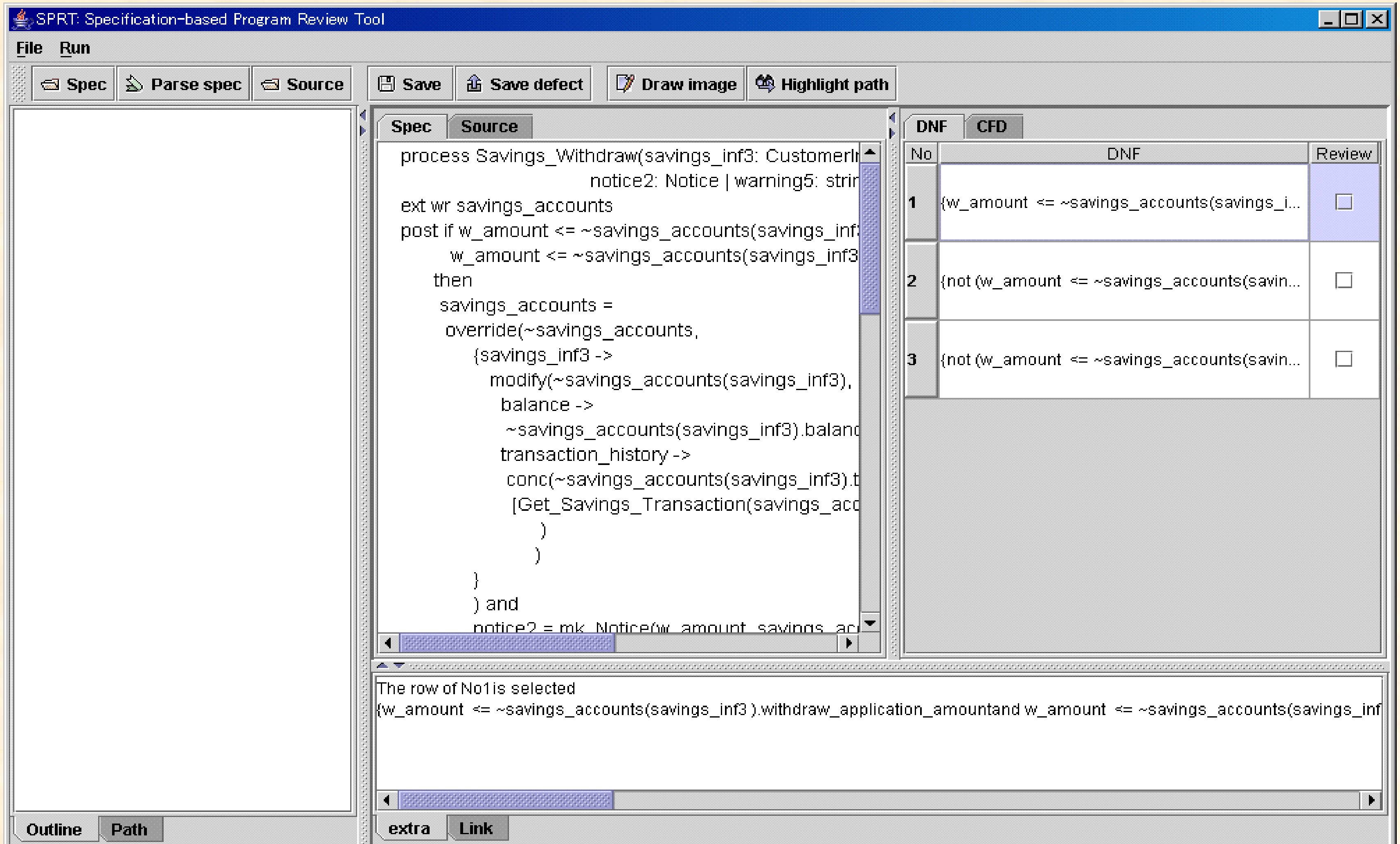
```
x > 0;
x <= 10;
y = x – 1;
return y;
```

3

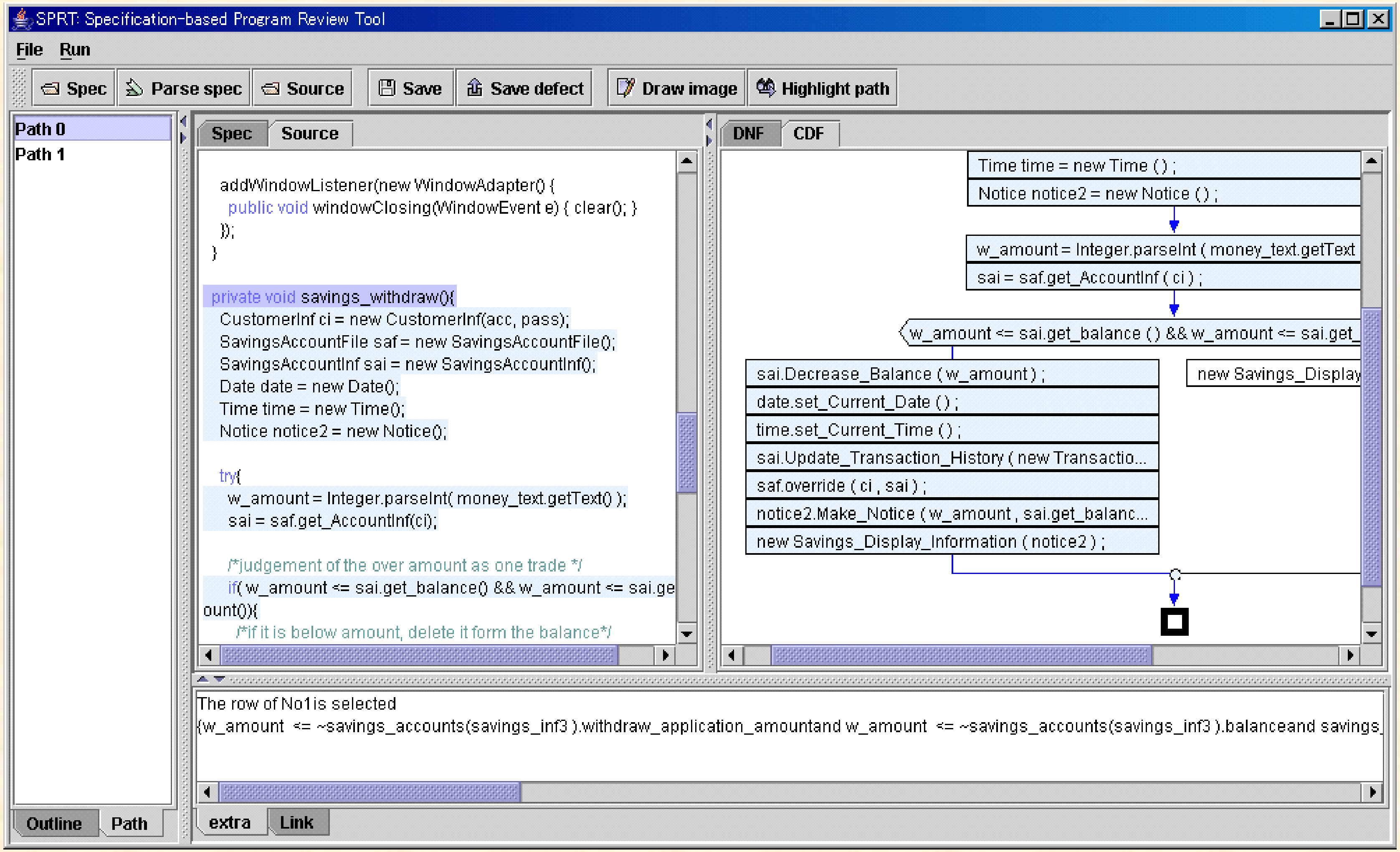
```
x <= 0;
System.err.println("
the pre is violated");
```

(5) A Prototype Software Tool

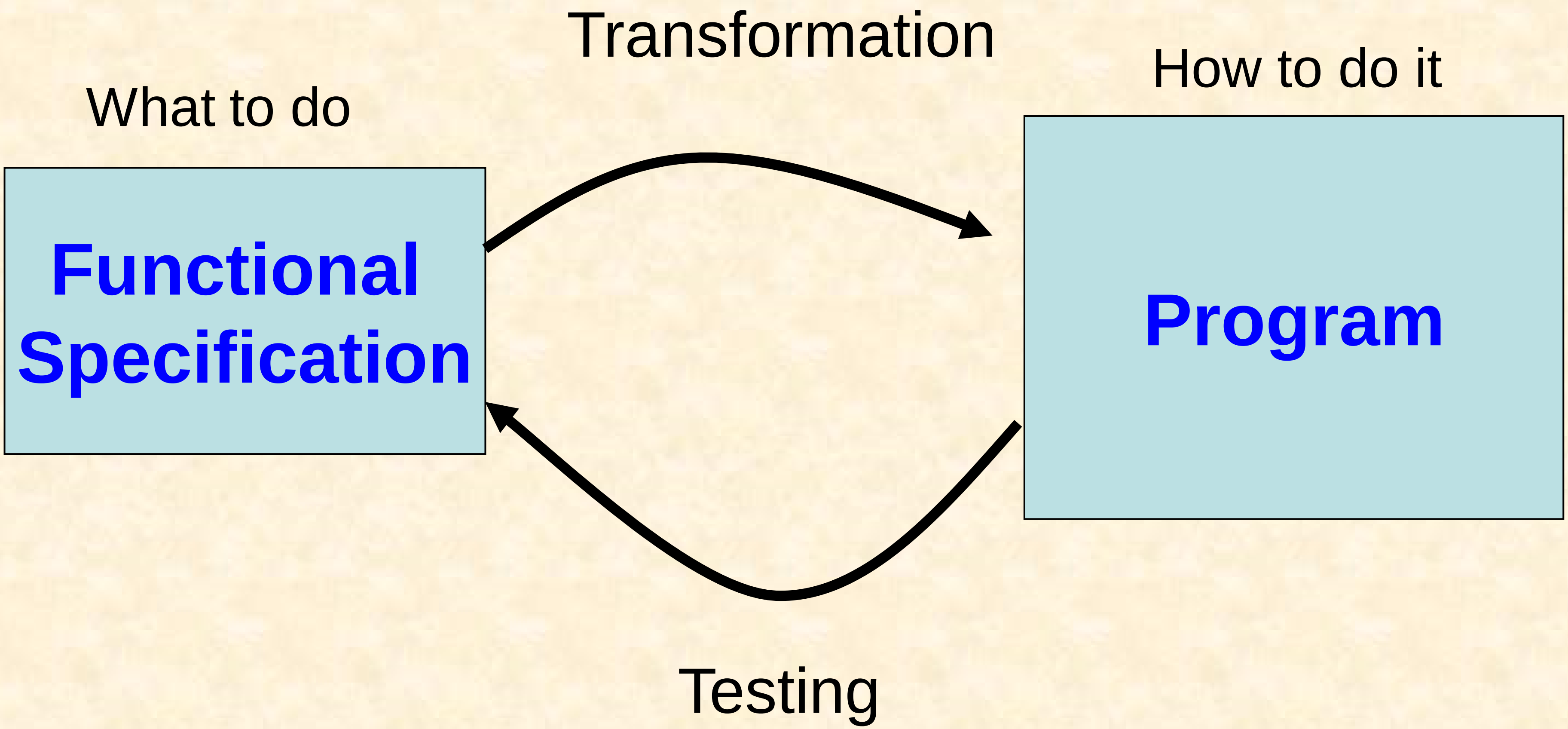
Automatic transformation from a SOFL specification to a set of functional scenarios.



Automatic derivation of program paths from a Java method.



IV.2 Specification-Based Program Testing



The goal:

Dynamically check whether the functions defined in the specification are ``correctly" implemented by the program.

A program **P** correctly implements a specification **S** iff

$$\forall \tilde{\sigma} \in \Sigma \cdot S_{pre}(\tilde{\sigma}) \Rightarrow S_{post}(\tilde{\sigma}, P(\tilde{\sigma}))$$

The features of specification-based testing:

(1) Test cases are generated based on the specification.

(2) The program is executed using the test cases.

(3) Decisions about the existence of bugs in the program are made based on the test cases, execution results, and the specification.

How to automatically test a program to ensure that all the requirements in the specification and all the program paths are checked in specification-based testing.

- Test automation
  - Test case generation
  - Test oracle generation
- Check all the requirements
  - Specification → Functional scenarios
- Check all the program paths
  - Program (Code)
  - Dynamic exploration of paths

Functional scenario-based testing

Specification (in SOFL) | Program

```
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
```

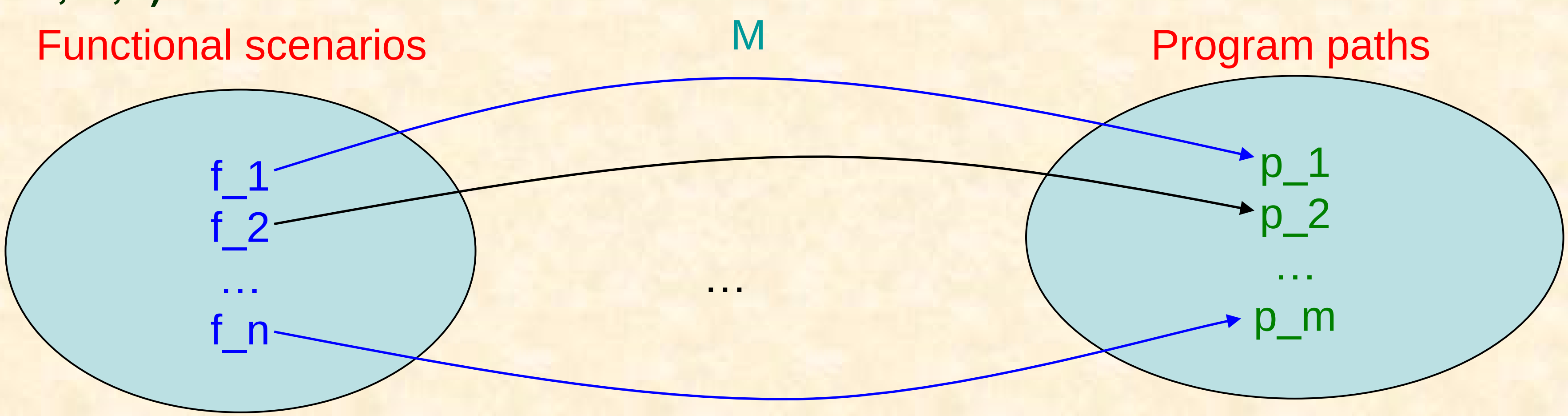
```
int A(int x) {
If (x > 0) {
  if (x > 10)  y := x * 1;
  else  y := x – 1;
   return y; }
 else System.out.println("the
       pre is violated")  }
```

Satisfy?

Functional scenario:

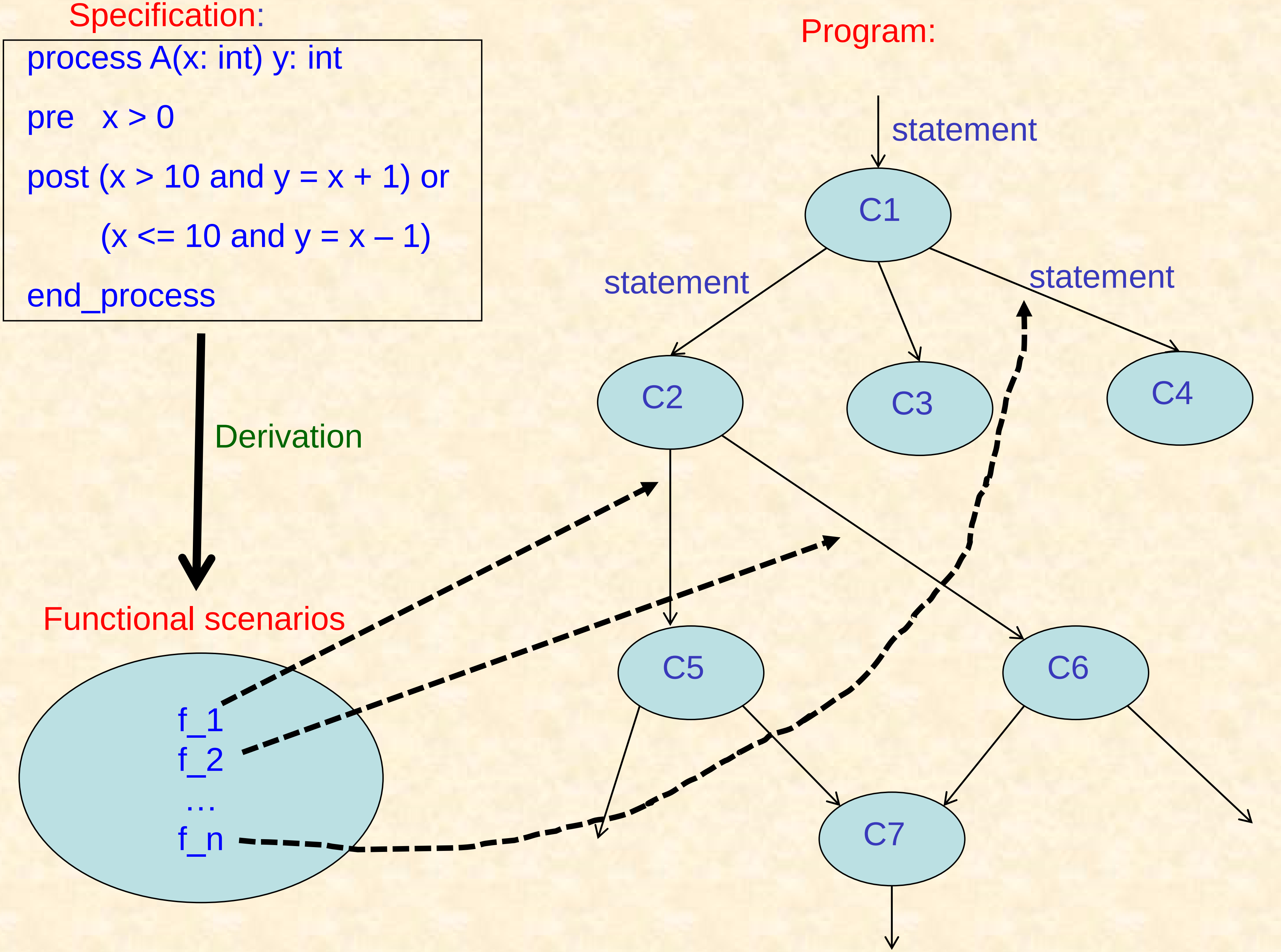
$A_{pre} \wedge G_i \wedge D_i$

(i=1,…,n)

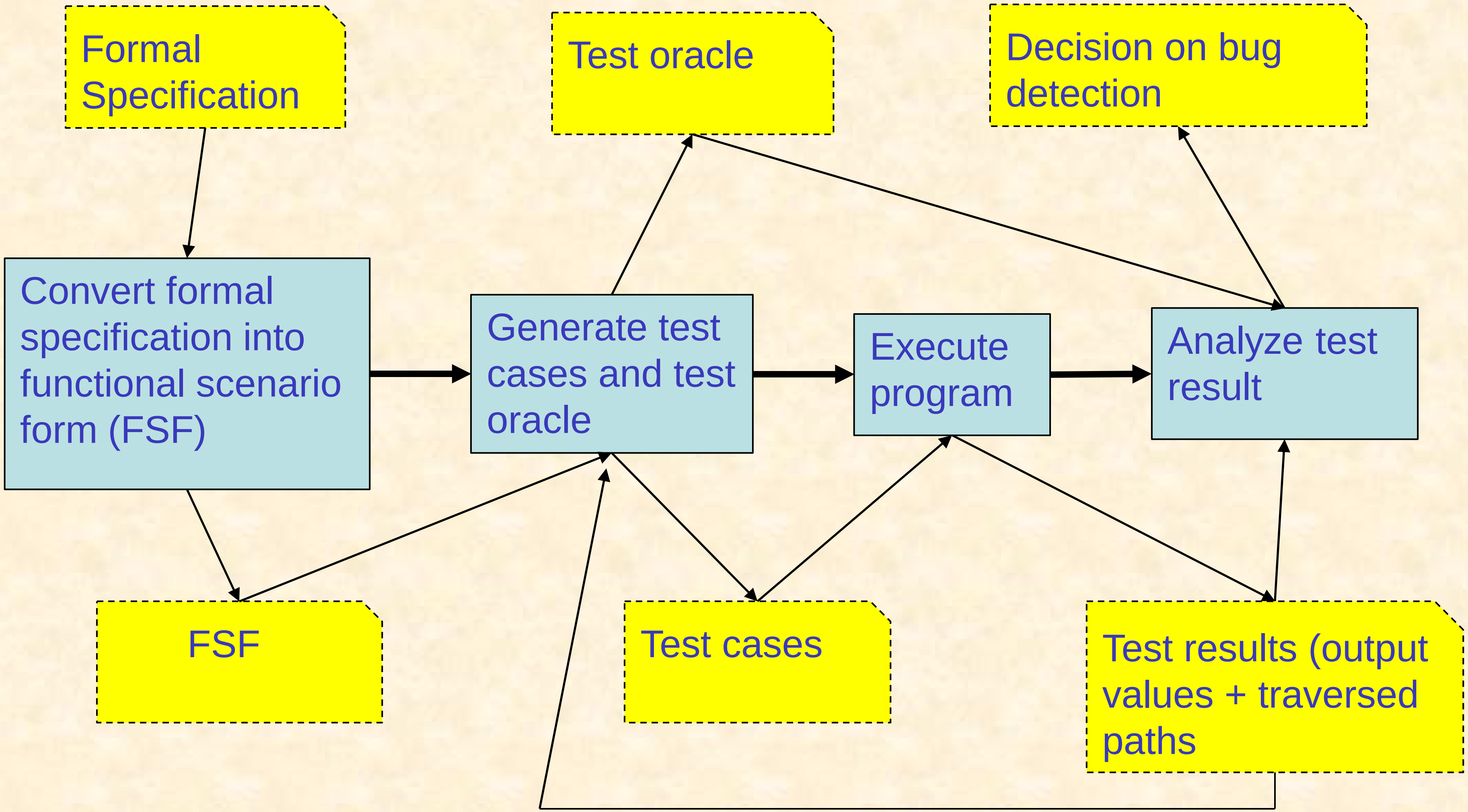


Specification:

```
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
```



The framework for functional scenario-based testing (V-Method)



Functional scenario form (FSF) and functional scenario

Let

denote a process specification. A set of functional scenarios can be derived from the specification, each defining an independent function in terms of input-output relation.

Definition (FSF) Let

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

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:

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

where G_i differs from G_j if i differs from j.

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

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

Definition (complete specification) Let $(S_{pre} \wedge G_1 \wedge D_1) \vee (S_{pre} \wedge G_2 \wedge D_2) \vee \cdots \vee (S_{pre} \wedge G_n \wedge D_n)$ be an FSF of specification S. Then, S is said to be complete if and only if the following condition holds:

$$S_{pre} \Rightarrow G_1 \vee G_2 \vee \cdots \vee G_n$$

The completeness of a specification is a necessary condition for the scenario-based testing method to work effectively.

Example

```
process Tell_Railway_Fare(status : string; fare : nat0)
                         actual_fare : real
ext wr card: Card
pre fare * 0.5 <= card.buffer
post case status of
“Infant” –> actual_fare = 0 and card = ~card;
“Student” –> actual_fare = fare * 0.5 and
                     card = modify(~card, buffer –> ~card.buffer – actual_fare);
“Normal” –> actual_fare = fare and
                    card = modify(~card, buffer –> ~card.buffer – actual_fare);
“Pensioner” –> actual_fare = fare – fare * 0.3 and
                         card = modify(~card, buffer –> ~card.buffer – actual_fare);
“Disable” –> actual_fare = fare - fare * 0.3 and
                     card = modify(~card, buffer –> ~card.buffer – actual_fare);
default –> actual_fare = -1 and card = ~card
        end
end_process
```

Functional scenarios of the process Tell_Railway_Fare specification

S1: fare * 0.5 <= card.buffer and
status = “Infant” and actual_fare = 0 and card = ~card

S2: fare * 0.5 <= card.buffer and
status = “Student” and actual_fare = fare * 0.5 and
card = modify(~card, buffer –> ~card.buffer – actual_fare)

S3: fare * 0.5 <= card.buffer and
status = “Normal” and actual_fare = fare and
card = modify(~card, buffer –> ~card.buffer – actual_fare)

S4: fare * 0.5 <= card.buffer and
status = “Pensioner” and actual_fare = fare – fare * 0.3 and
card = modify(~card, buffer –> ~card.buffer – actual_fare)

S5: fare * 0.5 <= card.buffer and

status = “Disable” and actual_fare = fare - fare * 0.3 and

card = modify(~card, buffer –> ~card.buffer – actual_fare)

S6: fare * 0.5 <= card.buffer and

status notin {“Infant”, “Student”, “Normal”, “Pensioner”,

“Disable”} and

actual_fare = -1 and card = ~card

Test Case Generation Criteria

Generate a test set T based on the FSF of specification S $(S_{pre} \wedge G_1 \wedge D_1) \vee (S_{pre} \wedge G_2 \wedge D_2) \vee \cdots \vee (S_{pre} \wedge G_n \wedge D_n)$ such that the following conditions are satisfied:

(1) For every functional scenario Sc, there must exist a test case tc in T such that tc satisfies the test condition of Sc. Precisely,

$\forall_{i \in \{1,\ldots,n\}} \exists_{tc \in T} \cdot S_{pre}(tc) \wedge G_i(tc)$

(2) If $G_1 \vee G_2 \vee \cdots \vee G_n$ is not a tautology, there must exist a test case tc in T such that tc satisfies the condition:

$S_{pre}(tc) \wedge \neg(G_1 \vee G_2 \vee \cdots \vee G_n)(tc)$

(3)There must exist a test case tc in T such that it satisfies the condition:

$\neg S_{pre}(tc)$

(4) For every path p in the representative path set R_{PP}, there must exist a test case tc in T such that the following condition is satisfied:

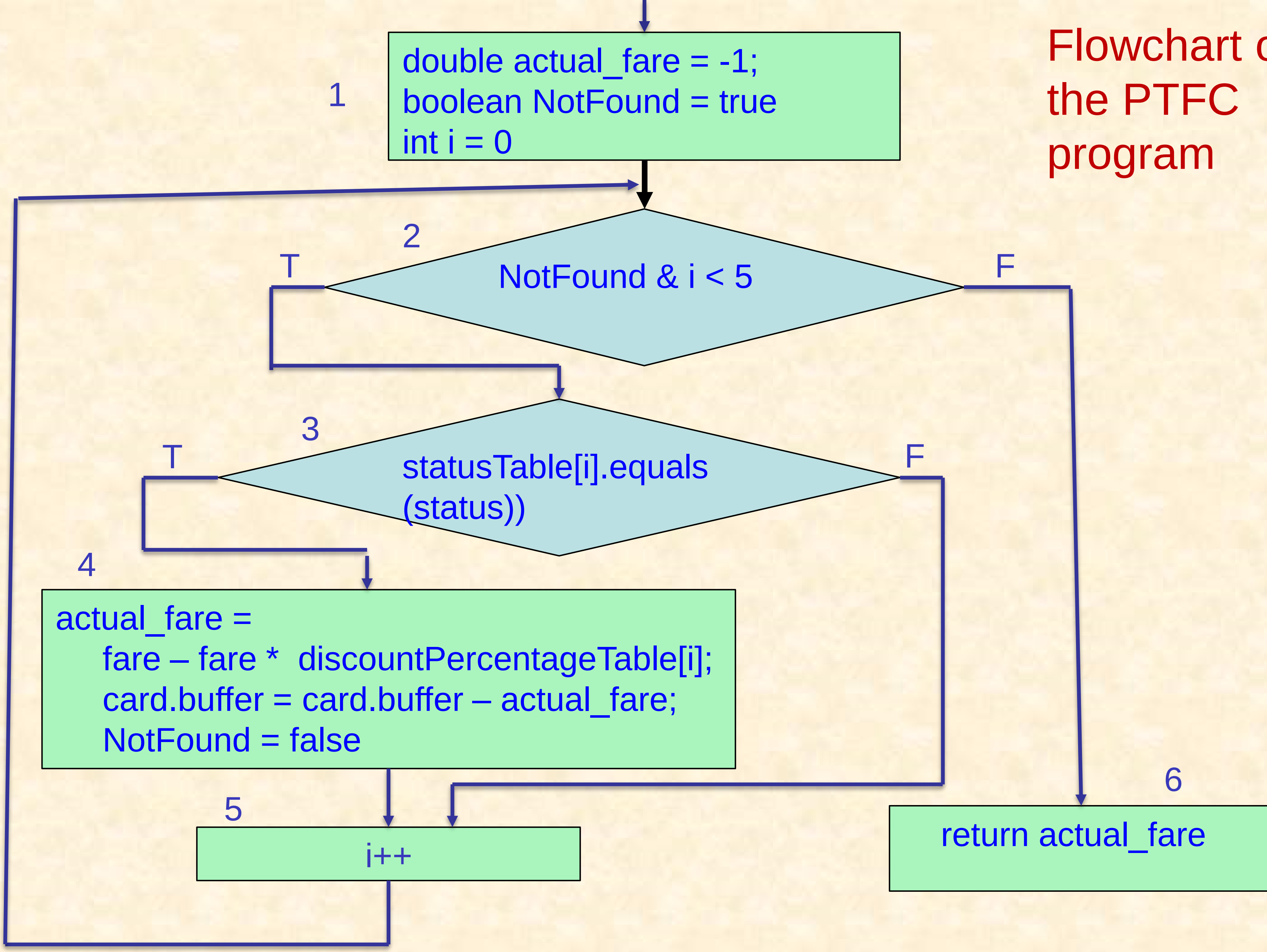
traversed(p, tc)

Example

```
public double Tell_Raiway_Fare(String status, double fare) {
double actual_fare = -1;
boolean NotFound = true;

for(int i = 0; NotFound & i < 5; i++) {
  if(statusTable[i].equals(status)) {
   actual_fare =
    fare – fare *  discountPercentageTable[i];
   card.buffer = card.buffer – actual_fare;
   NotFound = false;
}
}
 return actual_fare;
}
```

Flowchart of the PTFC program



Representative paths (Branch sequences)

We have the following representative paths:

p1: [1, 2, 3, 4, 5, -2, 6]

p2: [1, 2, -3, 5, …, -2, 6]

p3: [1, -2, 6] (infeasible)

Example of test cases satisfying the Criterion

tc	status	fare	~card. buffer	actual_ fare	card. buffer	coverage
1	380	“infant”	1500	0	1500	S1 & p1
2	1200	“student”	2300	600	1700	S2 & p2
3	530	“normal”	3800	530	3270	S3 & p2
4	960	“pensioner”	4300	672	3128	S4 & p2
5	130	“disable”	4100	91	4009	S5 & p2
6	240	“superman”	5205	-1	5205	S6 & p2
7	1500	“anything”	1200	-1	1200	¬ pre- & p2

Test oracle generation for test result analysis

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

$$\exists_{tc \in T} \cdot S_{pre}(tc) \wedge G(tc) \wedge \neg D(tc, P(tc))$$

holds, it indicates that a bug in program P is found by tc (also by T).

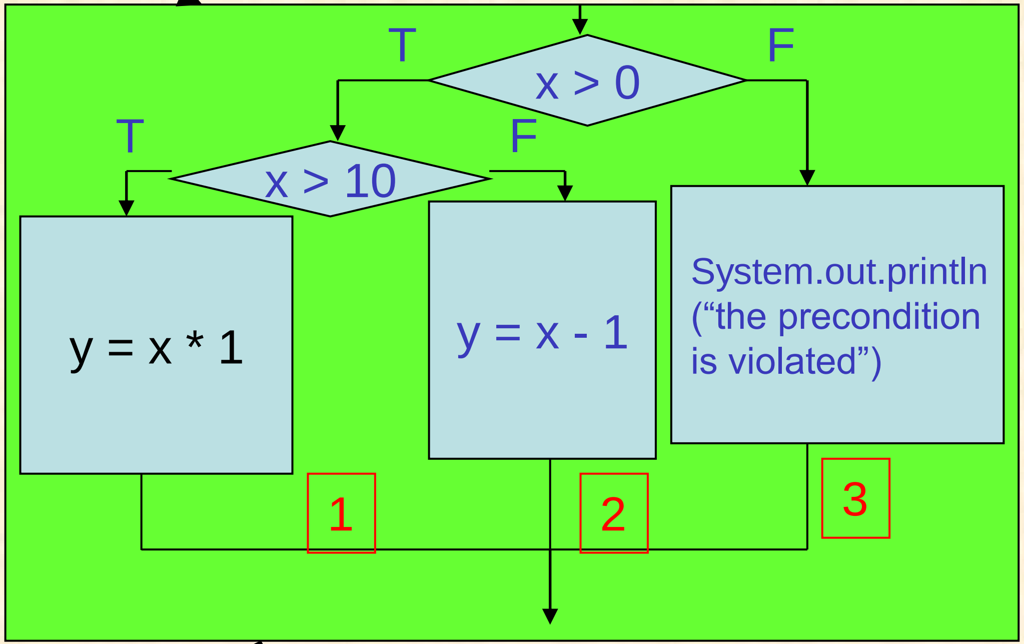
Test case generation

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	$A_{pre} \wedge G(tc)$	D(tc)	$A_{pre} \wedge G(tc) \wedge \neg D(tc)$
15	15	true	false	true
5	4	true	true	false

Algorithms for automatic test data generation

We need to provide algorithms for test case generation from (1) atomic predicates, (2) conjunctions, and (3) disjunctions, respectively.

The algorithms should also be able to deal with both numeric values and compound values, such as sets, sequences, composite objects, and maps.

Algorithms for generating test data from atomic predicates

Atomic predicate: $Q(x_1, x_2, \ldots, x_q)$

Relational operator: $\ominus \in \{=, >, <, >=, <=, <>\}$

Format of the atomic predicate:

(1) $x \ominus E$, where E is a constant

(2) $E_1 \ominus E_2$, where E_1 and E_2 involves only variable x_1.

(3) $E_1 \ominus E_2$, where E_1 and E_2 may involve x_1, x_2, …, x_q.

(1) Algorithms for atomic predicates: $x \ominus E$

Algorithm No.	Relational operator	Algorithm of generating a value for x_1	Algorithm of generating a value for the remaining variables x_i ($i = 2, \ldots, q$)
1	=	$x_1 := E$	$x_i :=$ any $\in$ Type(x_i)
2	>, >= , <>	$x_1 := E + \delta$	$x_i :=$ any $\in$ Type(x_i)
3	<, <=	$x_1 := E - \delta$	$x_i :=$ any $\in$ Type(x_i)

where $\delta > 0$

(2) Algorithm for generating test data from atomic predicate: $(E_1 \ominus E_2)(x_1)$

Step 1: Transform $(E_1 \ominus E_2)(x_1)$ into the format $x_1 = E$, where E is a constant.

Step 2: Apply the corresponding algorithm for generating test data from $x_1 \ominus E$ given previously.

(3) Algorithm for generating test data from atomic predicate: $(E_1 \ominus E_2)(x_1, x_2, \ldots, x_q)$

Step 1: Transform $(E_1 \ominus E_2)(x_1, x_2, \ldots, x_q)$ into the format $(E_1 \ominus E_2)(x_1)$ by first randomly generating test data for each of $x_2, \ldots, x_q$.

Step 2: Apply the corresponding algorithm for generating test data from $(E_1 \ominus E_2)(x_1)$ given previously.

Algorithms for generate test data from atomic predicates involving variables of compound data types

Compound types:

Set types

Sequence types

Composite types

Algorithms for generating test data from atomic predicates involving variables of the above compound types can be found in our paper.

Algorithms for generate test data from a conjunction

Let Q be a conjunction of predicates:

$Q_1 \wedge Q_2 \wedge \cdots \wedge Q_n$

(1) A primitive algorithm (PA) for generating test data from Q:

Step 1: Generate a test data tc satisfying Q_1

Step 2: Check whether tc satisfies the remaining predicates $Q_2, \ldots, Q_n$. If yes, then tc will be treated as a qualified test data. Otherwise, replace Q_1 with another predicate to repeat the two steps until a qualified test data is found or a termination condition is met.

The problem with PA

Example:

$x + y > 10 \wedge x > 5 \wedge y < 7$

If we start generating test data from the first atomic predicate $x + y > 10$, then it might fail to generate a qualified test data:

Let $x = 4$, $y = 7$. Then, this test data will fail to satisfy $x > 5$ and $y < 7$.

If we start with $x > 5$ and $y < 7$. Then, the success of generating a qualified test data will be higher.

Let $x = 6$, $y = 6$. Then, it satisfies all predicates.

(2) A more efficient algorithm (EA):

Definition (predicate dependency). Let E_1 and E_2 be two predicate expressions. If $Var(E_1) \subset Var(E_2)$, E_2 is said to be dependent on E_1, which is represented as $E_1 \sqsubset E_2$.

$Var(E)$ denotes the set of all free variables occurring in expression E.

For example, $Var(x * y > 20) = \{x, y\}$.

For example, predicate $x * y > 20$ is dependent on $x > 0$; that is, $x > 0 \sqsubset x * y > 20$

Definition (ordered partition). Let $\{R_1, R_2, \ldots, R_u\}$ be a set of predicate sets. If it satisfies the following two conditions:

(1) $\forall i \in [1..u-1] \forall E_1 \in R_i \exists E_2 \in R_{i+1} \cdot R_i \sqsubset R_{i+1}$

(2) $\forall i \in [1..u-1] \forall E_1, E_2 \in R_i \cdot \neg(E_1 \sqsubset E_2)$

$\{R_1, R_2, \ldots, R_u\}$ is said to be an ordered partition on $\sqsubset$.

Algorithm EA

No.1. Construct an ordered partition $\{R_1, R_2, \ldots, R_u\}$ for $\{Q_1, Q_2, \cdots, Q_n\}$.

No.2. $t_0 := \{\}$; $i := 1$; $flag := 0$; /*initializing

t_0 representing the generated test data*/

No.3. while ($i \le u$ & $flag \le$ NoOfFailure) {

A:= ObtainInstantiatedPredicates(R_i, t_{i-1});

/*A is an array of predicates*/

t_i := GenerateTestData(A); /*t_i is a new test data (possibly incomplete) generated based on the predicates in A*/

```
if (ti == {}){
  if (i > 1) {
    i := i -1; }
 flag := flag + 1;}
else {
  i := i + 1;}
 } //while loop ends
```

No.4. if (flag > NoOfFailure) {

Display a test data generation failure message}

else {

Display a test data generation success message and t_i is treated as the test data}

No.5. End.

Algorithm for generating test data from disjunctions

Let $P_1 \vee P_2 \vee \cdots \vee P_m$ be a disjunction of predicate expressions. Let $T(P_i)$ denote the test set generated from P_i. Then, an algorithm for generating a test set from the disjunction is as follows:

$$T(P_1 \vee P_2 \vee \cdots \vee P_m) =$$

$$T(P_1) \cup T(P_2) \cup \cdots \cup T(P_m)$$

Challenge

A challenge is how to automatically generate test cases from the specification so that all the representative paths of the program can be traversed at least once.

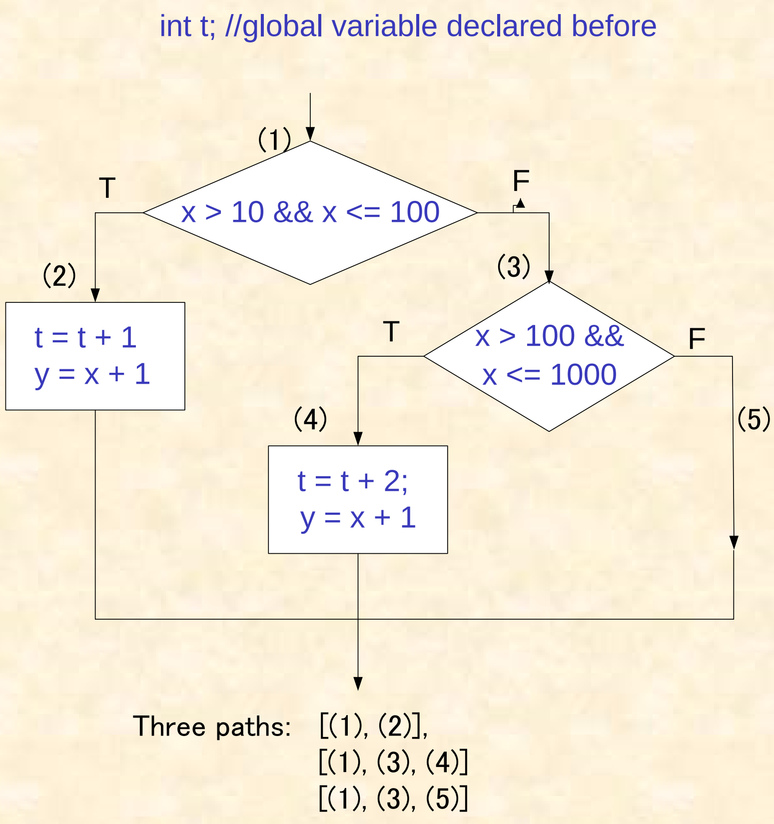
The reason why we face this challenge is that each functional scenario in the specification is usually refined into many paths in the code and it is extremely difficult to establish a theory that tells how test cases generated only from the specification can ensure that all the paths can be traversed.

Example: the functional scenario:

$x > 10 \land x <= 1000 \land$
$y = x+1$

x is input

y is output



A "Vibration" method (V-Method) for test set generation

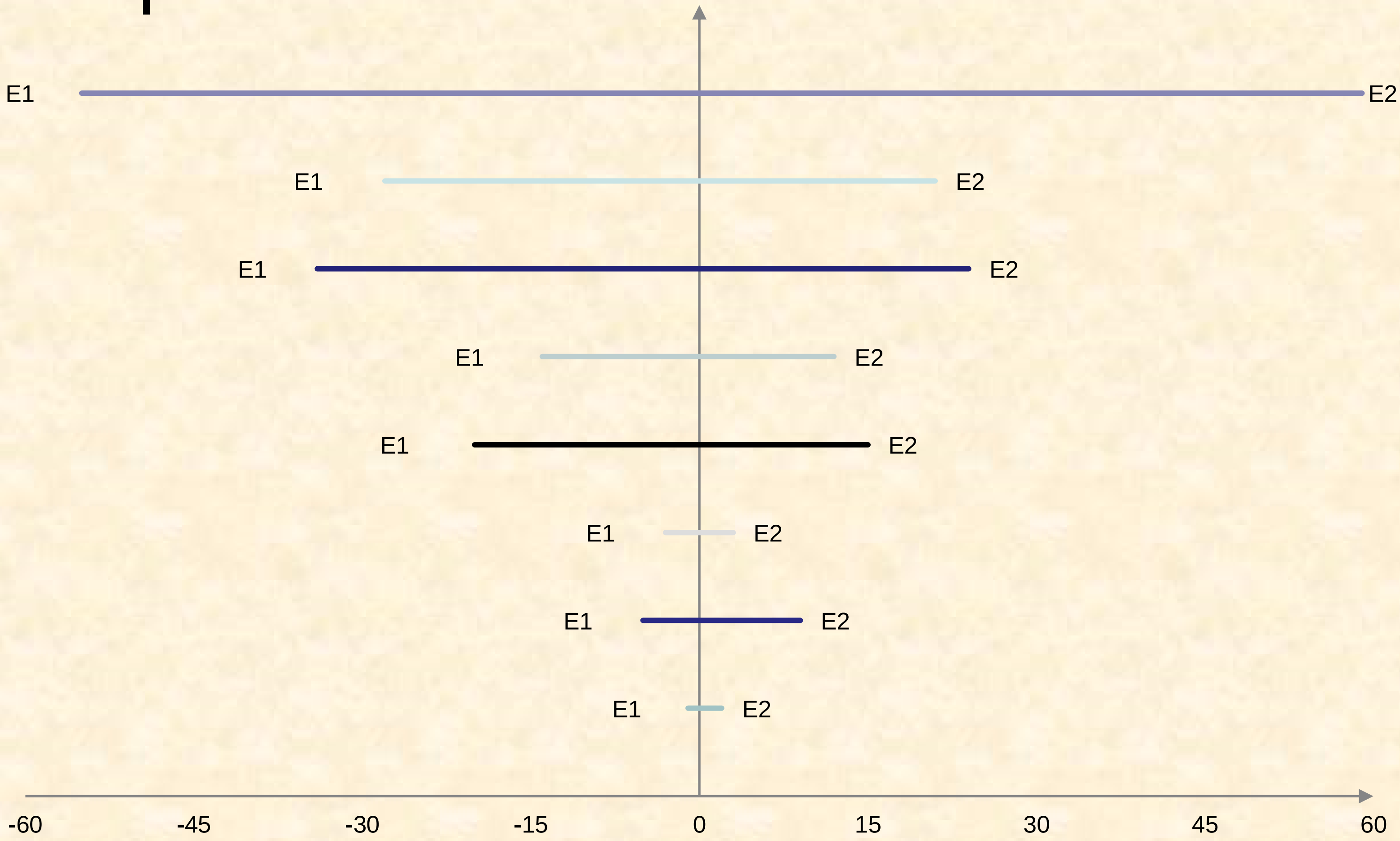
Let $E_1(x_1, x_2, ..., x_n)$ R $E_2(x_1, x_2, ..., x_n)$ denote that expressions E_1 and E_2 have relation R, where $x_1, x_2, ..., x_n$ are all input variables involved in these expressions.

Question: how test cases can be generated based on the relation so that they can quickly cover all the paths refining the functional scenario involving the relation in the specification?

V-Method:

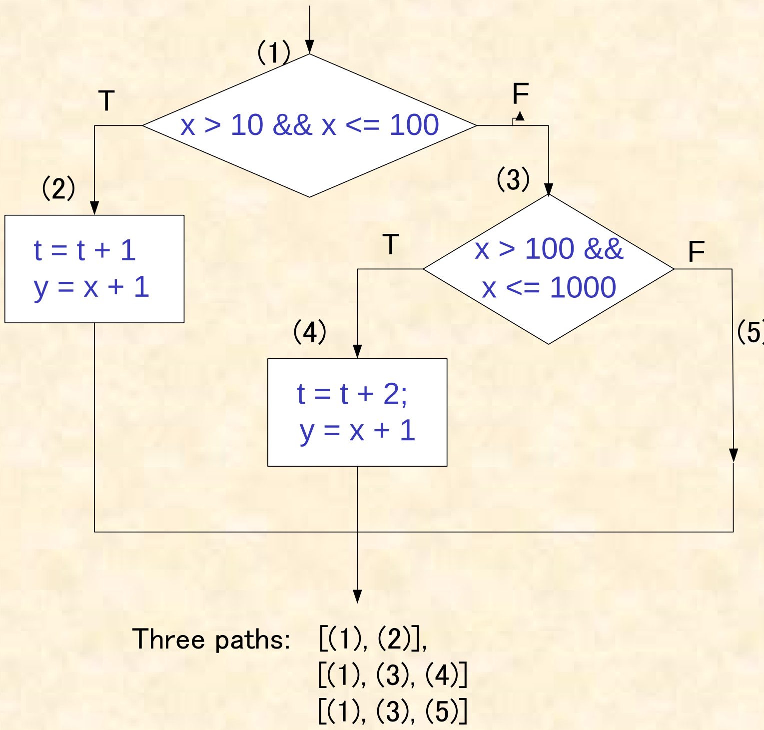
We first produce values for $x_1, x_2, \ldots, x_n$ such that the relation $E_1(x_1, x_2, \ldots, x_n)\ R\ E_2(x_1, x_2, \ldots, x_n)$ holds with an initial "distance" between E_1 and E_2, and then repeatedly create more values for the variables such that the relation still holds but the “distance” between E_1 and E_2 “vibrates” (changes repeatedly) between the initial “distance” and the maximum “distance”.

Example: E1 > E2



Example

int t; //global variable declared before



Example: the functional scenario:

x >10 ∧ x <=1000 ∧
y = x+1

x is input

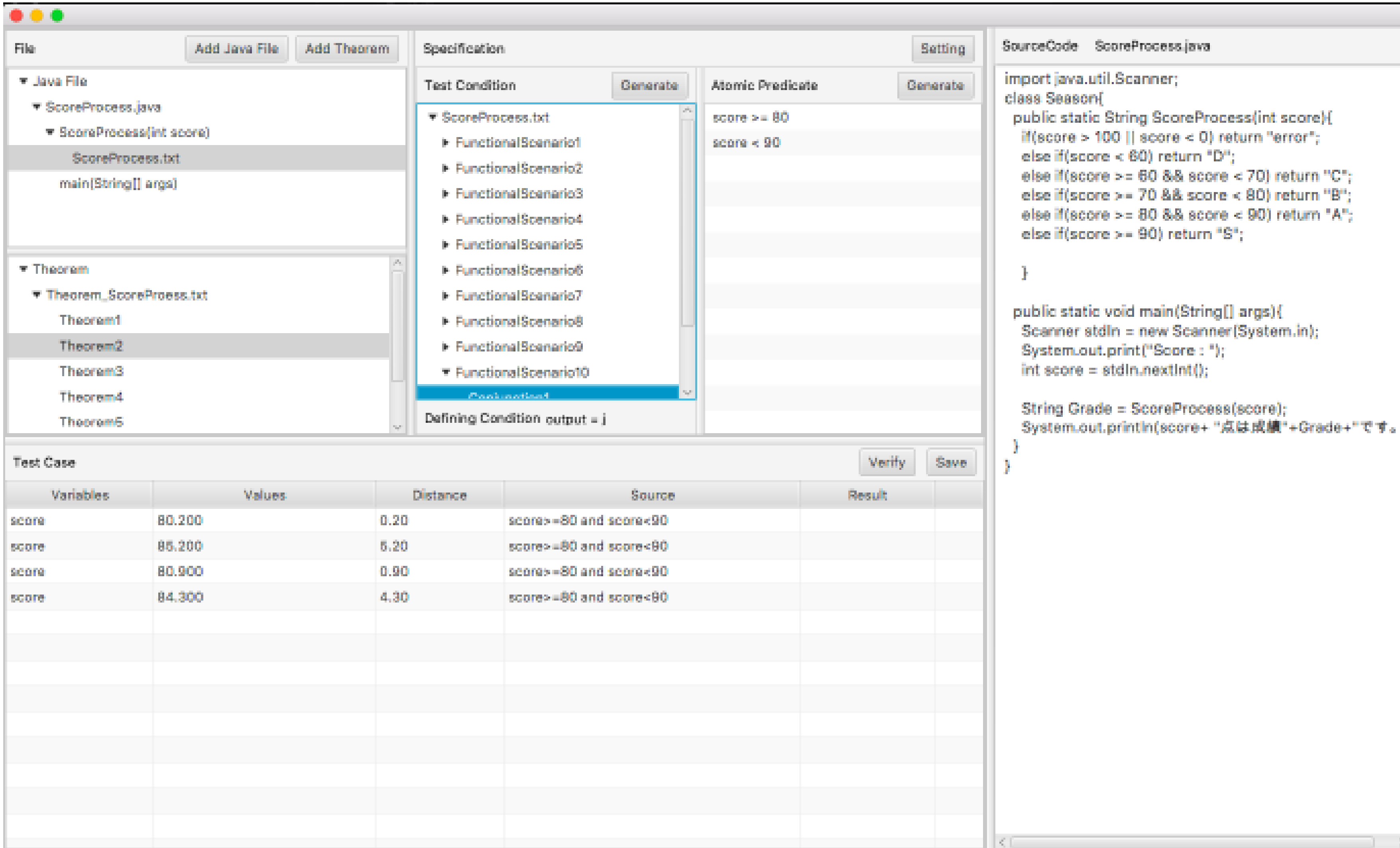
y is output

No.1. Let d (distance) = 8. Generate x > 10 + 8 = 19. This test traverses the path: [(1), (2)].

No.2. Let d = 100. Generate x > 10 + 100 = 111. It traverses the path: [(1), (3), (4)]

No.3. Let d = 200. Generate x > 10 + 200 = 211. It traverses the same path: [(1), (3), (4)]

Prototype tools for V-Method



K. Saiki, S. Liu, H. Okamura, T. Dohi, “A Tool to Support Vibration Testing Method for Automatic Test Case Generation and Test Result Analysis”, The 21st IEEE International Conference on Software Quality, Reliability, and Security (QRS 2021), IEEE CPS, pp. 149-156, Dec. 6-10, 2021, doi: 10.1109/QRS54544.2021.00026.

Conclusions

- The V-Method is characterized by using functional scenarios as the foundation for test case and test oracle generation and using “vibration” step to gain more path coverage in the program. It is unique among the existing specification-based testing techniques.
- Automatic testing based on specifications is an efficient way to reduce cost and avoid mistakes in the generation of test cases and test oracles, but the specification must be written in a formal notation.
- The capability of our V-Method indicates the value of writing a formal specification properly for software projects.

Future work

- Try to establish a theory to improve our V-Method to ensure that both functional scenarios in the specification and the corresponding paths in the program can be efficiently covered by the generated test cases (e.g., by utilizing genetic algorithms and/or symbolic execution)
- Continue to improve the prototype tool for the V-Method to support test case generation from more complex data structures.
- Integrate the V-Method with Hoare logic to support testing-based formal verification(TBFV)

The end!

Thank you!