Software Fault Prevention and Verification in Human-Machine Pair Programming

Shaoying Liu

Graduate School of Advanced Science and Engineering &

School of Informatics and Data Science

Hiroshima University, Japan

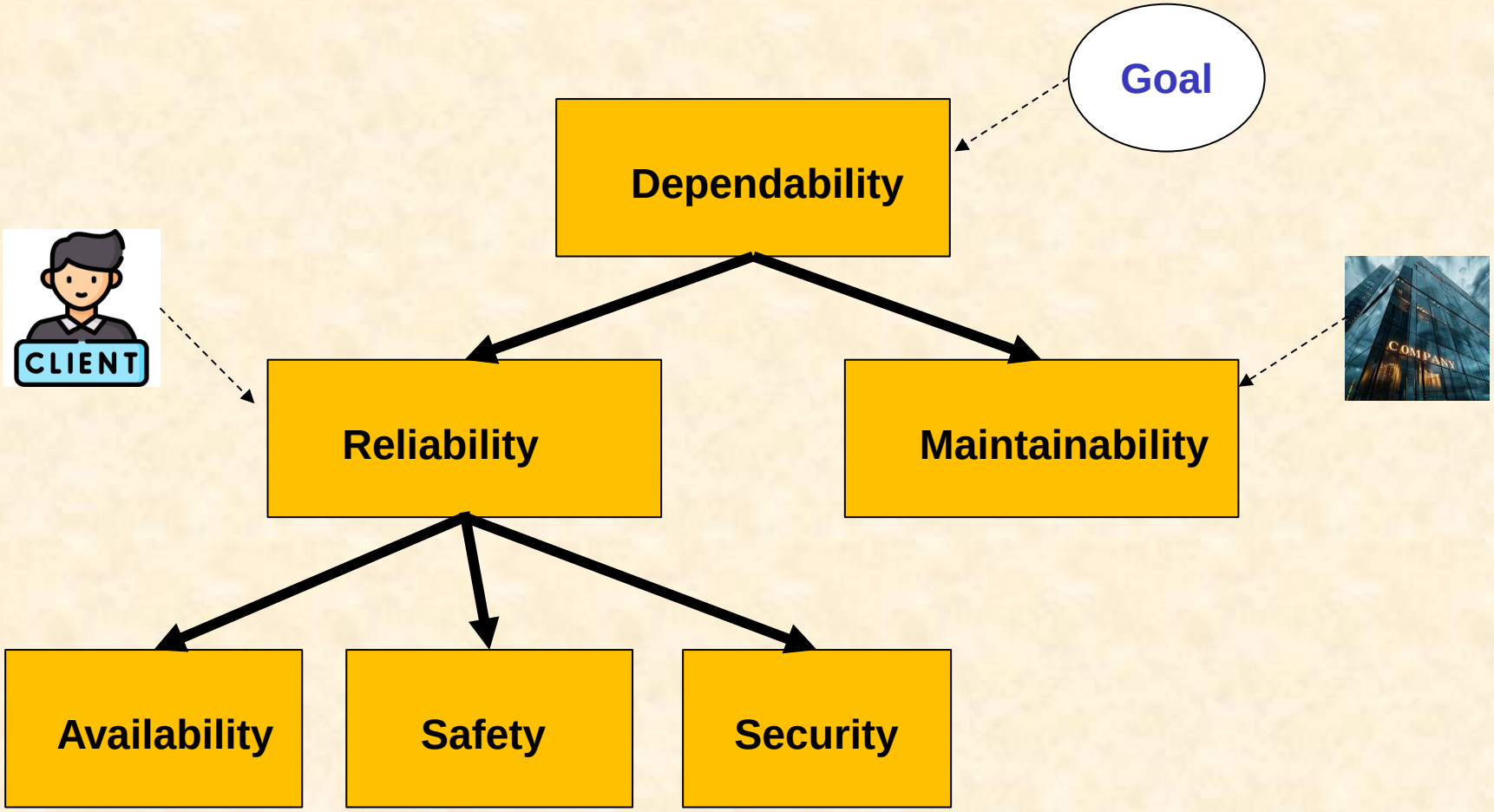
Email: sliu@Hiroshima-u.ac.jp

HP: https://home.Hiroshima-u.ac.jp/sliu/

Overview

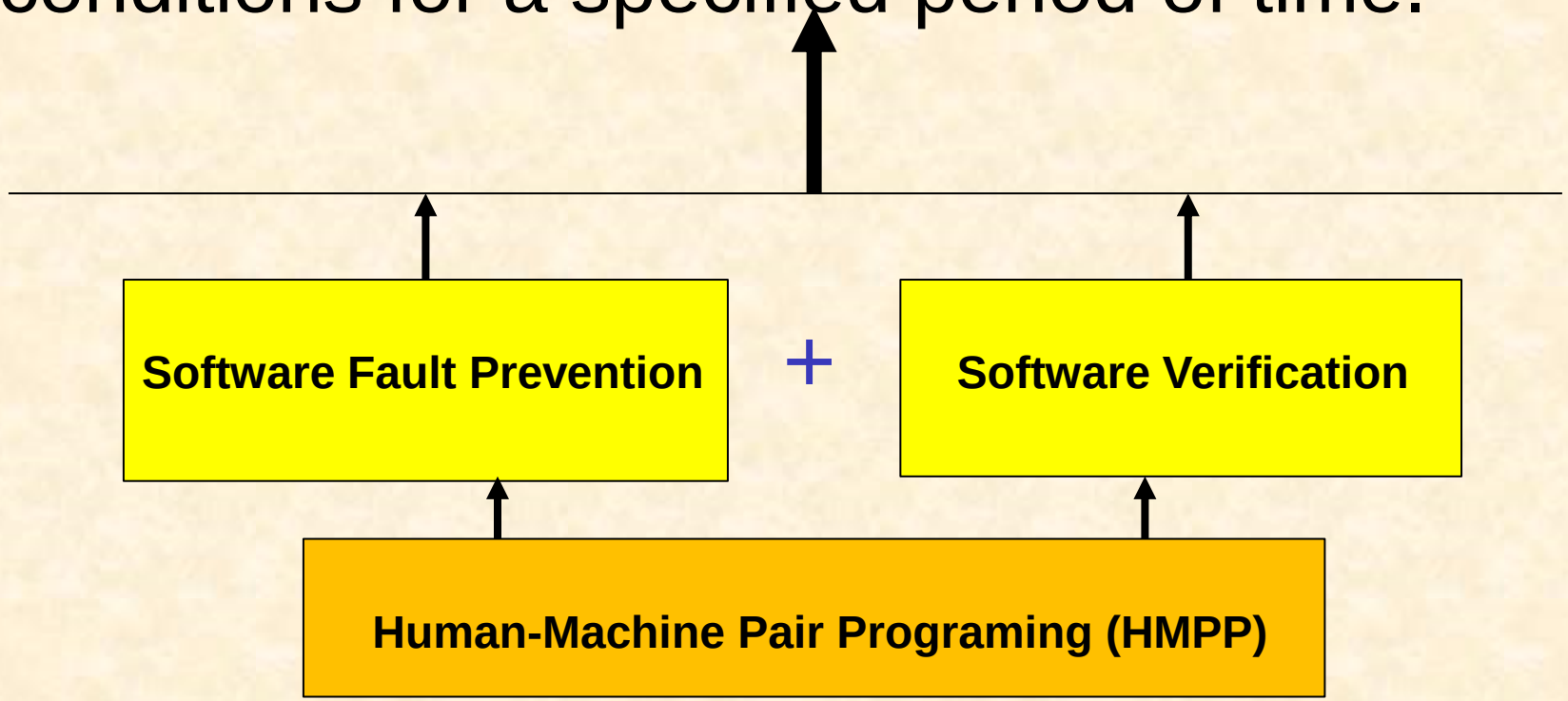
1. Important Goal in Developing Dependable Systems
2. Human-Machine Pair Programming (HMPP)
3. Fault Prevention in HMPP
4. Program Verification in HMPP
5. Conclusions
6. Future work

1. Important Goal in Developing Dependable Systems



Question: how to ensure reliability?

Definition (reliability): Software reliability is defined as the probability that a software system will function as required under given conditions for a specified period of time.



2. Human-Machine Pair Programming (HMPP)?

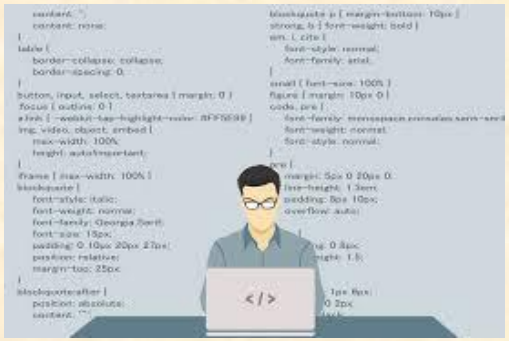
Definition: **HMPP** means that a programmer and computer work together to construct a program, where the programmer acts as a *driver* and the computer acts as an *observer*.

The *driver* takes care of creation of the program while the *observer* takes care of fault prevention and program verification.

program = specification or

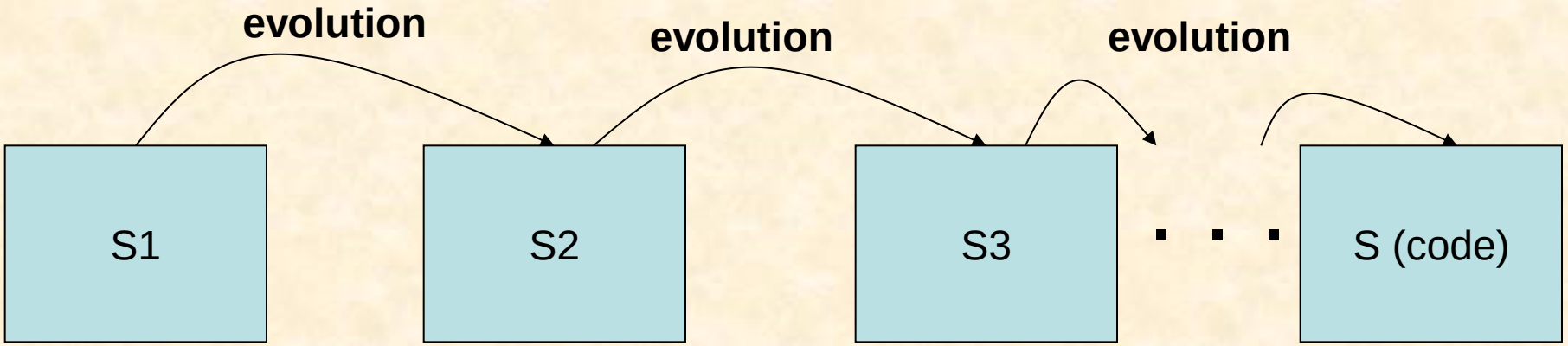
code or

combination of both



An Evolutionary View of Programming

The research on HMPP focuses on the observer's role of computer and on dealing with the problem of how to construct a correct program S through a series of evolutions of partial programs:



$$S1 \ll S2 \ll \cdots \ll Sn = S$$

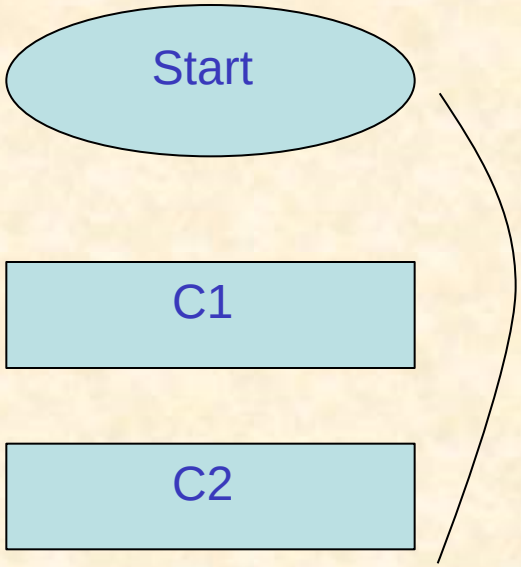
$S1 \ll S2$ means that S1 is evolved to S2, or S2 is an evolution of S1.

Program Evolution

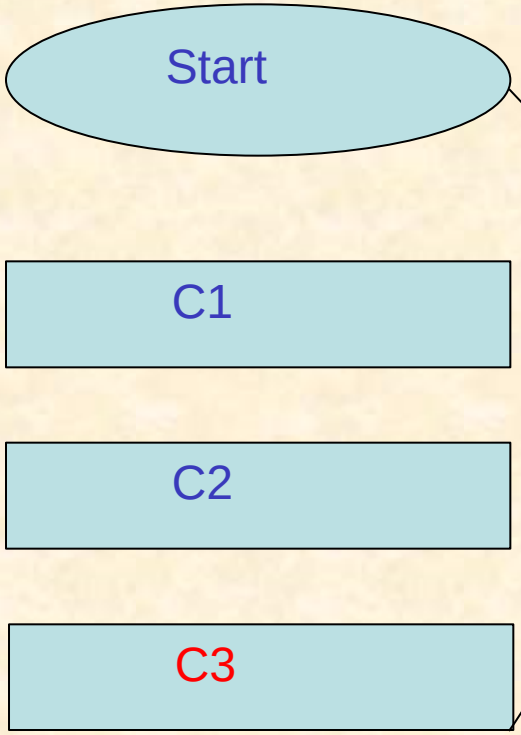
Definition: A partial program S is a sequence of commands, denoted by S = [C1, C2, …, Cm], where each Ci (i = 1, …, m) is a command (a specification or code).

Definition: Let S1 = [C1, C2, …, Cm] and S2 = [C1', C2', …, Cn']. Then, S1 is extended to S2, denoted by S1 $\leqq$ S2, if and only if they satisfy the condition:

Ci = Ci' (for i = 1, …, m) and n > m



Current partial program



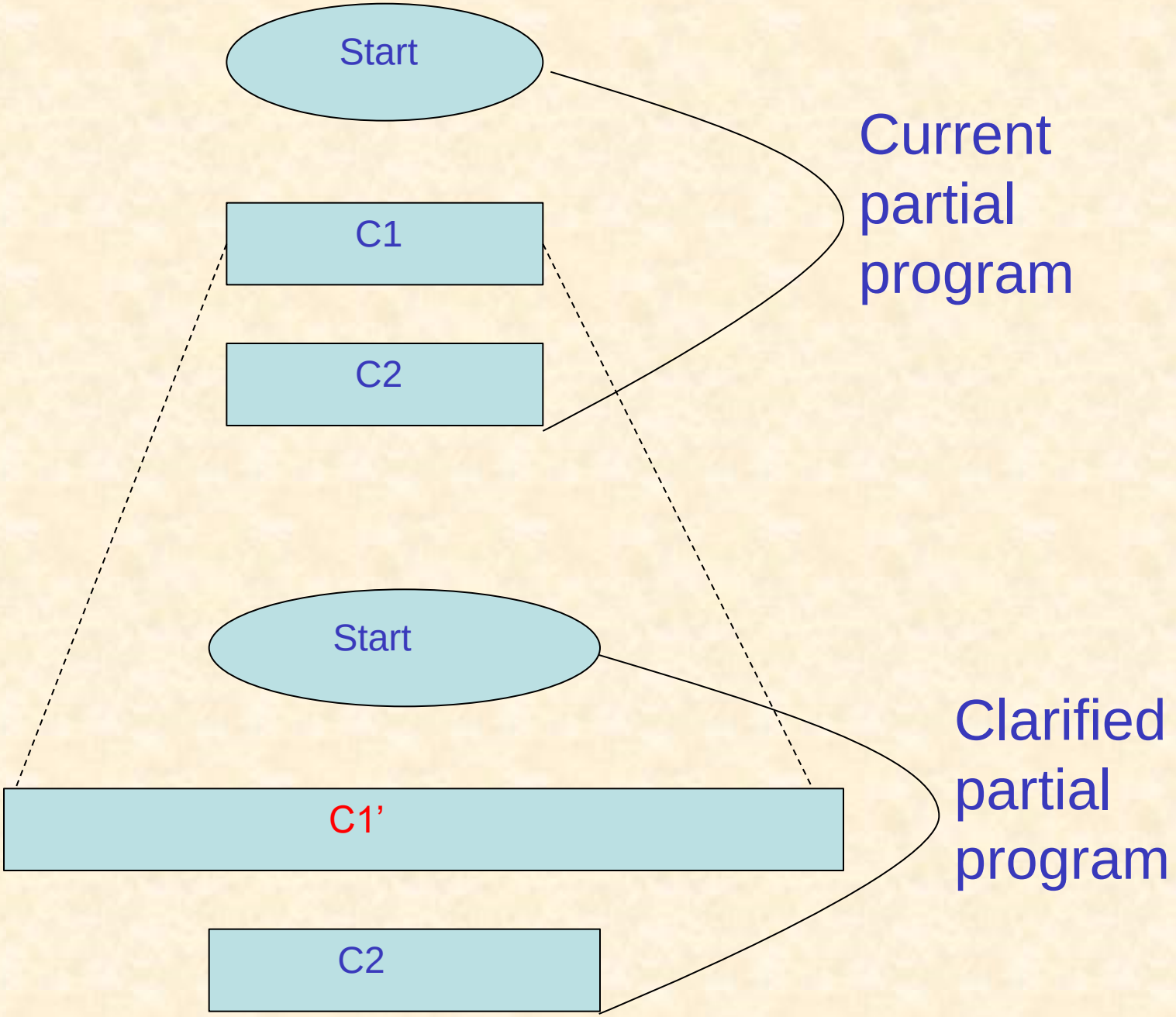
Extended partial program

Definition: Let S1 = [C1, C2, …, Cm] and S2 = [C1', C2', …, Cn']. Then, S1 is clarified to S2, denoted by S1 ⊏ S2, if and only if they satisfy the condition:

n = m and for some Ci (1 <= i <= m), Ci is redescribed more clearly by Ci'.

In this case, we also say S2 is a clarification of S1.

The clarification also defines the human's responsibility.


Start
C1
C2
Current partial program
Start
C1'
C2
Clarified partial program

Definition: Let S1 and S2 be two partial programs. Then, S1 is evolved to S2, denoted by $S1 \ll S2$, if and only if they satisfy the condition:

S1 is extended to S2 ($S1 \leqq S2$) or

S1 is clarified to S2 ($S1 \sqsubset S2$).

Questions?

- How to prevent faults during program construction?
- How to verify the correctness of programs after they are completed?

3. Fault Prevention

What is fault prevention?

Fault prevention refers to the measures and techniques for stopping faults from being introduced into the finally implemented code.

Where can the fault prevention be done?

- Requirements analysis phase (for faults related to requirements)
- Design phase (for faults related to architecture, data structure, and algorithm)
- Implementation phase (for faults related to the use of programming languages)

Techniques for fault prevention

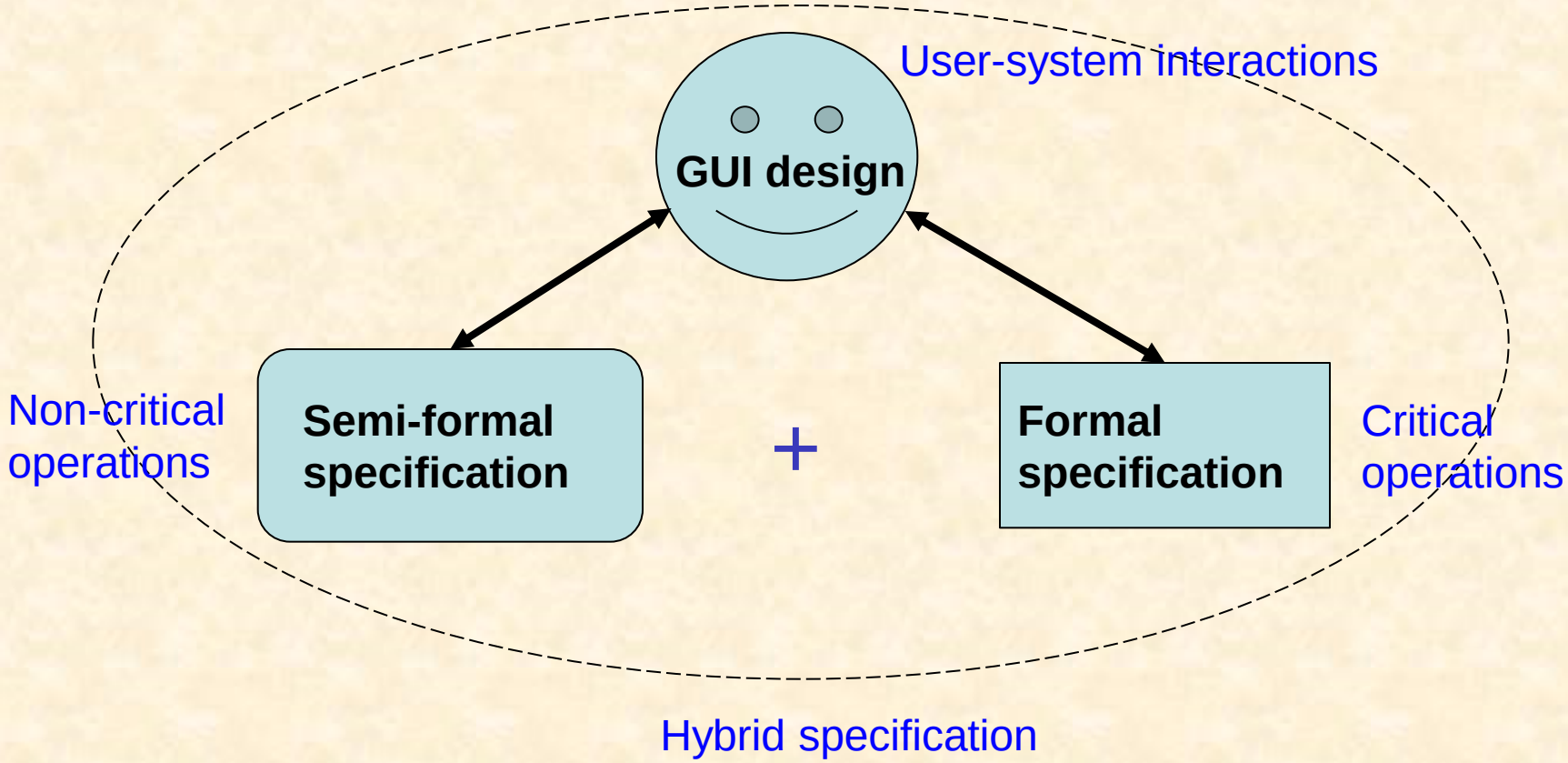
- Using a systematic evolutionary process to construct a precise **hybrid specification** to prevent **requirement-related and design-related faults**.

- Using various techniques to prevent **implementation-related faults** during program construction.

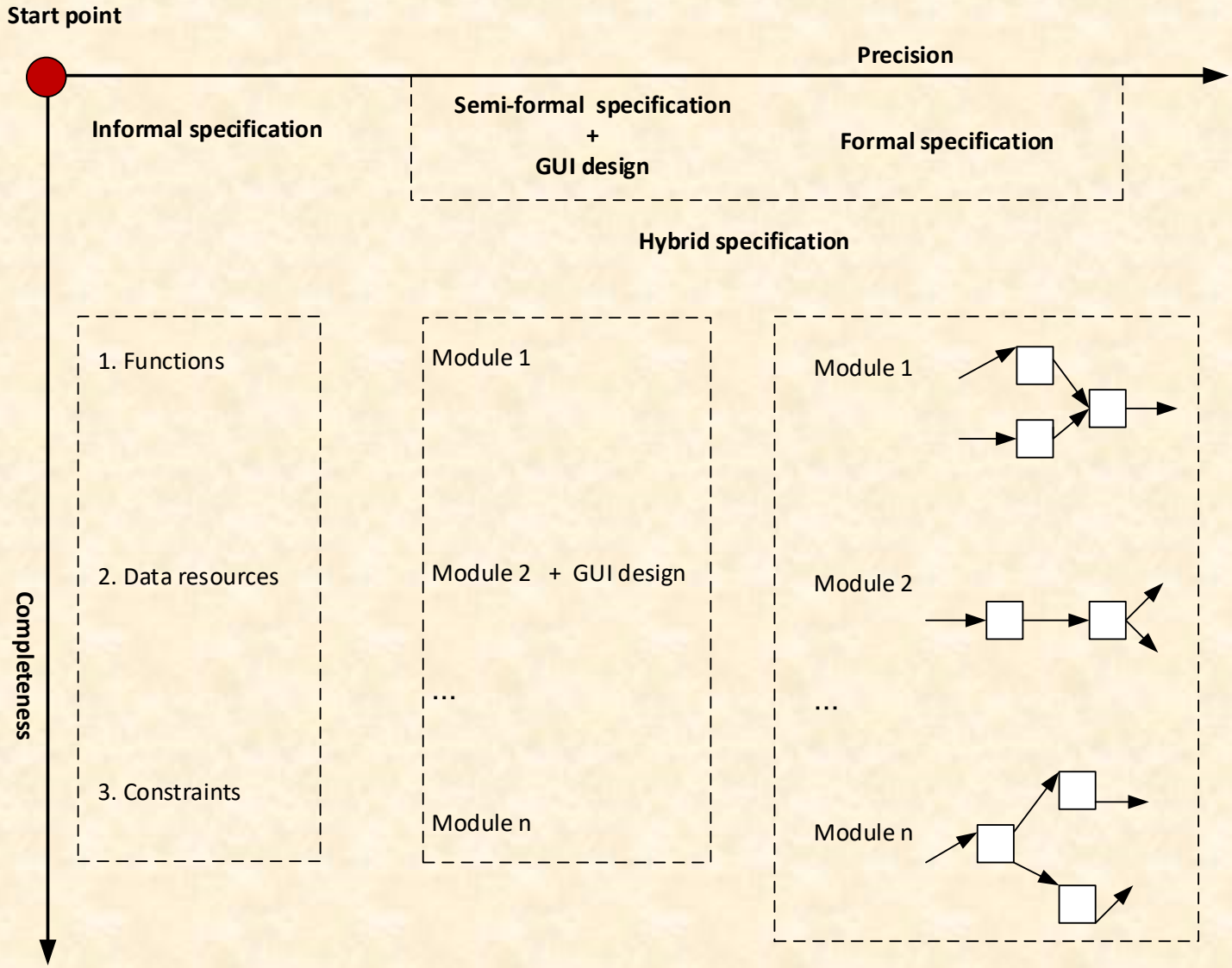
- Using **testing-based formal verification (TBFV)** to verify the completed programs.

3.1 Hybrid specification

A hybrid specification is a combination of semi-formal specification, GUI design (specification), and formal specification.



3.2 A three-step hybrid specification approach (TSHSA)



Informal specification:

Informal Specification of the Universal Card Railway Services System

1 Functions

1.1 Make a new UC card.
1.2 Cancel a UC card.
1.3 Charge the UC card using cash.
1.4 Cancel the charge of the UC card and return the cash.
1.5 Charge the UC card from the related bank account.
1.6 Cancel the charge and return the money to the related bank account.
1.7 Use the UC card to buy a ticket.
1.8 Cancel the ticket and return the money to the UC card.
1.9 Use the UC card to buy a fixed-period commute pass with the choice of one-month, three-month, and six-month.
1.10 Cancel the bought fixed-period commute pass
1.11 Use the UC card as a railway pass to go through the control of entrance of a railway station.
  1.11.1 Check the UC card to see whether it can be used.
  1.11.2 Use the UC card as a fixed-period railway pass.
  1.11.3 Use the UC card as a normal non fixed-period railway pass.
  1.11.4 Use the UC card as the combination of the fixed-period and non fixed-period railway pass.
1.12 Use the UC card to go out of the railway station.

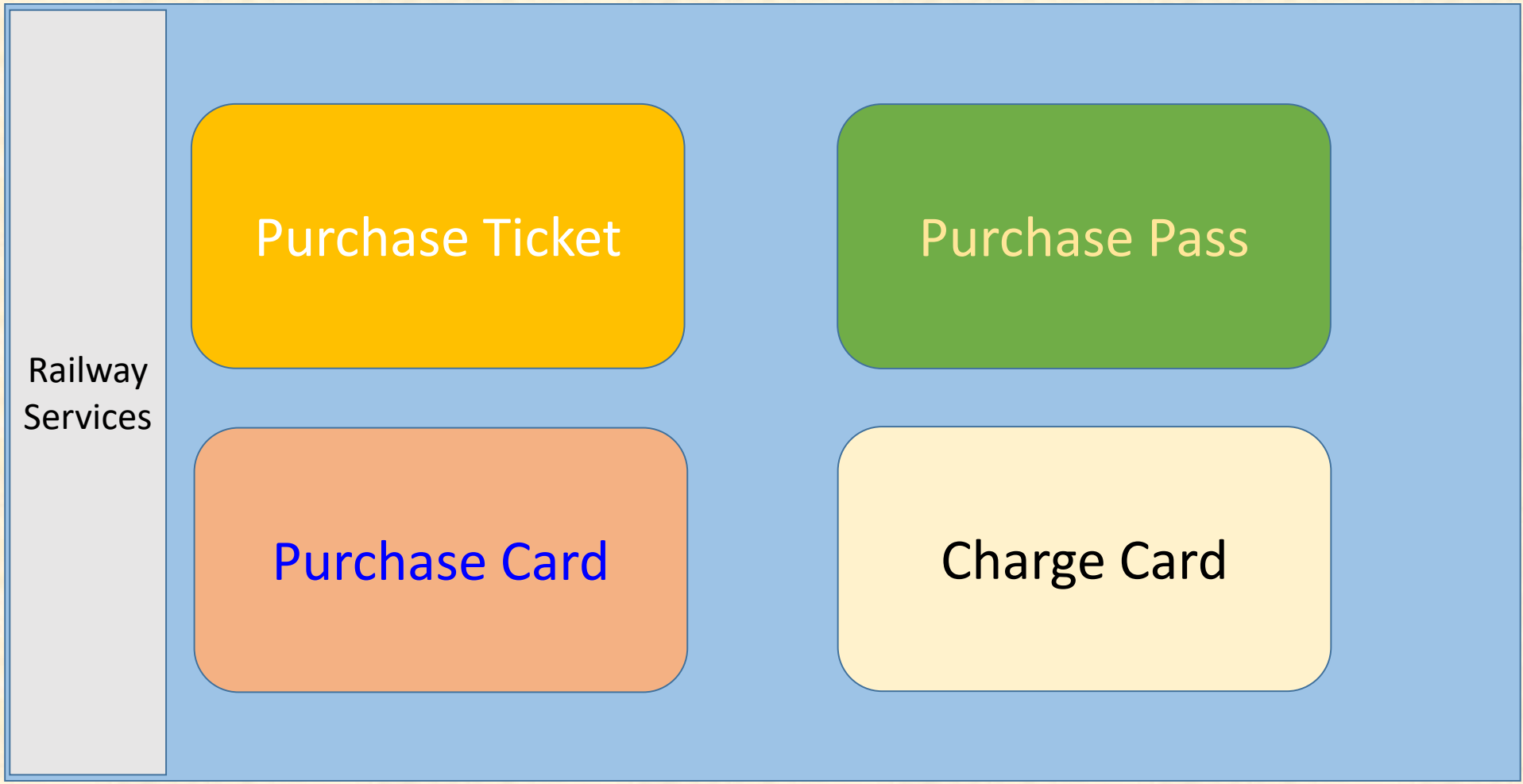
2 Data Resources

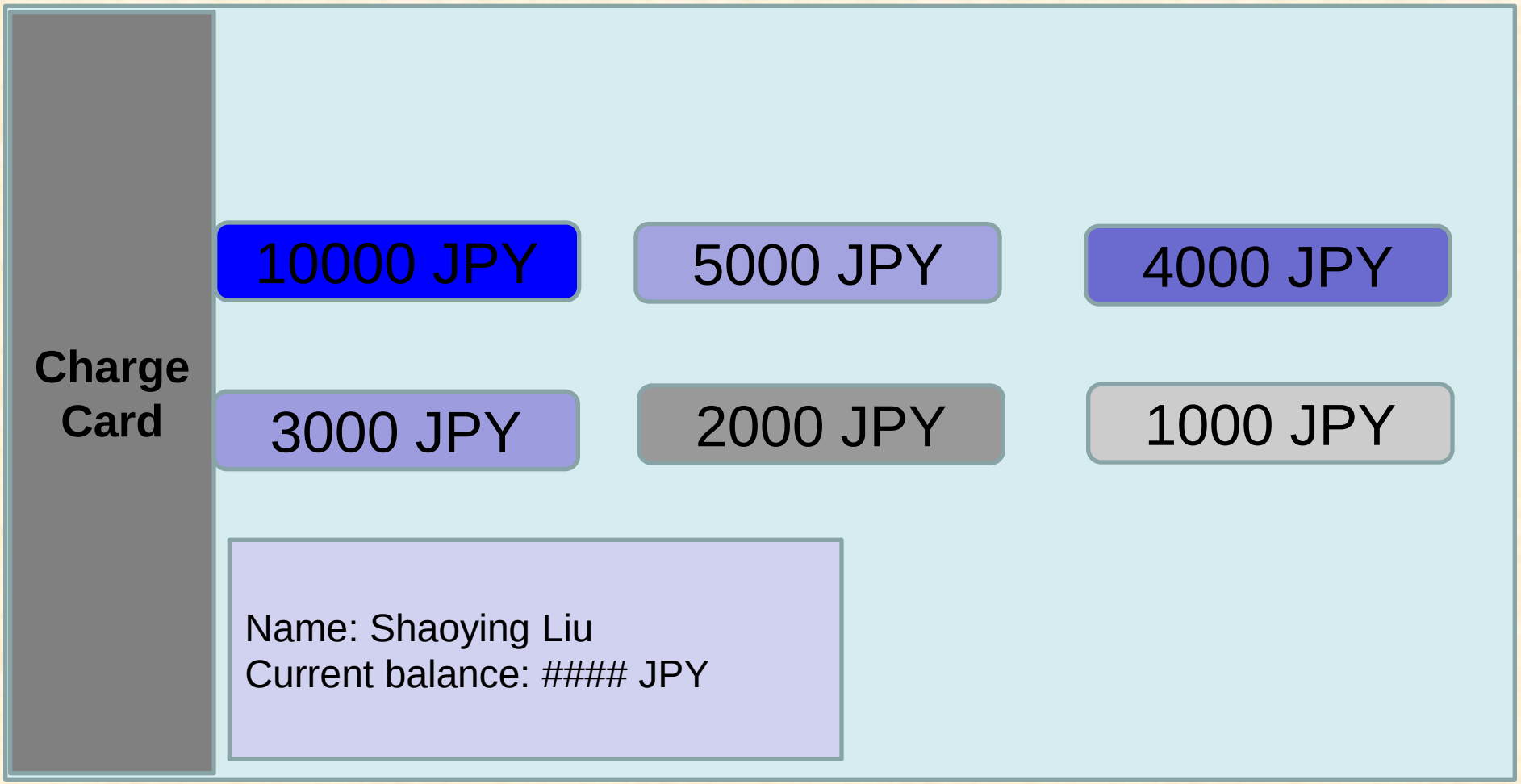
2.1 Personal information (F1.1)
2.2 UC card (F1.1 to F1.12)
2.3 Station price table (F1.5, F1.12)
2.4 UC card ID table (F1.3 to F1.12)
2.5 Maximum amount constraint table (F1.3, F1.5)

3 Constraints

3.1 The maximum amount of the UC card buffer is 50,000 JPY (F1.3, F1.5, D2.5).
3.2 The money in the buffer of the UC card must first be used when buying a ticket with cash (F1.7).

GUI design:


Railway Services
Purchase Ticket
Purchase Pass
Purchase Card
Charge Card


Charge Card
10000 JPY
5000 JPY
4000 JPY
3000 JPY
2000 JPY
1000 JPY
Name: Shaoying Liu
Current balance: #### JPY

GUI specification: defining GUI view changes

```
module GUI_Design;
type
TopPage =  {<Purchse Ticket>, <Purchase Pass>, <Purchase Card>, <Charge Card>};

Amount_Items = {<10000 JPY>, <5000 JPY>, <4000 JPY>, <3000 JPY>, <2000 JPY>, <1000 JPY>};

ChargeCardPage = composed of
                        amount_sel: seq of Amount_Items;
                        message: string
                        end;

AnotherPage = given;  /*indicating that AnotherPage is a given type. */

var
UC_card: Universal_Card_Railway_Services.UniversalCard;
          /*using the type defined in the module Universal_Card_Railway_Services*/

inv
 forall[cp: ChargeCarPage] |
    cp.amount_sel = [<10000 JPY>, <5000 JPY>, <4000 JPY>, <3000 JPY>, <2000 JPY>, <1000 JPY>]
  /* The field amount_sel of any sequence in the sequence type ChargeCardPage must always be the
     given fixed sequence. */

process SelectService(top_page: TopPage)
                        charge_amount_sel: ChargeCardPage | another_page: AnotherPage
pre true
post If top_page = <Charge Card>, i.e., the charge card item on the top page of the GUI is selected,
       then the page charge_amount_sel (No. 2) showing the charge amount items will be shown
            with the necessary card information.
      else another page showing the corresponding information will be displayed.
end_process;

process SelectChargeAmount(charge_amount_sel: ChargeCardPage, amount_item: Amount_Items)
                                charge_amount_result: ChargeCardPage
ext wr UC_card
pre true
post If amount_item is one of the items in the sequence charge_amount_sel.amount_sel and
          the addition of the amount_item and the current buffer of the UC_card is not over 50000 JPY
       then the message of the resultant page (No. 3), charge_amount_result, displays the amount of
            the buffer of the UC_card after the charge
       else the message of the resultant page (No. 4), charge_amount_result, displays an error message, indicating
            that the charge is over the limit.
…

end_module;
```

Semi-formal specification:

module Universal_Card_Railway_Services;
type
UniversalCard = composed of
first_name: string
last_name: string
card_id: string
buffer: nat0 /*JPY only*/
ChargeReceipt = composed of
input_amount: nat0
updated_UC_card_buffer: nat0
end;
var
constrained_amounts: map ItemNames to nat0;

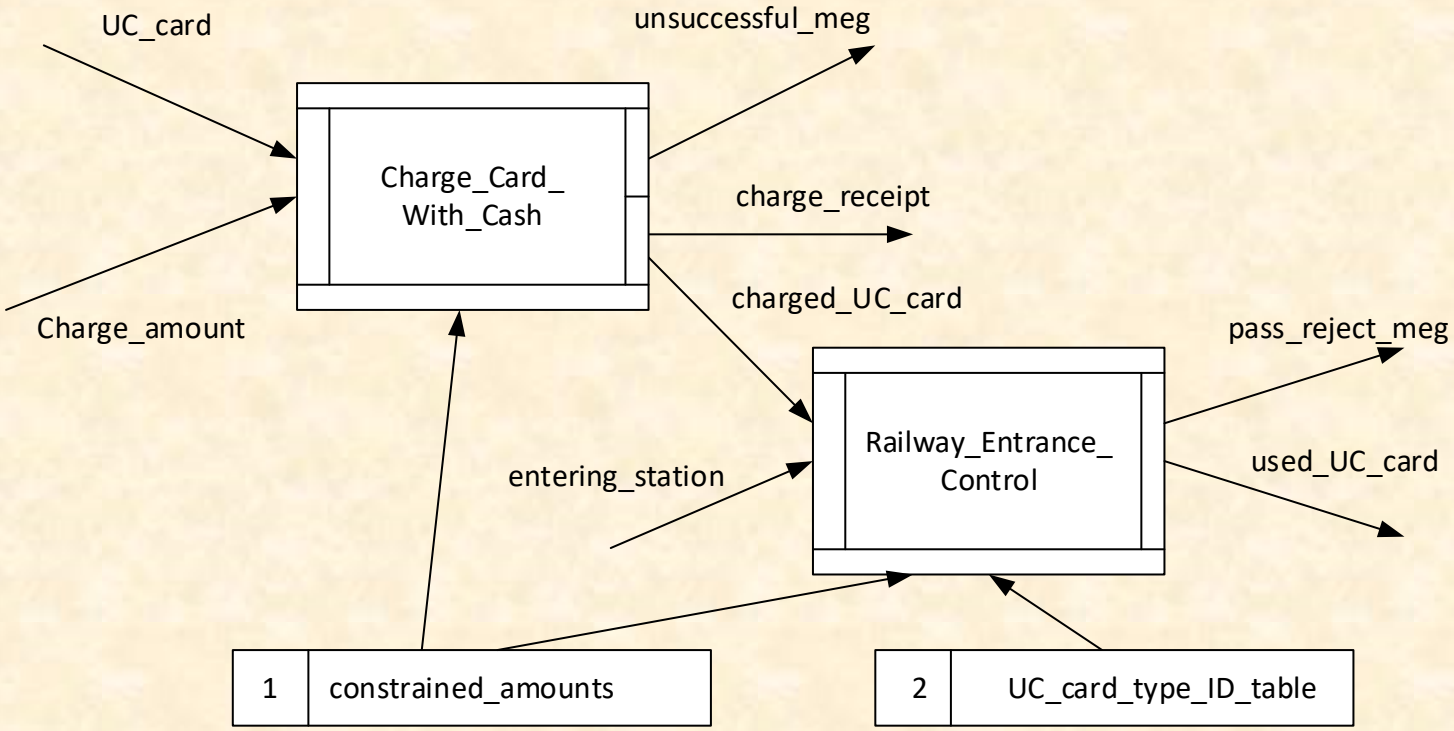
inv
The buffer of every specific card in the type UniversalCard must not exceed 50000 JPY.

process Charge_Card_With_Cash(UC_card: UniversalCard,
charge_amount: nat0)
charged_UC_card: UniversalCard,
charge_receipt: ChargeReceipt |
unsuccessful_meg: string
ext rd constrained_amounts: map ItemNames to nat0
pre true
post if the total amount of *UC_card.buffer* and *charge_amount* is
not over the *UC card* buffer maximum defined in the *constrained_amounts*
then the buffer of the *UC_card* will be updated to reflect the increased *charge_amount* and
the *charge_receipt* is issued properly
else the error message represented by the output *unsuccessful_meg* Will be issued.
end_process;

…

end_module;

The corresponding CDFD:



Formal specification:

```
module Universal_Card_Railway_Services;
type
UniversalCard =  composed of
                  first_name: string
                  last_name: string
                  card_id: string
                  buffer: nat0 /*JPY only*/
ChargeReceipt = composed of
                  input_amount: nat0
                  updated_UC_card_buffer: nat0
                 end;
var
   constrained_amounts: map ItemNames to nat0;

inv
  The buffer of every specific card in the type UniversalCard must not exceed 50000 JPY.

process Charge_Card_With_Cash(UC_card: UniversalCard,
                              charge_amount: nat0)
                              charged_UC_card: UniversalCard,
                              charge_receipt: ChargeReceipt |
                              unsuccessful_meg: string
ext rd constrained_amounts: map ItemNames to nat0
pre true
post  UC_card.buffer + charge_amount <= constrained_amounts(<UC card buffer maximum>) and
      charged_UC_card = modify(UC_card, buffer -> UC_card.buffer + charge_amount) and
       charge_receipt = mk_ChargeReceipt(charge_amount, charged_UC_card.buffer)
          or
       not UC_card.buffer + charge_amount <= constrained_amounts(<UC card buffer maximum>) and
       unsuccessful_meg = "The input charge amount makes the card buffer exceed the 50,000 JPY limit."
end_process;

…

end_module;
```

Specific techniques for fault prevention

(1)Achieving “symmetric functions”

If the system has one function performing some task, it must also possess another function performing the “opposite” task.

Examples:

In an information system,

- “registration” versus “cancellation” (e.g., membership management system),
- “withdraw” versus “deposit” (e.g., ATM system)
- “borrow a book” versus “return a book” (e.g., library system)

(2) Achieving “processing functions”

Generally, a software system must possess functions that properly process the created data or information.

Example:

In an information system, if the “registration” function is desired, there must be the functions that process the registered information, such as “modify” and “extend”.

(3) Achieving “function-based data items”

Every function given in the specification must be examined to understand what data items it needs.

Examples:

- The “Registration” function needs the data items: name, address, telephone number.
- The “Withdraw” function needs the data items: withdrawal amount, password.

(4) Achieving “function-related constraints” and “data-related constraints”

Every function and data item is analyzed to learn what potential constraints (possibly related to safety, security, and/or availability) are needed.

Examples:

- For the “Withdraw” function, a possible constraint is the maximum withdrawal amount (e.g., 200,000 JPY each time).
- For the data item “registered_books” in a library system, each book on the list must have a unique identification number.

(5) Obtaining the completeness of process specifications.

Suppose the specification process S is expressed as a functional scenario form (FSF):

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

Then, we must ensure the completeness:

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

(2) $\forall \sigma \in \Sigma \cdot S_{pre}(\sigma) \wedge G_i(\sigma) \Rightarrow \exists \sigma' \in \Sigma \cdot D_i(\sigma, \sigma')$

(6) Achieving the requirements traceability during the evolution of specifications

To help track how the acquired functions, data items, and constraints in the informal specification are evolved in the hybrid specification, a requirement traceability should be established.

Examples:

For each function, data item, and constraint in the informal specification, we build a link connecting it to the corresponding representation in the hybrid specification.

3.3 Techniques for implementation-related faults prevention

Implementation-related faults refer to all the exceptions defined by the exception hierarchy in Java (for example).

The techniques for fault prevention:

(1) Pattern-based static analysis method

(2) Program segment testing

(1) Pattern-based static analysis method

Step 1: Construct a fault pattern base

Fault pattern = {Faulty Program fragment;
Condition for making faults;
Solution}

Example:

Zero-divisor-fault = { E1 / E2;
E2 is zero;
if (E2 != 0) {E1 / E2}}

Step 2: Apply the fault patterns to identify the potential faulty statements in the program.

Example:

```
public class Example{
  public int Divide(int numerator, int divisor) {
    return numerator / divisor;
}
```

Applying the Zero-divisor-fault pattern, we can find the potential fault in the return statement.

Research Topics

- How to **represent a machine understandable fault pattern**?
- How to **organize fault patterns** in the fault pattern base for efficient application?
- How to **ensure the completeness** of the fault pattern base?
- How to efficiently **apply fault patterns** for a given program?

(2) Program segment testing (PST)

Definition A program segment is a subprogram extracted from a large program.

The goal of the PST is to detect potential bugs through running the relevant program segment during the program construction.

The merits of PST include:

(1) Can efficiently find runtime bugs

(2) Can be applied during program construction for fault prevention.

Example. Suppose a Java program is:

```
int Search(int x, int a[]) {
 int p := -1;
 int k := 0;
 while (k <= a.length) {
    if (a[k] == x) {
      p := k;
      break;
    } else {
        k++;
    }
        return p;
}
```

To find out whether there is any potential array out of bound problem, we form the following program segment:

```
int Search(int x, int a[], int k) {  /*test variable*/
int p := -1;  /*environmental variable*/
while (k <= a.length) {
    if (a[k] == x) {
        p := k;
        break;
    }
     return p;
}
```

The procedure of PST

Step 1: extract a program segment from a given program (or partial program)

Step 2: Generate test data for test variables

Step 3: Execute the program segment with test data

Step 4: Analyze test results

Research topics

- How to automatically **extract an appropriate program segment** from a given program for a specified type of fault?
- How to automatically **generate effective test data** for testing program segments?
- How to automatically **analyze test results** to determine whether any potential faults exist?

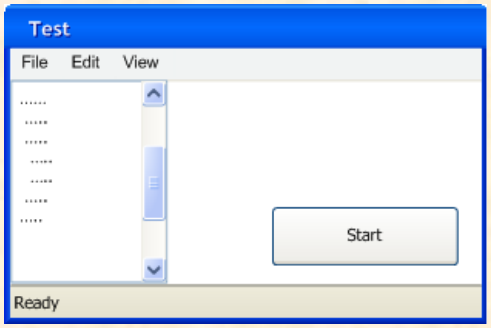
4. Program Verification

To overcome the weaknesses of testing and formal verification, we have developed the TBFV for program verification:

Testing-Based Formal Verification (TBFV)

The Goal of TBFV

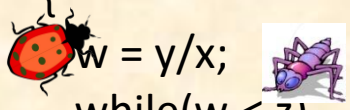
Press a Button



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

The Characteristics of TBFV

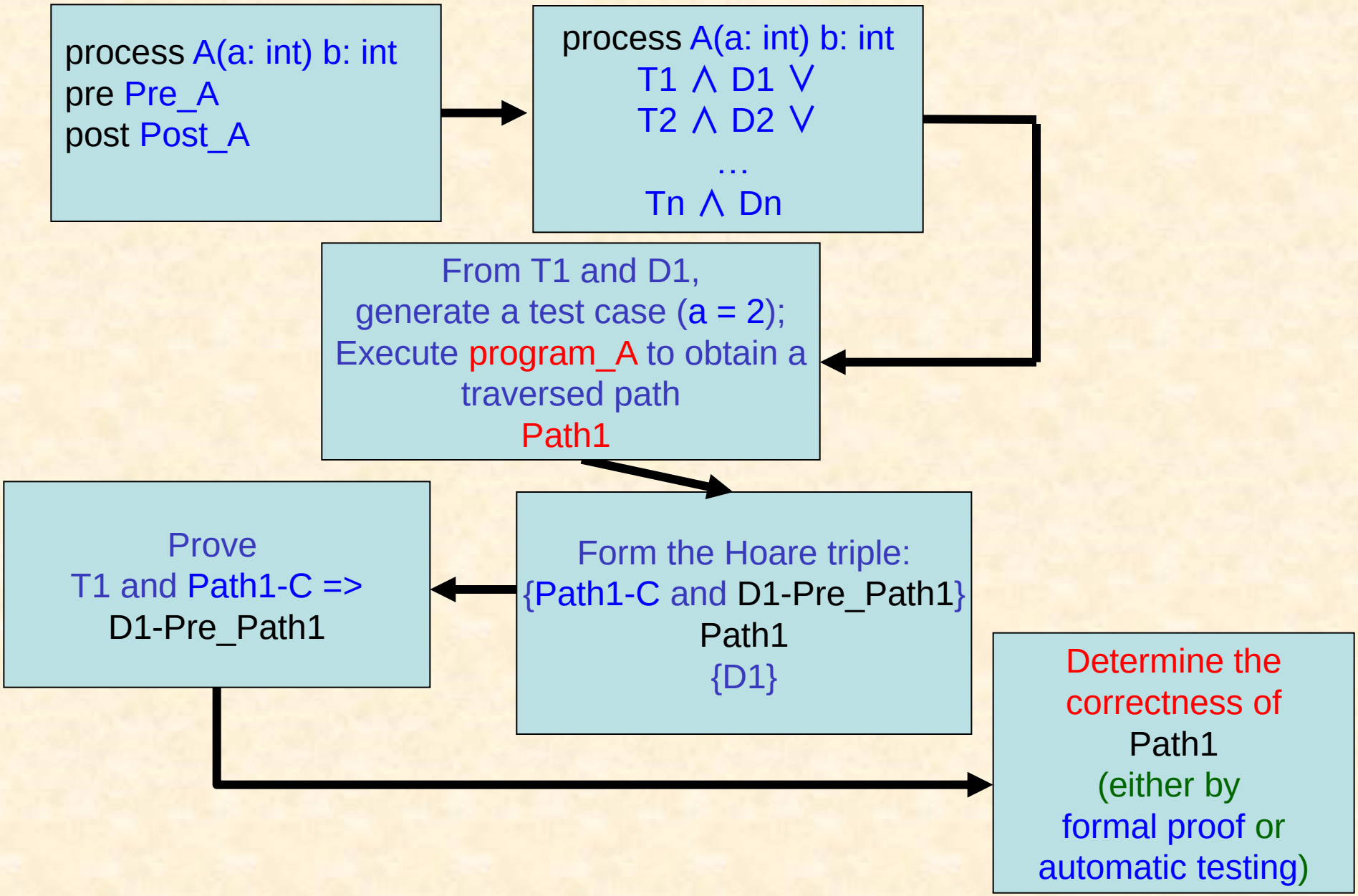
- TBFV is based on specification-based testing (SBT), symbolic execution (SE), and Hoare logic (HL). SBT is used to find program paths; SE is used to derive “path condition” and “state condition”; Hoare logic is used to help form the theorem for proving the correctness of the path.
- The correctness of a program path can be formally verified based only on one execution of the path.
- Program verification and validation can be done by verifying and validating all the representative program paths (branch sequence coverage).
- Deriving loop invariants is no longer necessary for formal verification in TBFV
- Fully automated

Formal specification of operations

A formal specification of a process in SOFL (Structured Object-Oriented Formal Language):

```
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 Principle of TBFV



A specification-based testing method used in the TBFV

Functional scenario-based testing is a specific method for specification-based testing that is suitable for automation.

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

Specification (in SOFL)

```
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

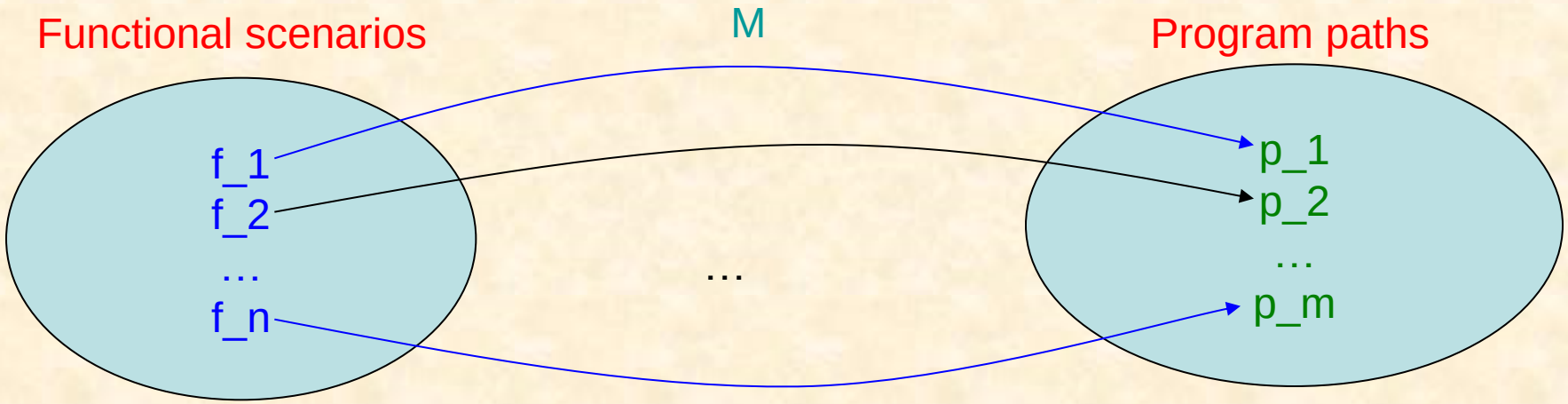
```
int A(int x) {
If (x > 0) {
 if (x > 10)  y := x * 1;
 else  y := x – 1;
 return y; }
else System.out.println("the
       pre is violated")  }
```

Satisfy?

Functional scenario:

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

(i=1,…,n)



Specification:

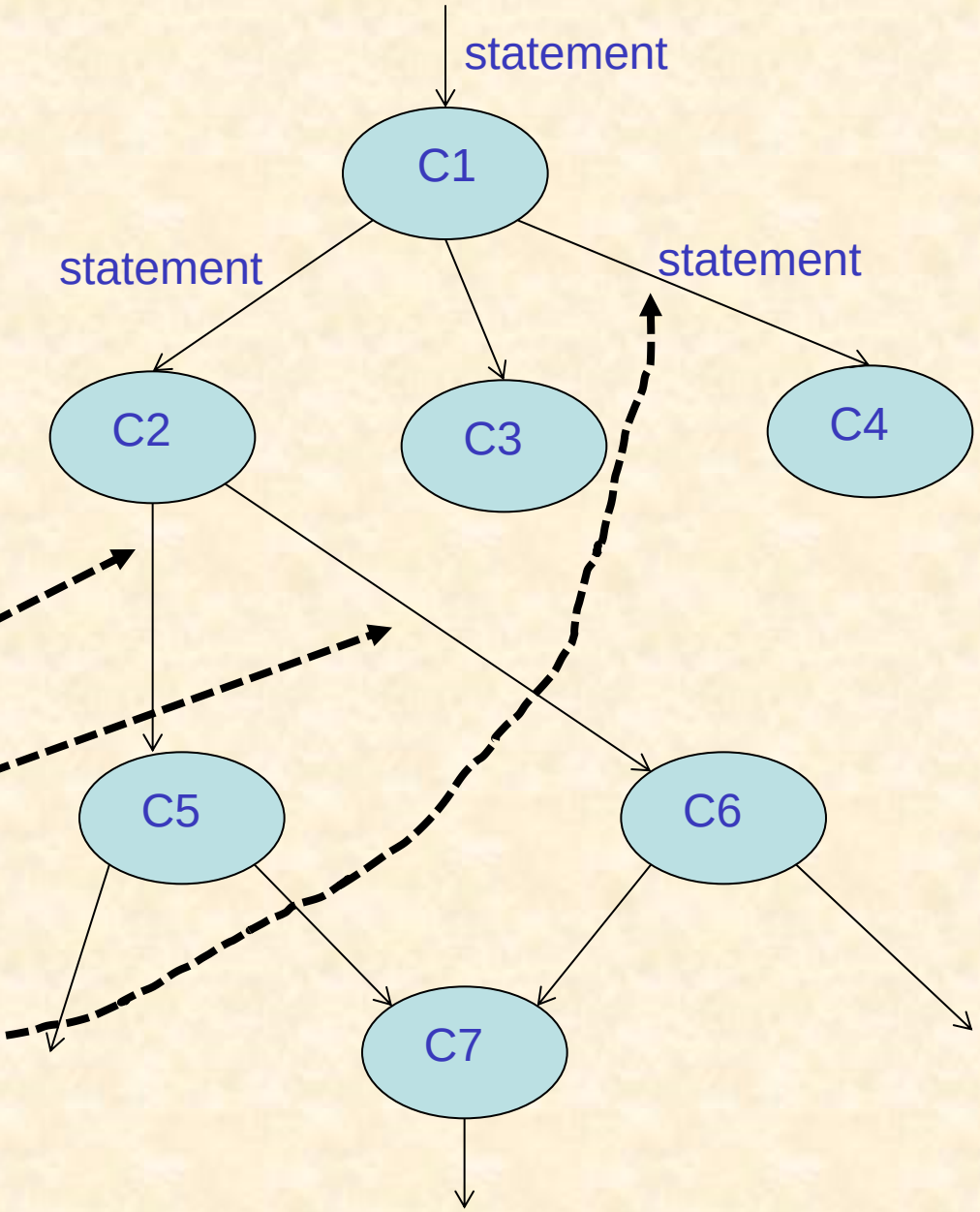
```
process A(x: int) y: int
pre   x > 0
post (x > 10 and y = x + 1) or
       (x <= 10 and y = x – 1)
end_process
```

Derivation

Functional scenarios

f_1
f_2
…
f_n

Program:



Definition 2.1 (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

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

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

Test strategy:

Let operation S have an FSF: $(S_{pre} \wedge G_1 \wedge D_1) \vee (S_{pre} \wedge G_2 \wedge D_2) \vee \cdots \vee (S_{pre} \wedge G_n \wedge D_n)$,
where $(n \geq 1)$.

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

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

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

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

Relevant axioms in Hoare logic:

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

(2) {Q} S {Q} where S is one of the non-state-change statements, such as

"return'' statement,

printing statement.

(3) {S ∧ 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:

Suppose we have a process that searches a given integer x in a given sequence a. If x is found, the position of its first appearance will be provided as the result. Otherwise, -1 will be the result.

```
process Search(x: int, a: seq of int) p: int
pre true
post  (exists[i: inds(a)] | a(i) = x) and
          p = hd([i | exists[i: inds(a)] | a(i) = x])
                              or
          x notin elems(a) and p = -1
```

A Java implementation:

```
int Search(int x, int a[]) {
 int p := -1;
 int k := 0;
 while (k < a.length) {
    if (a[k] == x) {
      p := k;
      break;
    } else {
        k++;
    }
        return p;
}
```

From the functional scenario:

(exists[i: inds(a)] | a[i] = x) and

p = hd([i | exists[i: inds(a)] | a(i) = x])

we generate a test data: x = 9, a = [3, 9, 20].

With this test data, we traversed the path:

```
 p := -1;
 k := 0;
k < a.length
a[k] <> x;
 k := k + 1;
k < a.length;
a[k] == x;
 p := k;
 break;
 return p;
```

Then, we apply the relevant axioms to derive the Path1-C and D1-Pre_Path1:

{Path1-C and D1-Pre_Path1}

```
{0 < a.length  and a[0] <> x and 0 + 1 < a.length and a[0 + 1] = x and
   0 + 1 = hd([i | exists[i: inds(a)] | a(i) = x])}
  p := -1;
{0 < a.length  and a[0] <> x and 0 + 1 < a.length and a[0 + 1] = x and
   0 + 1 = hd([i | exists[i: inds(a)] | a(i) = x])}
  k := 0;
{k < a.length  and a[k] <> x and k + 1 < a.length and a[k + 1] = x and
   k + 1 = hd([i | exists[i: inds(a)] | a(i) = x])}
 k < a.length
{a[k] <> x and k + 1 < a.length and a[k + 1] = x and k + 1 = hd([i | exists[i: inds(a)] | a(i) = x])}
 a[k] <> x;
{k + 1 < a.length and a[k + 1] = x and k + 1 = hd([i | exists[i: inds(a)] | a(i) = x])}
  k := k + 1;
{k < a.length and a[k] = x and k = hd([i | exists[i: inds(a)] | a(i) = x])}
 k < a.length;
{a[k] = x and k = hd([i | exists[i: inds(a)] | a(i) = x])}
 a[k] == x;
{k = hd([i | exists[i: inds(a)] | a(i) = x])}
 p := k;
{p = hd([i | exists[i: inds(a)] | a(i) = x])}
 break;
 {p = hd([i | exists[i: inds(a)] | a(i) = x])}
  return p;
  {p = hd([i | exists[i: inds(a)] | a(i) = x])}   /*We use D1 to denote this post-assertion.*/
```

We need to prove for the correctness of Path1:

T1 and Path1-C => D1-Pre_Path1

Specifically, we need to prove:

(exists[i: inds(a)] | a(i) = x) and
0 < a.length and a[0] <> x and
0 + 1 < a.length and a[0 + 1] = x =>
0 + 1 = hd([i | exists[i: inds(a)] | a(i) = x])

Methods for verifying this theorem:

(1) Formal proof

(2) Automatic testing

5. Conclusions

- Human-Machine Pair Programming (HMPP) is a promising technology for software development, but the research on it is just beginning.
- Fault prevention is an effective and efficient way to enhance software productivity and reliability.
- Testing-Based Formal Verification (TBFV) has set up an attractive direction for future research on the integration of testing and formal verification to provide effective technologies for both software verification and validation.

5. Future Work

(1) Develop effective technologies for **integrating specification animation into the three-step approach** to constructing hybrid specifications for fault prevention.

(2) Develop **specific techniques** for automatic pattern-based static analysis and program segment testing.

(3) Improve the testing-based formal verification (**TBFV**) method to support **scenario-based program verification and validation**.

Reference Books

"**Formal Engineering for Industrial Software Development Using the SOFL Method**",

by **Shaoying Liu**,

Springer-Verlag, 2004,

ISBN 3-540-20602-7

软件开发的形式化工程方法

——结构化+面向对象+形式化

清华大学出版社

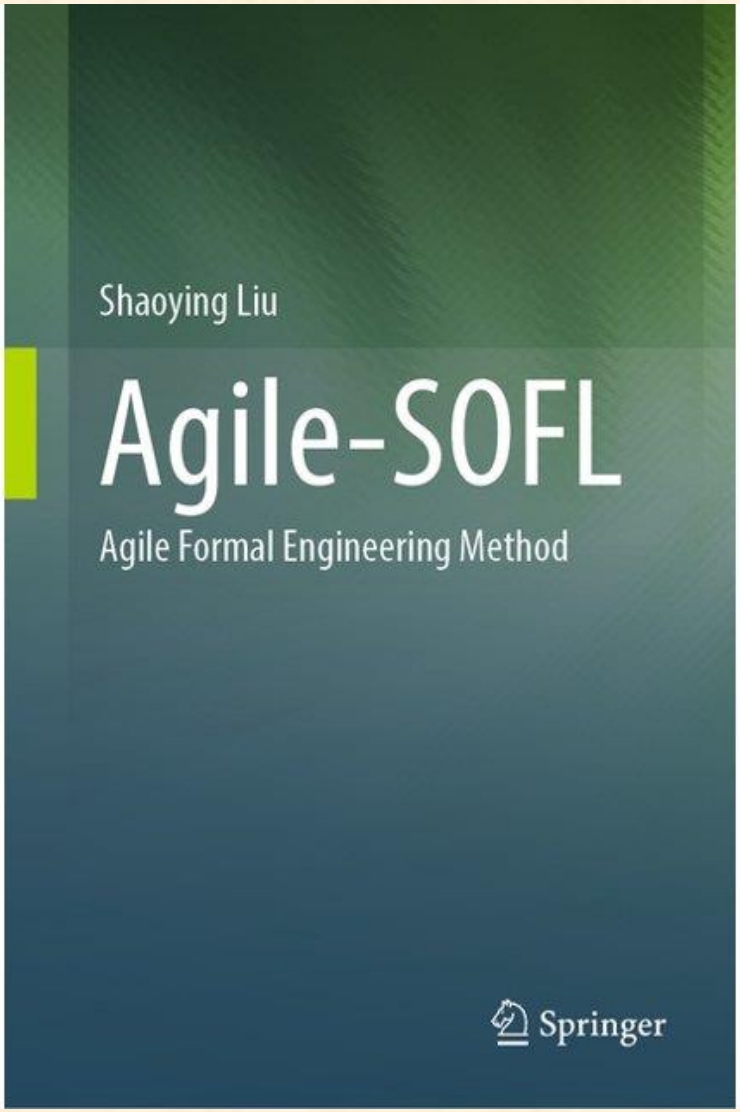
Agile-SOFL Book

"**Agile-SOFL: Agile Formal Engineering Method**",

by **Shaoying Liu**,

Springer-Verlag, June 2024,

ISBN: 978-981-97-2284-6
978-981-97-2285-3 (eBook)



Related Publications

My homepage

https://home.hiroshima-u.ac.jp/sliu/

Acknowledgement

I appreciate my group members over the last 25 years for their contributions to help advance the technologies in formal engineering methods and HMPP, including

Ai Liu, Rong Wang, Yuting Chen, Fumiko Nagoya, Weikai Miao, Xi Wang, Mo Li, Jiandong Li, Dingbang Fang, Haiyi Liu, Pingyan Wang, Lei Rao, Yang Li, Yujun Dai, Xiong Deng, Wen Jiang, Tetsuo Fukuzaki, Yuxiang Shang, Yu Du, and Yanzhao Xia

Thank you !