

# Agile Formal Engineering Method for High Productivity and Reliability

**Shaoying Liu**

**Department of Computer Science**



**Hosei University, Tokyo, Japan**

**HP: <https://sliu.cis.k.hosei.ac.jp/>**

**This work is supported by JSPS KAKENHI  
grant number 26240008.**

# Overview

1. Can We “Fall in Love” with Agile Approaches?
2. The SOFL Formal Engineering Method
3. Agile-SOFL: Agile Formal Engineering Method
4. Agile-SOFL Three-Step Specification and Animation
5. Specification-Based Incremental Implementation
6. Testing-Based Formal Verification
7. Tool Support for Agile-SOFL
8. Conclusions and Future Research
9. Reference

# 1. Can We “Fall in Love” with Agile Approaches?

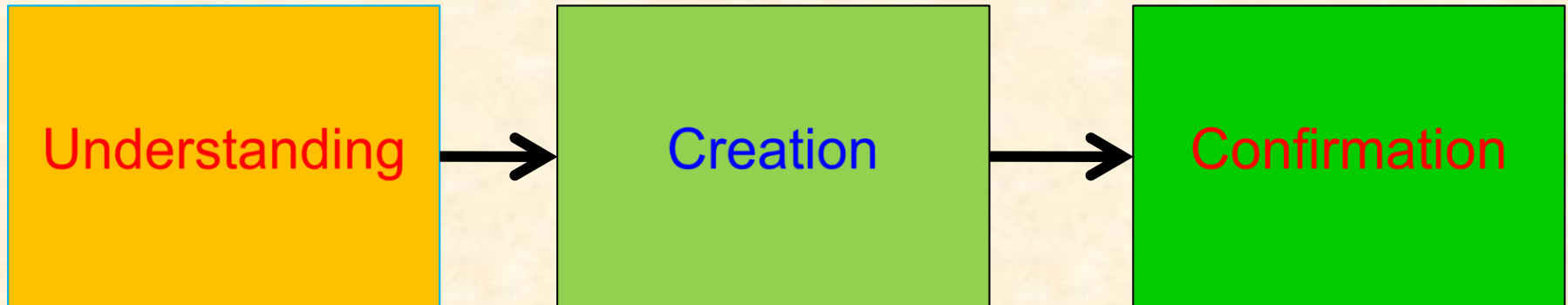
My answer: **Yes and No!**

**Yes:** if your project is small and short  
( $\leq 5000$  LOC,  $\leq 5$  months).

**No:** if your project is large and long, especially  
for a critical system.

**Why?**

# Necessary activities for producing highly reliable software systems:



# Agile manifesto:

- (1) Individuals and interactions over processes and tools**
- (2) Working software over comprehensive documentation**
- (3) Customer collaboration over contract negotiation**
- (4) Responding to change over following a plan**

# Advantages and Disadvantages of Agile Approaches

## Advantages:

- Working software can help strengthen the communication between the developer and the end-user.
- No comprehensive documentation except code can help reduce the time for documentation and the time for configuration management.
- Quick releases can be expected.

## Disadvantages:

- Frequent **changes of code** is inevitable (for lacking sufficient understanding of the requirements in the beginning), which can be extremely difficult and time-consuming.
- **Understanding of code** is required, which can be extremely hard as well.
- Frequent changes may **create more bugs** in code and testing to uncover the bugs is **time-consuming**.

# 2. The SOFL Formal Engineering Method

## Characteristics:

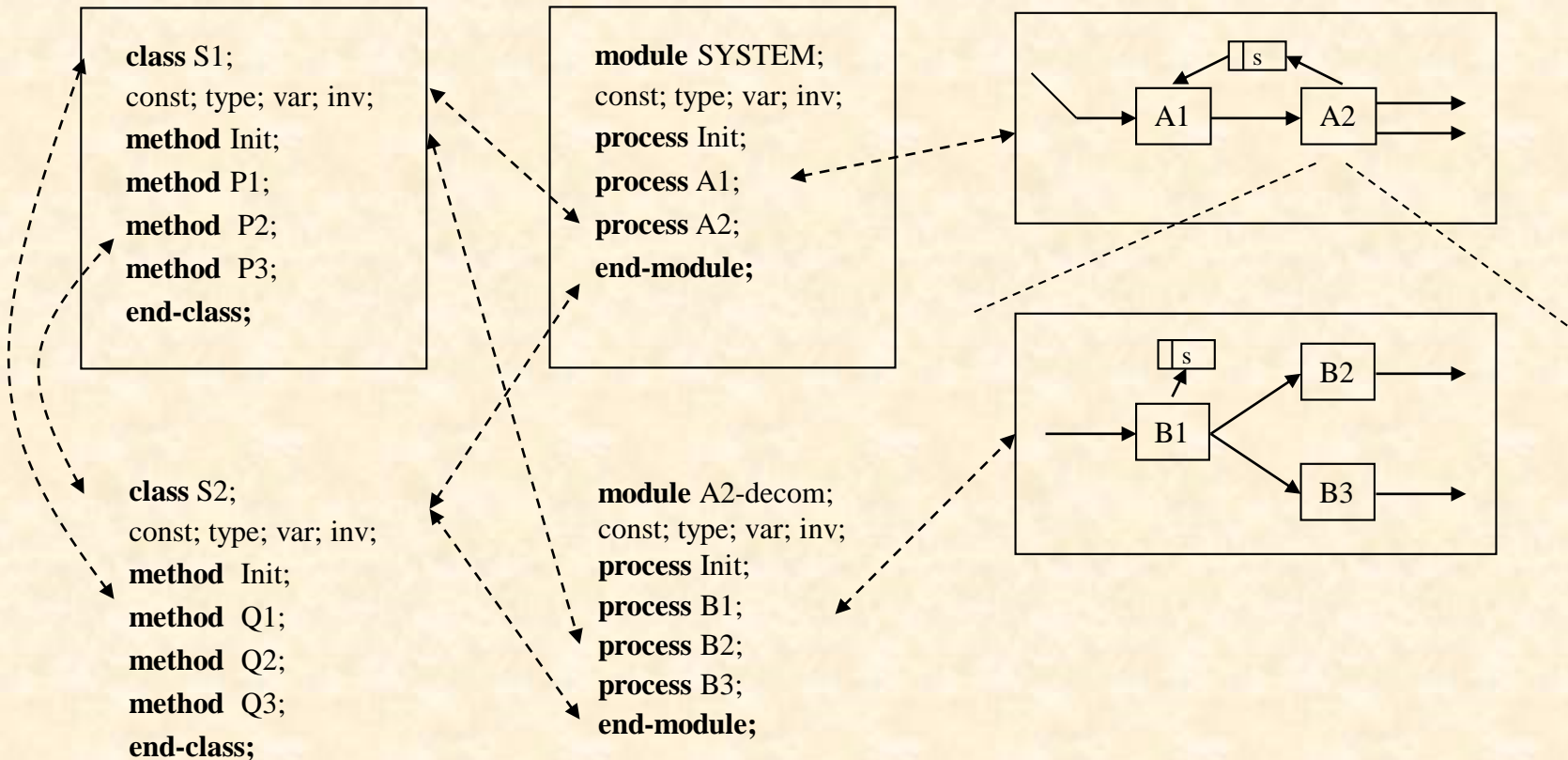
- Integration of formal methods (FM) with conventional software engineering technologies
- Comprehensible formal specification-based software construction and verification (inspection and testing), more practical than FM
- High automation in inspection and testing

## Challenges:

- Time consuming for formal specification construction and evolution to keep consistency with the code.
- Difficult in communication between stakeholders via formal specifications.



# The structure of a SOFL specification: CDFDs + modules + classes



# Questions?

**(1) Whether the disadvantages of Agile approaches can be overcome by taking advantage of the SOFL formal engineering method?**

**(2) If yes, how?**

# 3. Agile-SOFL: Agile Formal Engineering Method

**Agile-SOFL is a FEM with effective techniques to achieve the values given in the Agile manifesto.**

**Characteristics:**

1. A three-step approach to building comprehensible **hybrid specification** for analyzing requirements and defining what to be done by the potential system.
2. **Animation**-based techniques for specification validation.
3. **Testing-Based Formal Verification (TBFV)** for program verification.
4. **Incremental implementation** together with the application of TBFV in small cycles

# Principle of Agile-SOFL

The Agile-SOFL Three-Step Specification

+

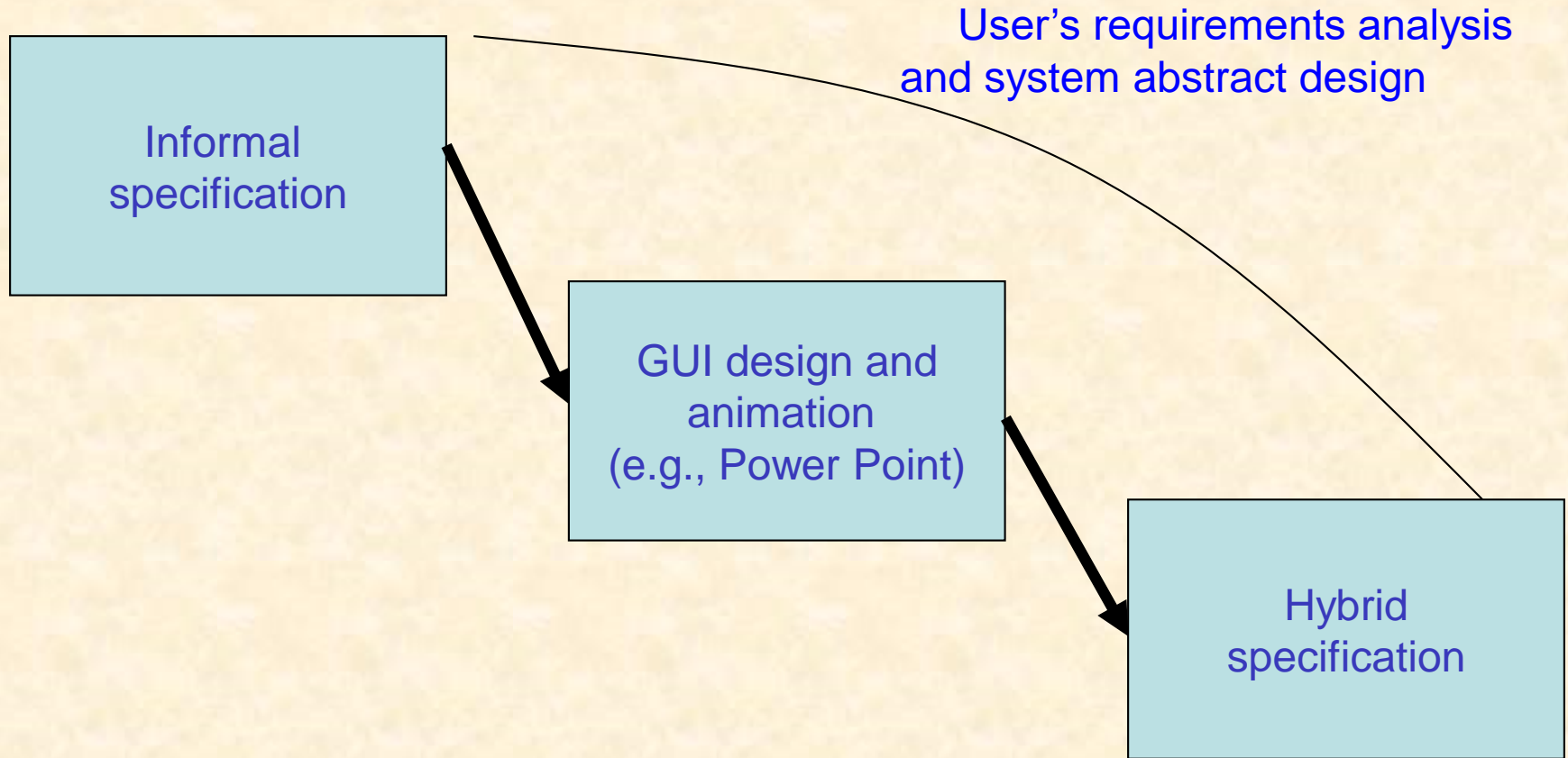
GUI-Based Specification Animation

Software  
defects and  
errors

Specification-Based  
Incremental  
Implementation

Testing-Based  
Formal  
Verification

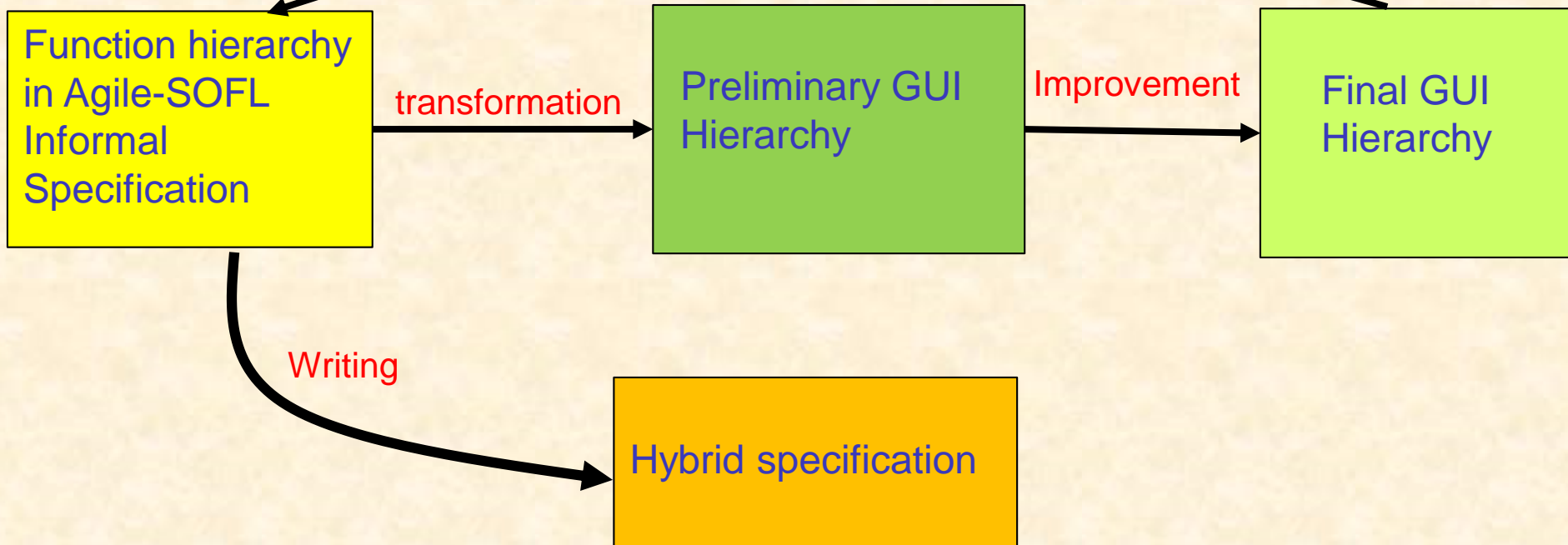
# 4. Agile-SOFL Three-Step Specification



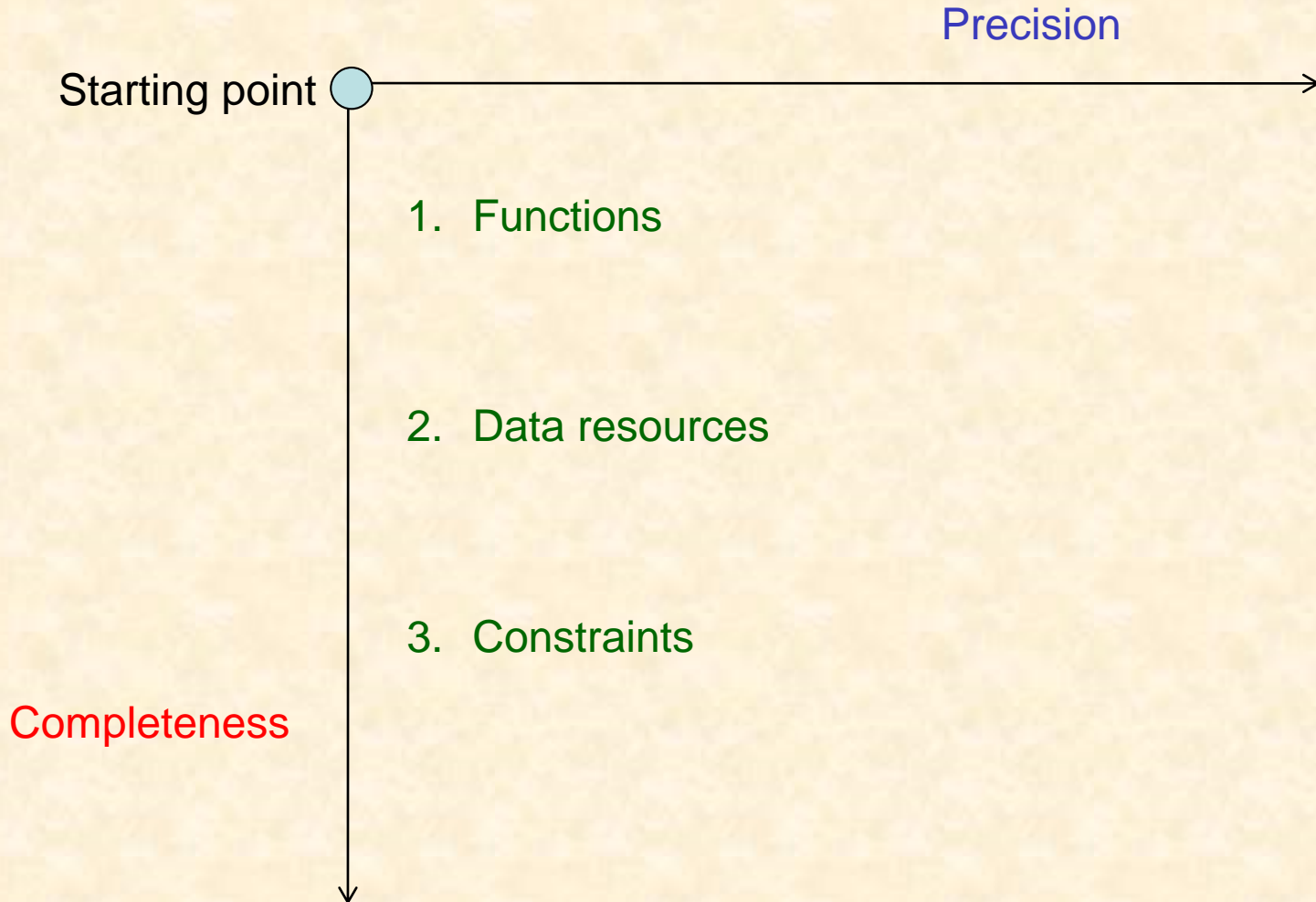
**An Agile-SOFL hybrid specification is a specification written in SOFL that contains both semi-formal specifications and formal specifications for operations.**

# Major Ideas of the GUI-Aided Approach to Writing Hybrid Specifications

Animation and evolution for completeness and detailed information



Tasks for informal specification: **Capturing** desired **functions**, necessary **data resources**, and **constraints** on both functions and data resources.



# Informal Specification

**Informal specification for a simplified ATM software:**

## **1. Functions**

- 1.1 Register a customer**
- 1.2 Withdraw from the bank account**
  - 1.2.1 Check the card id and password**
  - 1.2.2 Check the amount for withdrawal**
  - 1.2.3 Update the account balance after withdrawal**
- 1.3 Deposit to the bank account**
- 1.4 Transfer from one bank account to another**
- 1.5 Inquire about the balance of the bank account**
- 1.6 Finish operations**

## **2. Data resources**

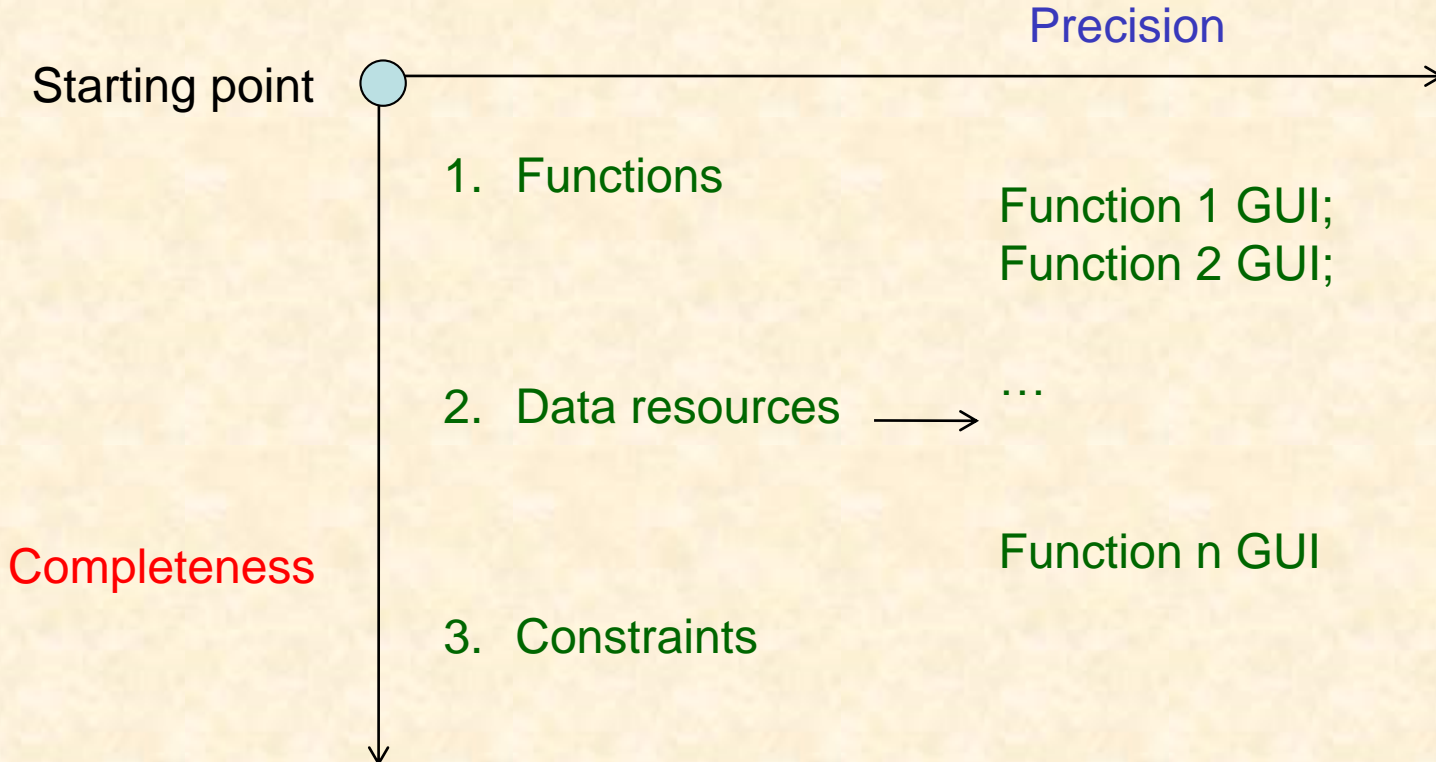
- 2.1 Bank account (F1.2, F1.3, F1.4, F1.5)**
  - 2.1.1 Account name**
  - 2.1.2 Account number**
  - 2.1.3 Account password**
  - 2.1.4 Account balance**
  - 2.1.5 Bank name**
  - 2.1.6 Bank branch code**
- 2.2 Accounts file (F1.2, F1.3, F1.4, F1.5) /\*containing a set of bank accounts\*/**
- 2.3 Customer information(F1.1)**

## **3. Constraints**

- 3.1 Each withdrawal from a bank account must not exceed 200,000 JPY.**
- 3.2 The account balance cannot be less than 0.**
- 3.3 The amount of each transfer cannot exceed 1,000,000 JPY.**
- 3.4 The amount of each deposit cannot exceed 500,000 JPY**



# The result of the GUI design and animation phase:



# Example

**Derived GUI hierarchy  
from the ATM informal  
specification:**

1.1 Register ...

1.2 Withdraw ...

1.3 Deposit ...

1.4 Transfer ...

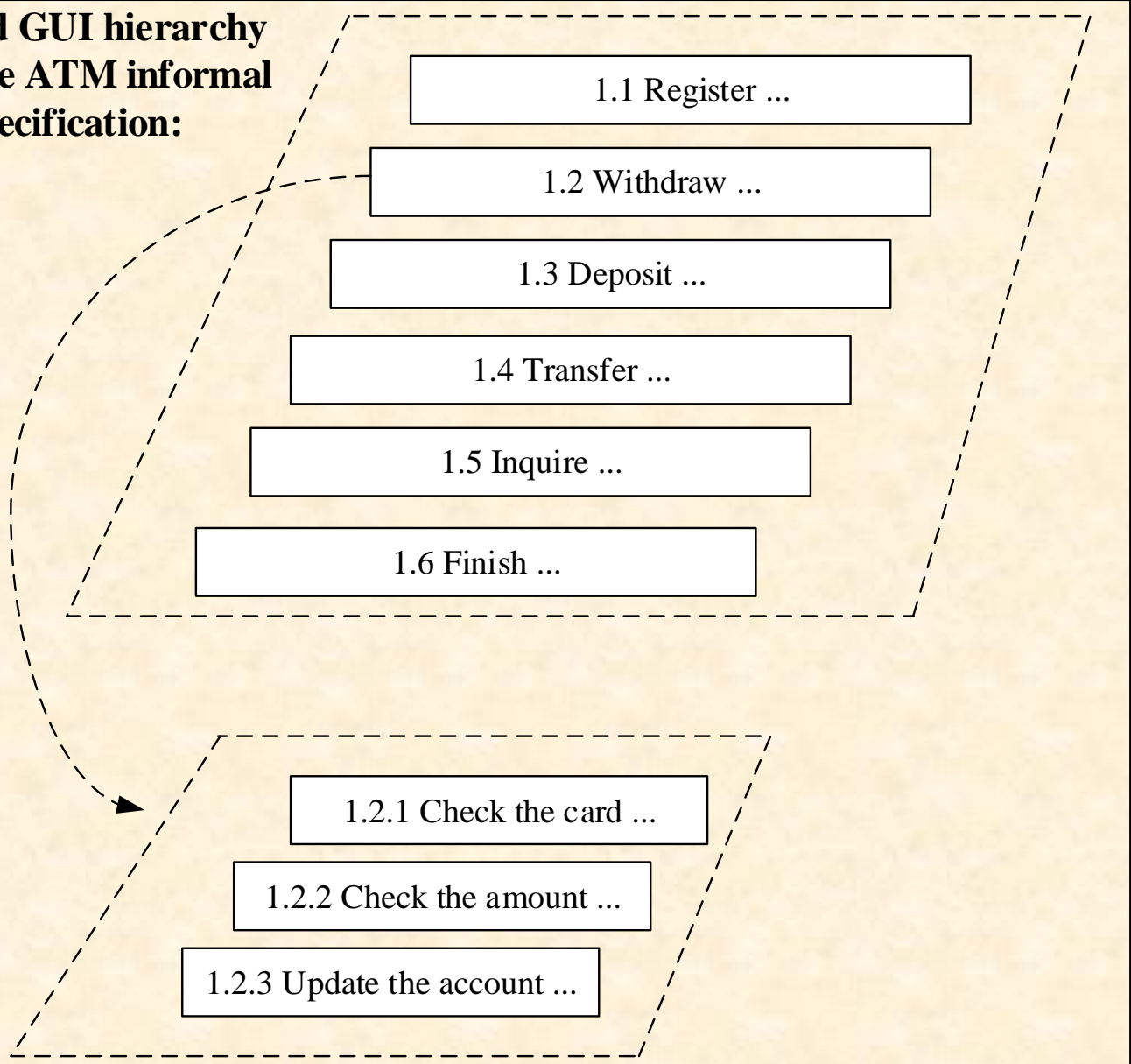
1.5 Inquire ...

1.6 Finish ...

1.2.1 Check the card ...

1.2.2 Check the amount ...

1.2.3 Update the account ...



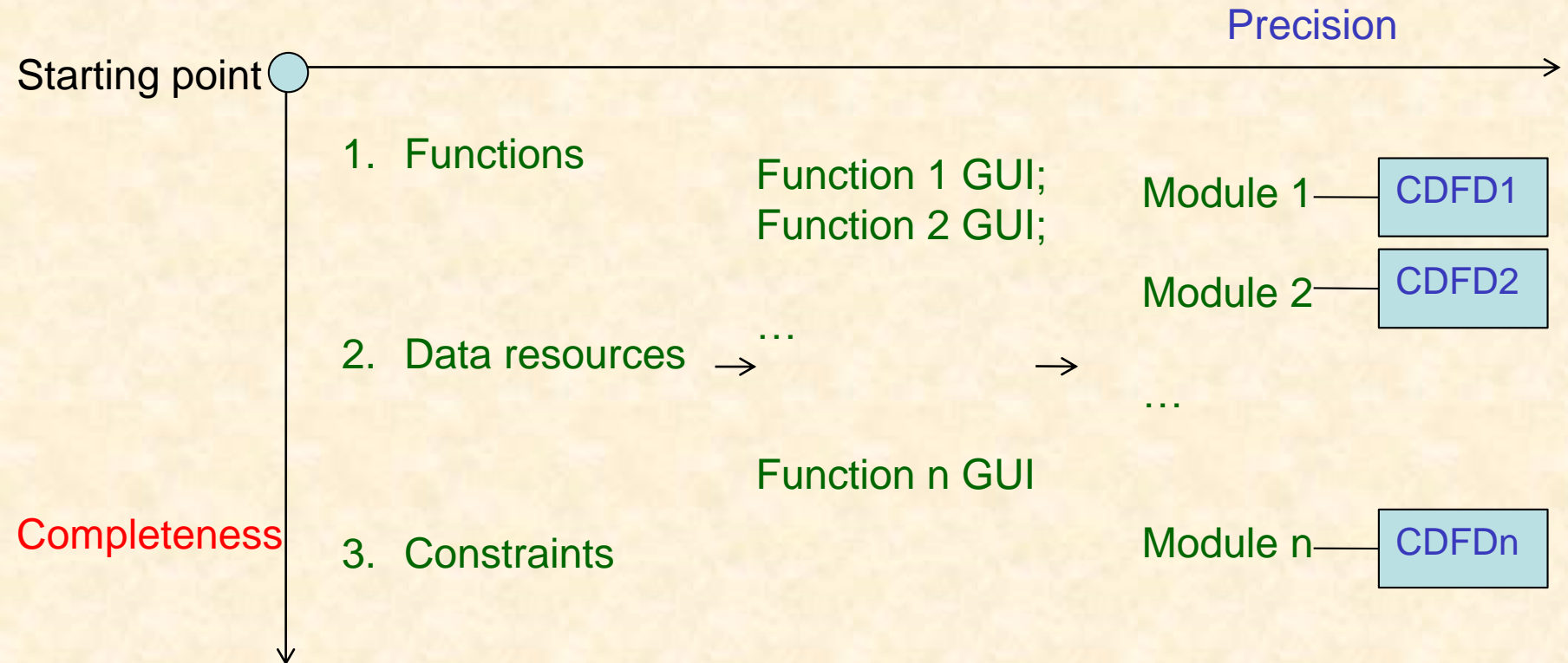
# Improved Final GUI



## Major tasks for hybrid specification:

- (1) Form processes for each function given in the informal specification and define their data flow dependence using CDFD (condition data flow diagram).
- (1) Write specification for each process occurring in the CDFD. Each specification is given in pre- and post-conditions, which can either be a restricted informal expression or a formal expression.

## The result of writing the hybrid specification:



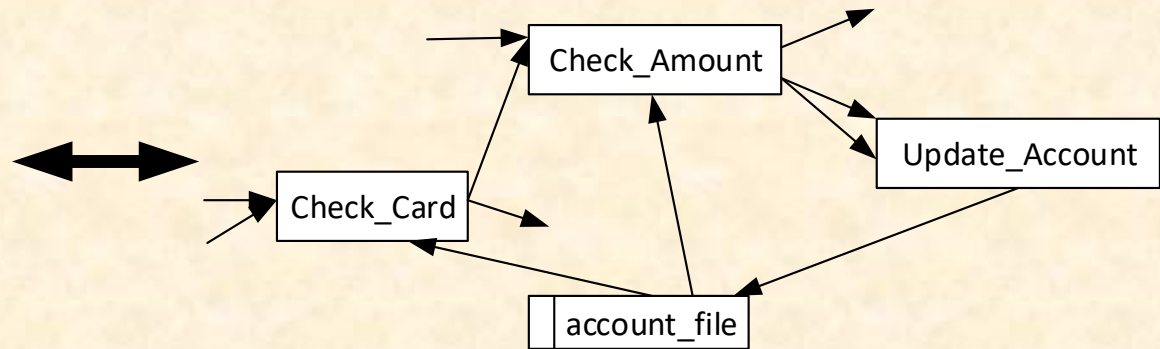
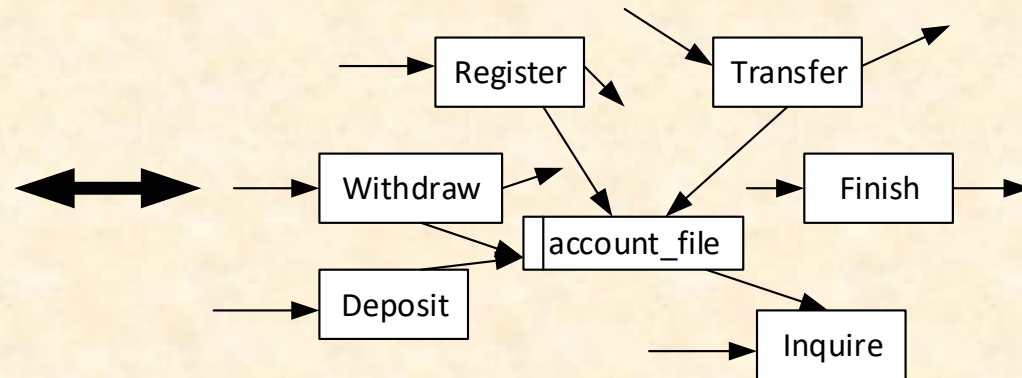
# Example

## Formal specification:

```
module SYSTEM_ATM;  
data items declarations;  
process Register  
process Withdraw  
process Deposit  
process Transfer  
process Inquire  
process Finish  
end_module;
```

```
module Withdraw_Decom /  
SYSTEM_ATM;  
data items declarations;  
process Check_Card  
process Check_Amount  
process Update_Account  
end_module;
```

No. 1



No. 2

## Details of the specification (example):

```
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;
```

```
  /*Account balance must be greater than or equal to zero. */
```

```
  ...
```

```
  behav CDFD_No.1;
```

```
process Withdraw(amount: real, account1: Account)
    e_msg: string | cash: real
ext wr account_file
pre account1 is a member of account_file
post if amount is less than the balance of account1
    then supply cash with the same amount as amount, and
        reduce the amount from the balance of the account.
    else output an appropriate error message e_msg.
end_process;
```

**/\*Semi-formal specification\*/**



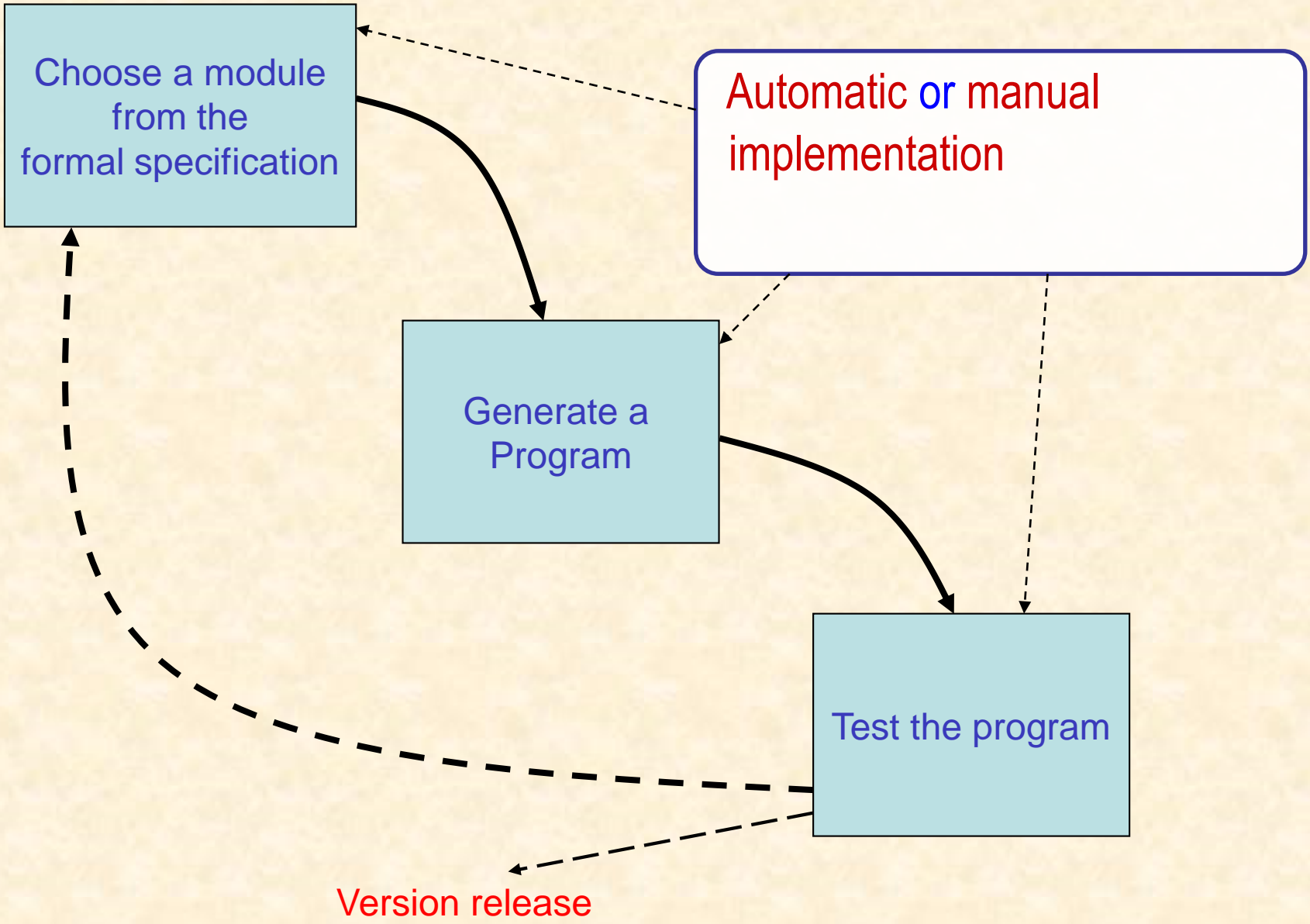
```
process Withdraw(amount: real, account1: Account)
    e_msg: string | cash: real
ext wr account_file
pre account1 inset account_file
post if amount <= account1.balance
    then
        cash = amount and
        let Newacc =
            modify(account1, balance -> account1.balance - amount)
        in
            account_file = union(diff(~account_file, {account1}), {Newacc})
    else
        e_meg = "The amount is over the limit. Reenter your amount."
comment
...
end_process;

/*Formal specification*/

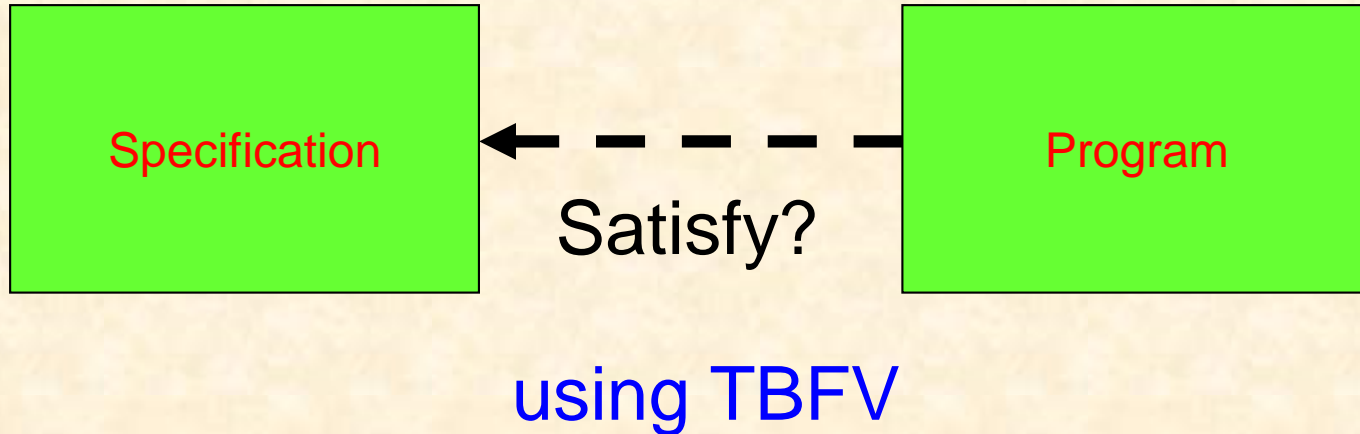
end_module
```

# 5. Specification-Based Incremental Implementation

We take the **bottom-up approach** to **automatically** or **manually** (with tool support) **implement** and **test** the system **based on the formal specification** in an incremental fashion.



# 6. Testing-Based Formal Verification



The goal:

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

# The theoretical foundation for TBFV

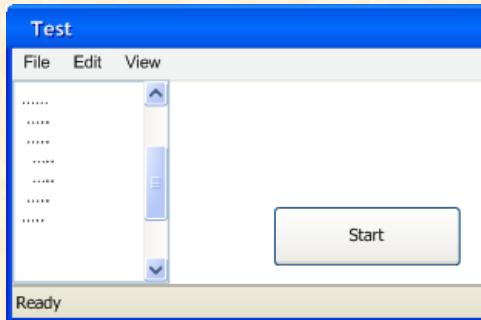
A program **P** correctly implements a specification **S** iff

$$\mathbf{S}_{\text{pre}}(\sim\sigma) \vdash \mathbf{S}_{\text{post}}(\sim\sigma, \mathbf{P}(\sim\sigma))$$

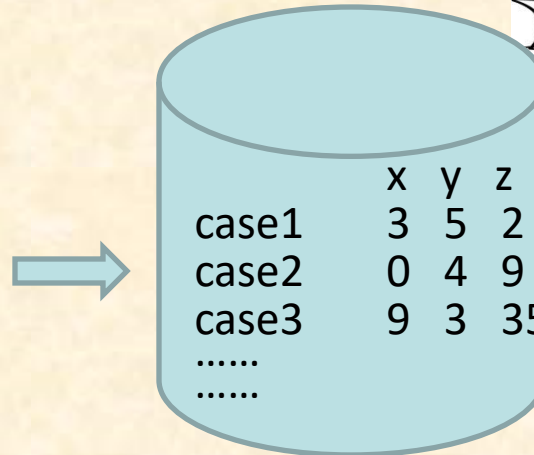
where  $\sim\sigma$  is any initial state and  $\mathbf{P}(\sim\sigma)$  is treated as a mathematical function whose definition may not be represented by a mathematical expression but can be represented by an **algorithm**. Therefore, existing formal proof techniques may not be applied for formal Verification of **P**.

# Goal of Automatic TBFV

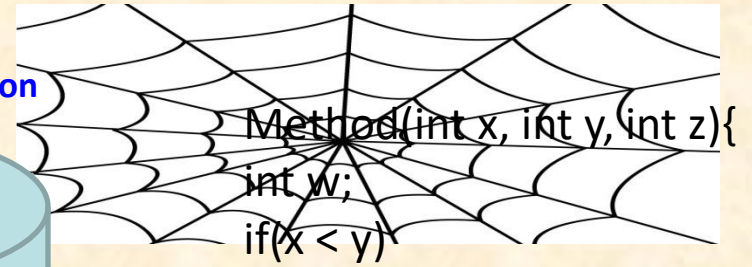
Press a Button



Automatic test data generation



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



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



Next

## Steps of TBFV:

- (1) Generate test data from the specification.
- (2) Execute the program using the test data.
- (3) Analyze test results to detect bugs based on the test data, the result of execution, and the specification.

# General criteria for test data generation and for test result analysis:

## Definition 5.1 (FSF)

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

$G_i$ : guard condition

$D_i$ : defining condition.

$i = 1, \dots, n$ .

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

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



**Criterion 5.1:** Let the FSF of specification **S**

be:

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

Then, a test set **T** must be generated to meet the following condition:

$$\left( \bigwedge G_i \exists t \in T \cdot S_{pre} \wedge G_i(t) \right) \wedge \left( \exists t \in T \cdot \neg S_{pre} \right)$$

where  $i = 1, \dots, n$

# A criterion for test result analysis:

**Criterion 5.2:** If the condition

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

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

## 7. Tool Support for Agile-SOFL

We have several prototype tools to support the SOFL specification language and method.

- Agile-SOFL specification construction tool (SpecTool)
- Tool for Specification-Based Testing

# SpecTool for A-SOFL specification

The screenshot displays the SOFL (Specification-Oriented Formal Language) environment. The interface is divided into several panes:

- Module Explorer:** Shows a hierarchical tree of modules. The selected module is `SuicaCard.fModule`, which contains sub-modules like `Init.fModule`, `Payment.fModule`, and `Update.fModule`.
- Diagram:** A state transition diagram for the `SuicaCard.cdfd` module. It features four main components: `Init`, `Update`, `payment`, and `Reference`.
  - `Init` receives `suicaData` and outputs `errorMessage` and `success`.
  - `Update` receives `buffer`, `user`, `commuter`, and `bank`, and outputs `errorMes` and `success`.
  - `payment` receives `payment` and outputs `errorMessage` and `success`.
  - `Reference` receives `ref` and outputs `suicaData`.Arrows indicate dependencies and data flow between these components.
- Properties:** A panel for the selected `Init` module, showing its name and output port numbers (1 and 2).
- Code Editor:** Displays the formal specification for the `SuicaCardModule`. It includes type declarations for `SuicaCard`, `User`, `Buffer`, `CommuterPass`, and `BankInfo`, along with a constant `minimum_require = 130` and a `use_history` sequence.

# A Tool for TBFV (SBTT)

The screenshot displays a software development environment. The main window has a menu bar with 'File', 'NewProject', 'OpenProject', 'SaveProject', 'CloseProject', and 'Test'. Below the menu is a project tree showing a folder 'MyTest1' containing two sub-items 'NewTest1'. The main workspace contains the following code:

```
class A(x : set of int , y : set of int) z : set of int
  t z = diff(x,y)
  _process

class B(x : seq of char , y : seq of char) z : set of char
  t (x <> [] and y <> [] and z = inter(elems(x),elems(y)))
  (x = [] or y = [] and z = {})
  _process
```

A smaller window titled 'NewTest1' is overlaid, showing a table with columns for input sequences, sets, and results:

x : seq of ch...	y : seq of ch...	z : set of char	pre	post	result
['a', 'b', '6']	['y']	[]	true	false	false
['q', 'l', 'a', 'b']	['q', 'b', '8']	[]	true	false	false

The main workspace also contains a code editor with the following code:

```
LinkedList inter(char[] seq1, char[] seq2){
  LinkedList list = new LinkedList();

  for(int i = 0; i < seq1.length; i++)
    list.add(new Character(seq1[i]));

  ListIterator ite = list.listIterator();
  Character obj;
  while(ite.hasNext()){
    obj = (Character)ite.next();
    for(int i = 0; i < seq2.length; i++)
      if(obj.equals(new Character(seq2[i])))
        list.remove(obj);
  }
  return list;
}
```

# 10. Conclusions and Future Work

## 10.1 Conclusions

- Agile-SOFL is believed to be much more effective than existing agile approaches for high productivity and reliability, and helpful for system maintenance and extension.
- Agile-SOFL is characterized by the **three-step specification approach, specification animation, specification-based incremental implementation, and testing-based formal verification (TBFV)** based on SOFL.
- Agile-SOFL supports the values emphasized in the Agile Manifesto, such as **individuals and interactions, working software, customer collaboration, and responding to changes.**

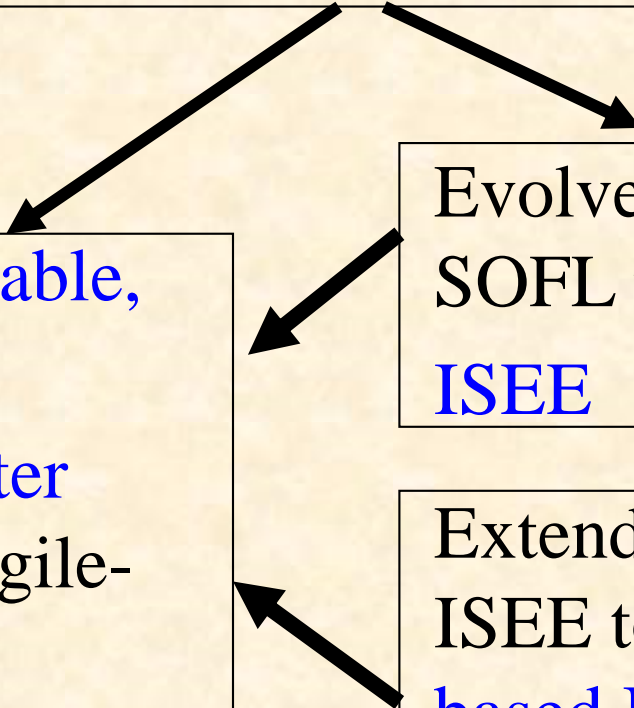
## 10.2. Future Work

Build a more **mature software engineering environment for Agile-SOFL** on the basis of the existing **prototype tools**.

**Develop dependable, large-scale, and complex computer systems** using Agile-SOFL under the support of its SEE

Evolve the SEE of Agile-SOFL to a **method-based ISEE**

Extend the method-based ISEE to a **method-domain-based ISEE** to support domain specific applications.



# Reference

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

by **Shaoying Liu**,  
Springer-Verlag, 2004,  
ISBN 3-540-20602-7

**URL:** <https://sliu.cis.k.hosei.ac.jp/>  
(for other publications)

