Validating Formal Specifications using Testing-Based Specification Animation

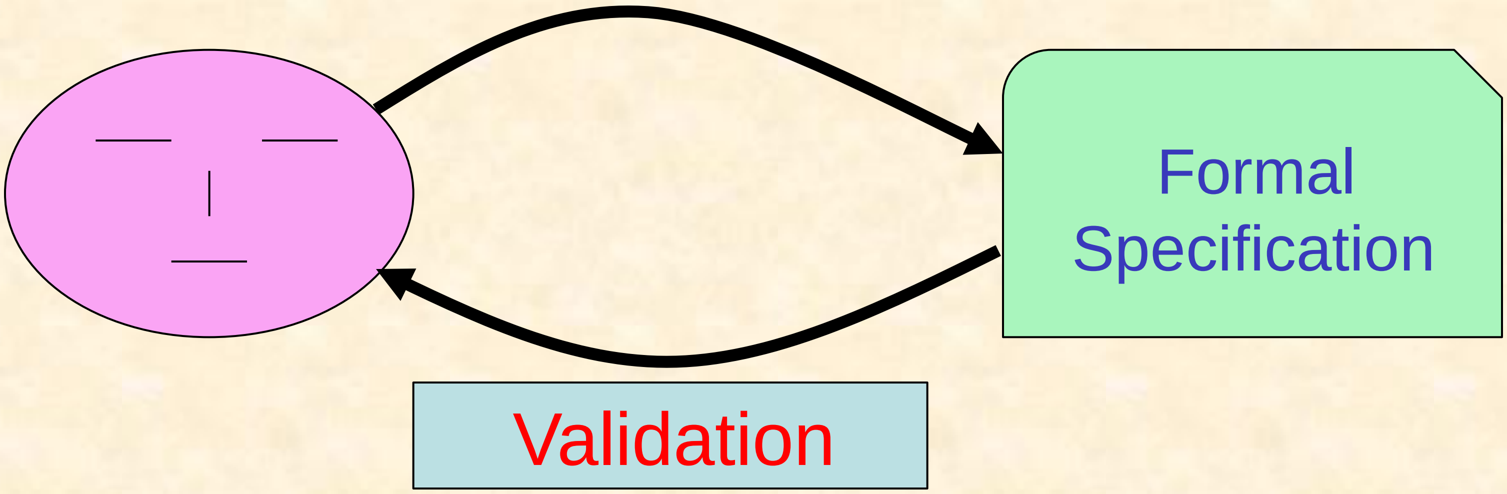
Shaoying Liu
Department of Computer Science
Faculty of Computer and Information Sciences
Hosei University, Japan
Email: sliu@hosei.ac.jp
HP: http://cis.k.hosei.ac.jp/~sliu/

This work was supported by JSPS KAKENHI Grant Number 26240008.

Overview

1. Challenge for Formal Methods in Validation
2. Testing-Based Specification Animation
3. Teat Case Generation
4. A Small Experiment
5. Conclusion
6. Future Work

1. Challenge for Formal Methods in Validation



Features of specification validation:

(1) Efficient and effective communications between the user and the analyst are required.

(2) Examples are needed because they are the most effective way to help the user understand the specification.

Formal proof cannot be applied for validation.

2. Testing-Based Specification Animation

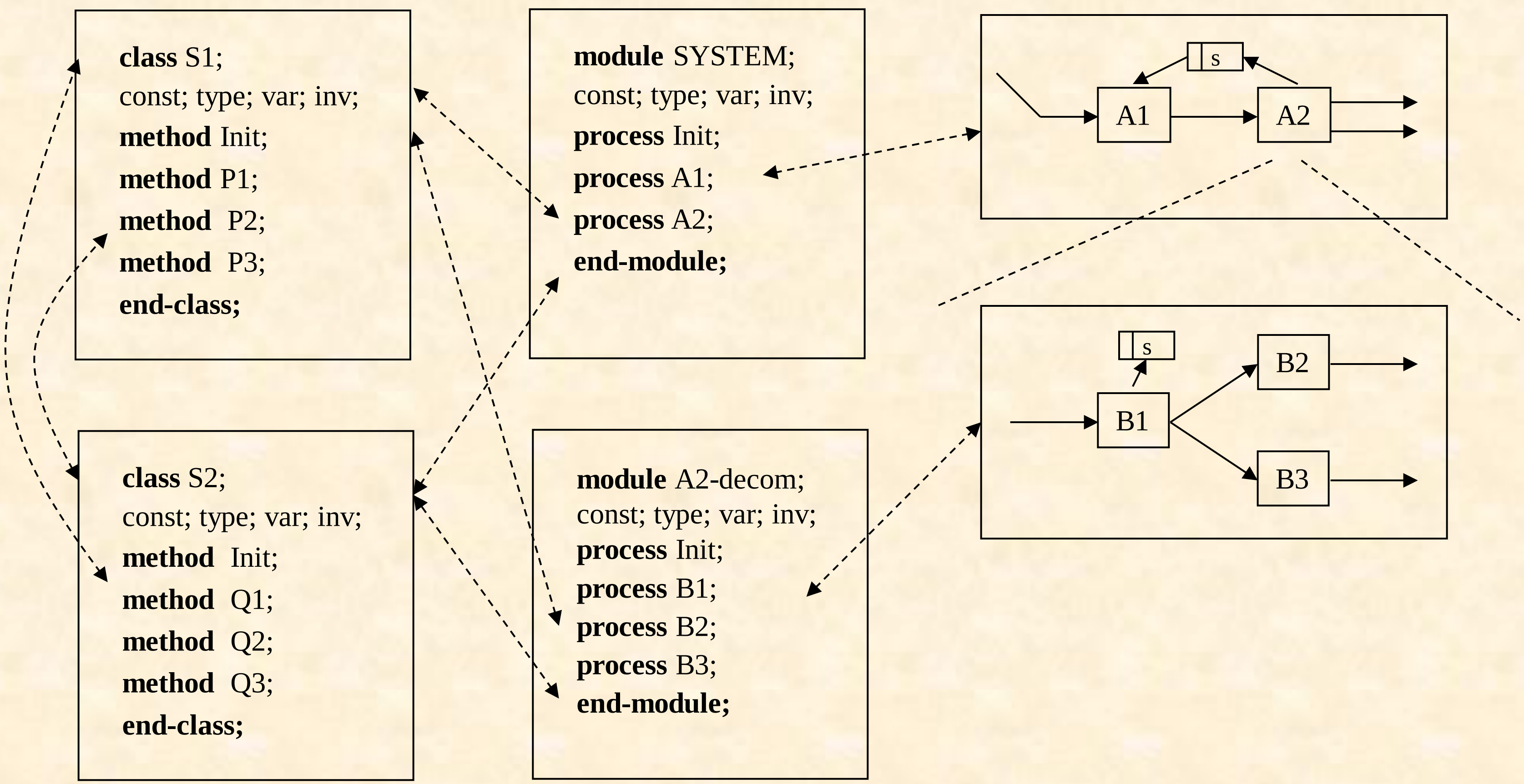
Specification animation is a technique and process to dynamically demonstrate the relation between input and output defined in the specification in a visualized fashion.

Testing-based specification animation is to use test cases to dynamically demonstrate the input-output relation in a visualized fashion.

SOFL: Structured Object-Oriented Formal Language

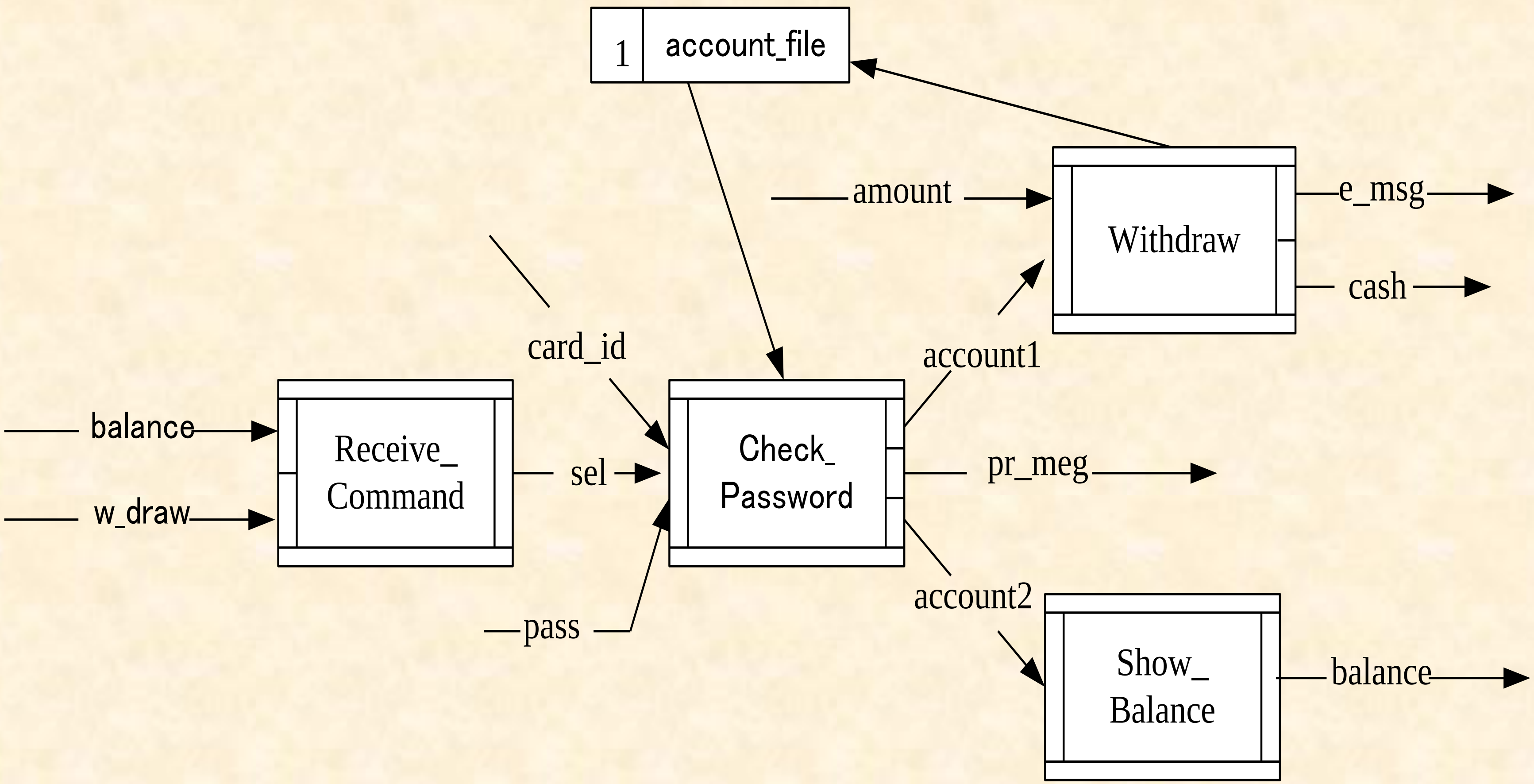
The structure of a SOFL specification:

CDFDs + modules + classes



Example:

A simplified ATM specification in SOFL:

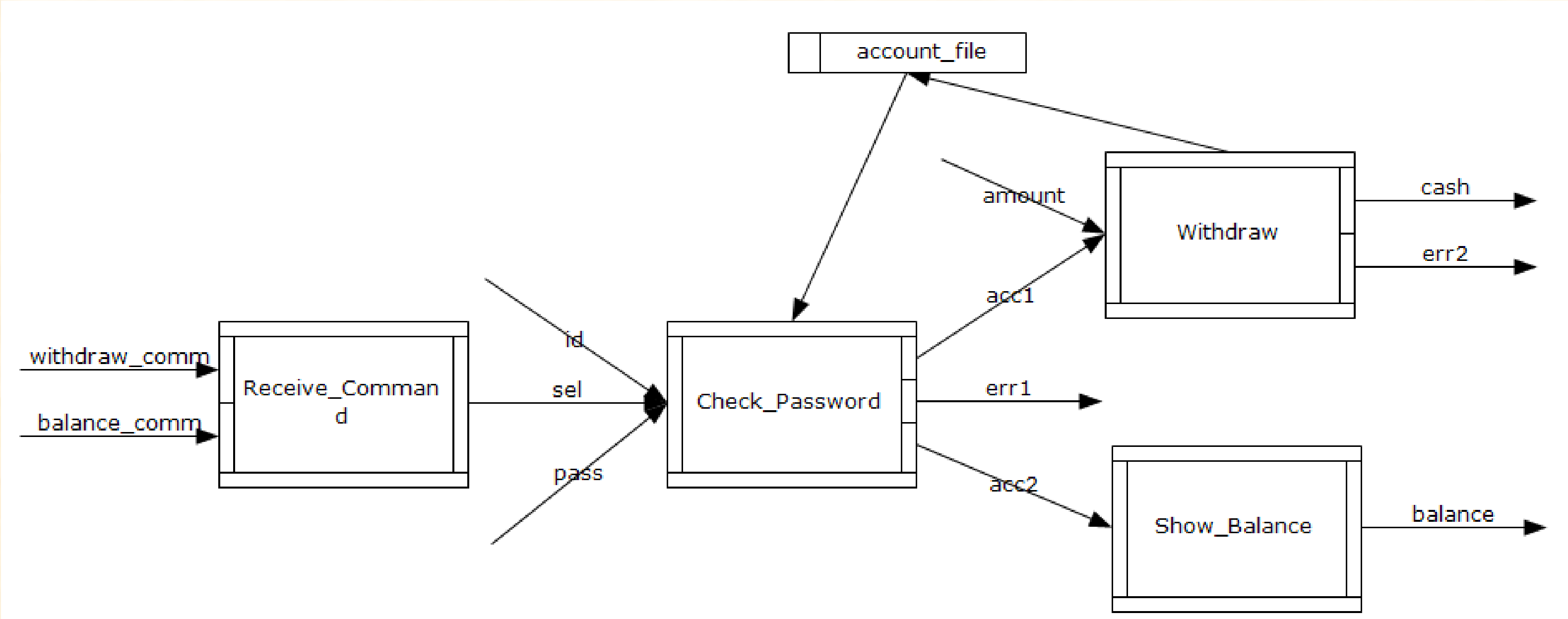


```
module SYSTEM_ATM;
  type
  Account = composed of
              account_no: nat
              password: nat
              balance: real
            end
var
  account_file: set of Account;
inv
   forall[x: account_file] | x.balance >= 0;

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 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}

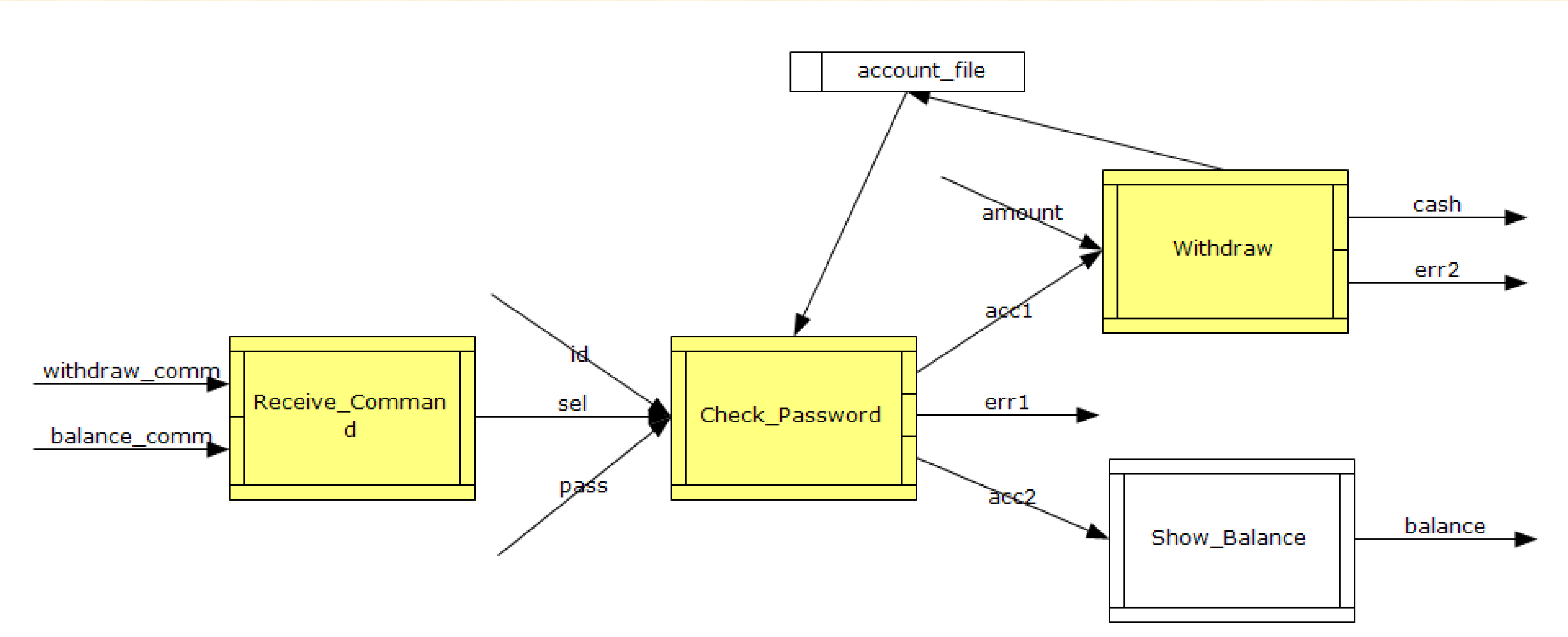
{balance_comm}[Receive_Command, Check_Password, Withdraw]{cash}

{balance_comm}[Receive_Command, Check_Password, Withdraw]{err2}

{balance_comm}[Receive_Command, Check_Password]{err1}

{balance_comm}[Receive_Command, Check_Password, Show_Balance]{balance}

Animation of a single scenario

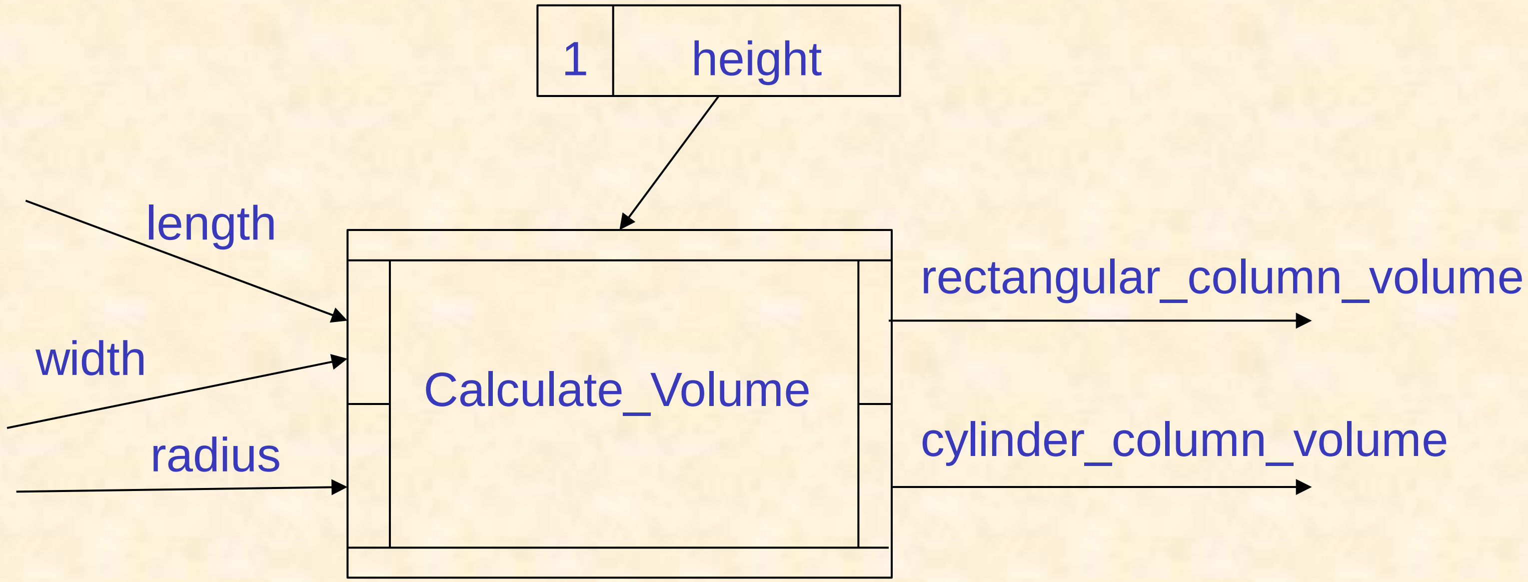


{withdraw_comm}[Receive_Command11, Check_Password11, Withdraw11]{cash}

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)}

Single Process Specification Animation

A process in SOFL is a six-tuple $(P, P_I, P_O, P_E, P_{pre}, P_{post})$.



```
process Calculate_Volume(length, width: real |
                                    radius: real)
          rectangular_column_volume: real |
          cylinder_volume: real
ext rd height: real
pre length >= 0 and width >= 0 and height >= 0 or
     radius >= 0 and height >= 0
post bound(length, width) and
        rectangular_column_volume = length * width * height    or
        bound(radius) and
        cylinder_volume = radius * radius * 3.14 * height
end_process;
```

3. Test Case Generation

We propose a functional scenario-based method for test case generation.

Definition 1: Let $P_{pre} = P_1 \vee P_2 \vee \ldots \vee P_n$ *and* $P_{post} = Q_1 \vee Q_2 \vee \ldots \vee Q_m$ be a disjunctive normal form, respectively. Then, we call a conjunction $P_i \wedge Q_j$ ($i = 1, \ldots, n$; $j = 1, \ldots, m$) a functional scenario.

Definition 2: The functional scenario $P_i \wedge Q_j$ of process P is said valid if and only if the following condition holds:

$$\forall in_1, in_2 \in P_I \cdot in_1 \neq in_2 \Rightarrow$$

$$(varSet(P_i) \subseteq in_1 \vee varSet(P_i) \subseteq in_2) \wedge$$

$$(varSet(P_i) \subseteq in_1 \Rightarrow varSet(Q_j) \cap in_2 = \{\}) \wedge$$

$$(varSet(P_i) \subseteq in_2 \Rightarrow varSet(Q_j) \cap in_1 = \{\})$$

A valid functional scenario $P_i \wedge Q_j$ ensures that the input satisfying P_i can be used in Q_j to define the output of the process, and therefore requires that P_i and Q_j do not contain input variables of different input ports.

Definition 3: A test case for a process *P* is a set of values for input, output, and external variables.

Example:

$tc = \{(x_1, 5), \ldots, (\sim z_1, 10), \ldots, (y_1, 50), \ldots, (z_1, 20), \ldots\}$

where x_i is an input variable, y_j an output variable, $\sim z_k$ an initial external variable, and z_k a final external variable.

Criteria for test case generation:

Criterion 1: Generate a test case for every group of input variables of every input port to ensure that at least one valid functional scenario is made true by each test case.

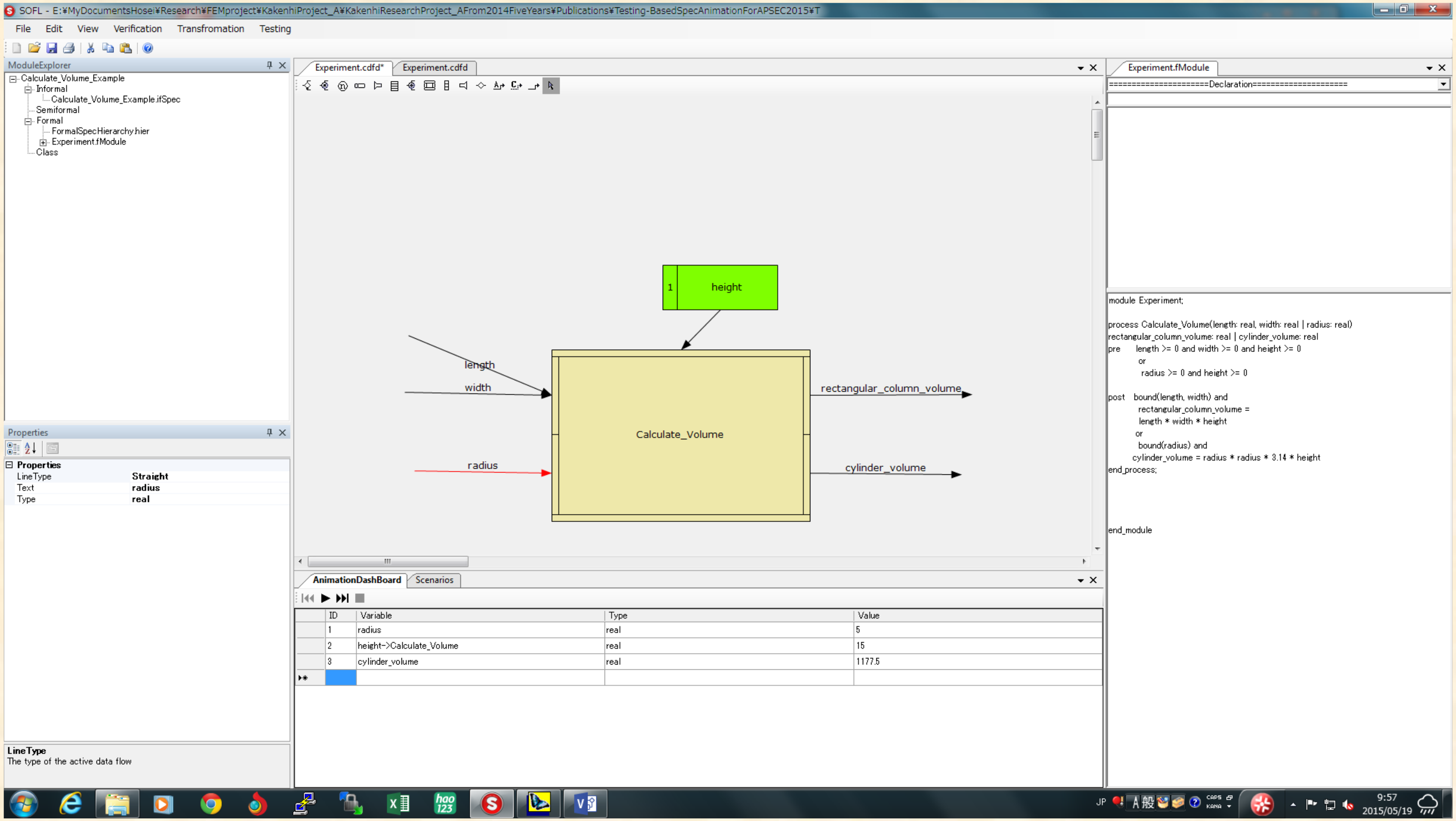
Criterion 2: Generate a test case for every group of output variables of every output port to ensure that at least one valid functional scenario is made true by each test case.

Criterion 3: Generate a test case for every initial external variable and every final external variable to ensure that at least one valid functional scenario is made true by each test case.

Criterion 4: For every valid functional scenario, generate a test case that makes the scenario true.

Criterion 5: For every function, data item, and constraint defined in the informal requirements specification, generate a test case to ensure that each of them is tested at least once.

Using a test case for animation: dynamic demonstration



4. A Small Experiment

The testing-based specification animation approach is compared to specification review on a railway card (called Suica card) system.

Processes	**Injected faults**	**Detected faults by Group *A***		**Detected faults by Group *B***	
		S1	***S2***	***S3***	***S4***
Register_Card	23	23	15	10	1
Charge_With_Cash	15	15	5	5	1
Charge_From_Bank	21	21	19	12	1
Buy_With_Card	15	14	5	7	1
Buy_With_Card_Cash	5	5	2	4	1
Entering_Station	24	19	11	10	2
Exiting_Station	34	34	17	23	2
Update_Commute_Ticket	19	17	19	14	4
Total	**156(100%)**	**148(95%)**	**93(60%)**	**85(54%)**	**13(8%)**

5. Conclusion

(1) The testing-based specification animation provides an effective approach to validating formal specifications. It does not require transformation from formal specifications to code.

(2) The test case generation criteria have proved to be effective for validation in the small experiment.

(3) The test cases generated for specification animation can be reused for testing the implementation.

6. Future Work

- Study more test case generation criteria for more effective specification animations.
- Study techniques for visualized demonstration of the input-output relation of a process, including both data visualization and functional visualization.
- Improve our current software tools to support automatic test case generation.
- Conduct more experiments on specifications of large scale software systems.

Thank You !