Software Formal Engineering Methods
（软件形式化工程方法）

Shaoying Liu（刘少英）

Graduate School of Advanced Science and Engineering

Hiroshima University, Japan

Email: **sliu@hiroshima-u.ac.jp**

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

The greatest challenge in software engineering

How to ensure the correctness and high reliability of software systems?

Disastrous consequences of software errors

1. **Errors With Rocket Launch**

Time: 1996

Accident: a satellite's payload was not delivered into the planned Earth's orbit using a European Ariane 5 rocket. The rocket self-destructed when it started disintegrating. **Cause:** a bug with the software used in launching the rocket. the problem was caused by the reuse of code from Ariane 4, the rocket's predecessor.

Disastrous consequences of software errors

2. Heathrow Terminal 5 Opening, 2008

Time: 2008

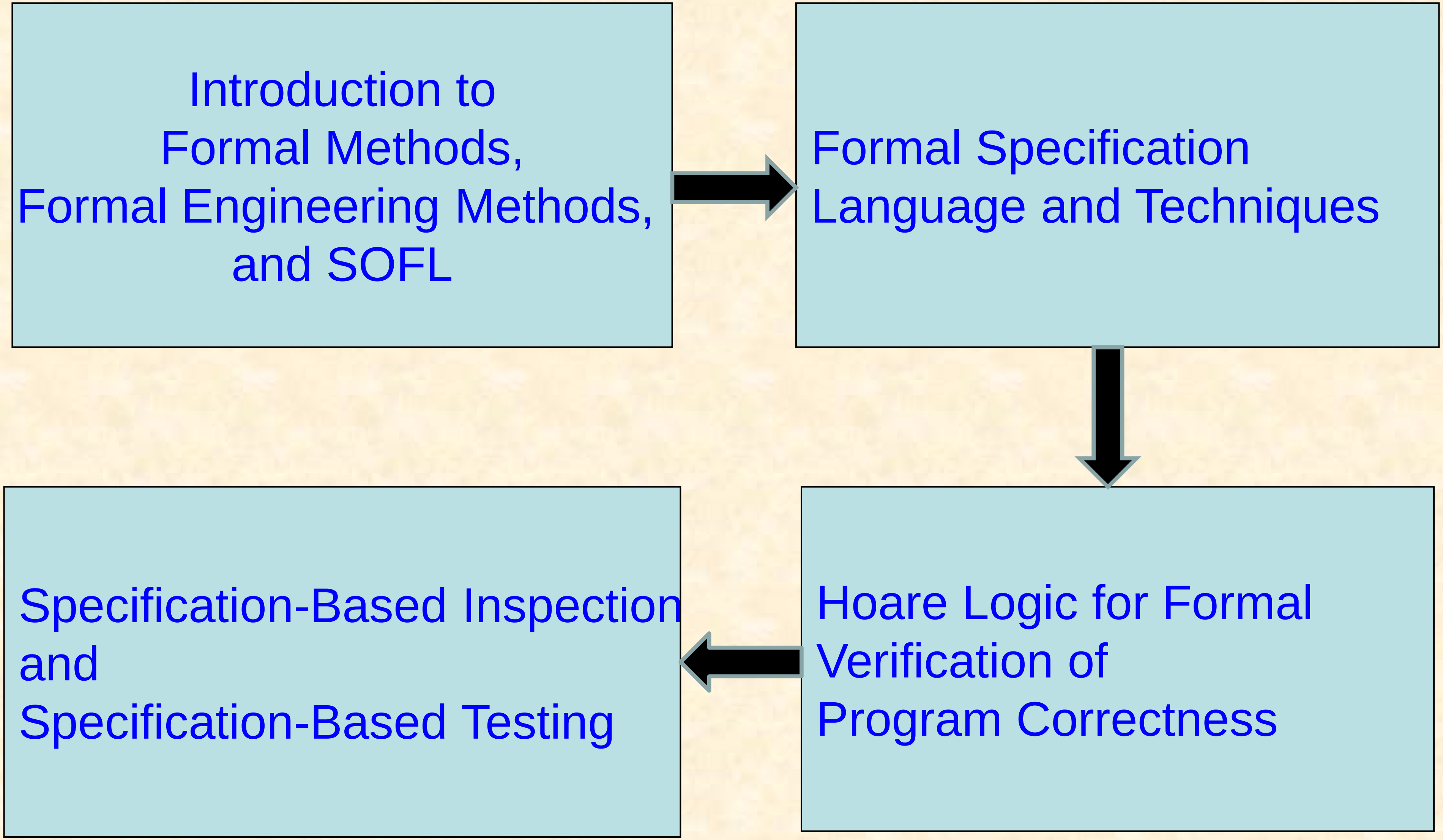
Accident: massive disruptions in managing baggage. Some 42,000 bags were lost and more than 500 flights canceled, costing more than £16 million.

Cause: a bug in the new baggage handling software system and problems with the wireless network system.

Goals of the course

- Understand what Formal Engineering Methods are and learn necessary techniques for writing formal specifications
- Understand the concept of program correctness and learn Hoare logic for formally verifying the correctness
- Learn the principles of specification-based inspection and specification-based testing as practical techniques for verifying programs

Contents of the course



Introduction to
Formal Methods,
Formal Engineering Methods,
and SOFL

Formal Specification
Language and Techniques

Hoare Logic for Formal
Verification of
Program Correctness

Specification-Based Inspection
and
Specification-Based Testing

Reference books

(1) C.A.R. Hoare and C.B. Jones, ``Essays in Computing Science'', Prentice Hall, 1989.

(2) Shaoying Liu, "Formal Engineering for Industrial Software Development",

Springer-Verlag, 2004.

(3) Latest publications on specification-based inspection and testing.

The Textbook

“**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

Shaoying Liu

Formal Engineering for Industrial Software Development

Using the SOFL Method

Springer

The ways to learn

- Attend lectures
- Work on class exercises and actively join discussions
- Complete a small project
- Join the final examination

Evaluation of Performance

The evaluation of each student’s performance for grading will be done based on a small project and the final examination.

(1) Small project: 40%

(2) Examination: 60% (90 minutes)

I. Introduction to Formal Methods, Formal Engineering Methods, and SOFL

1. Challenges in traditional Software Engineering

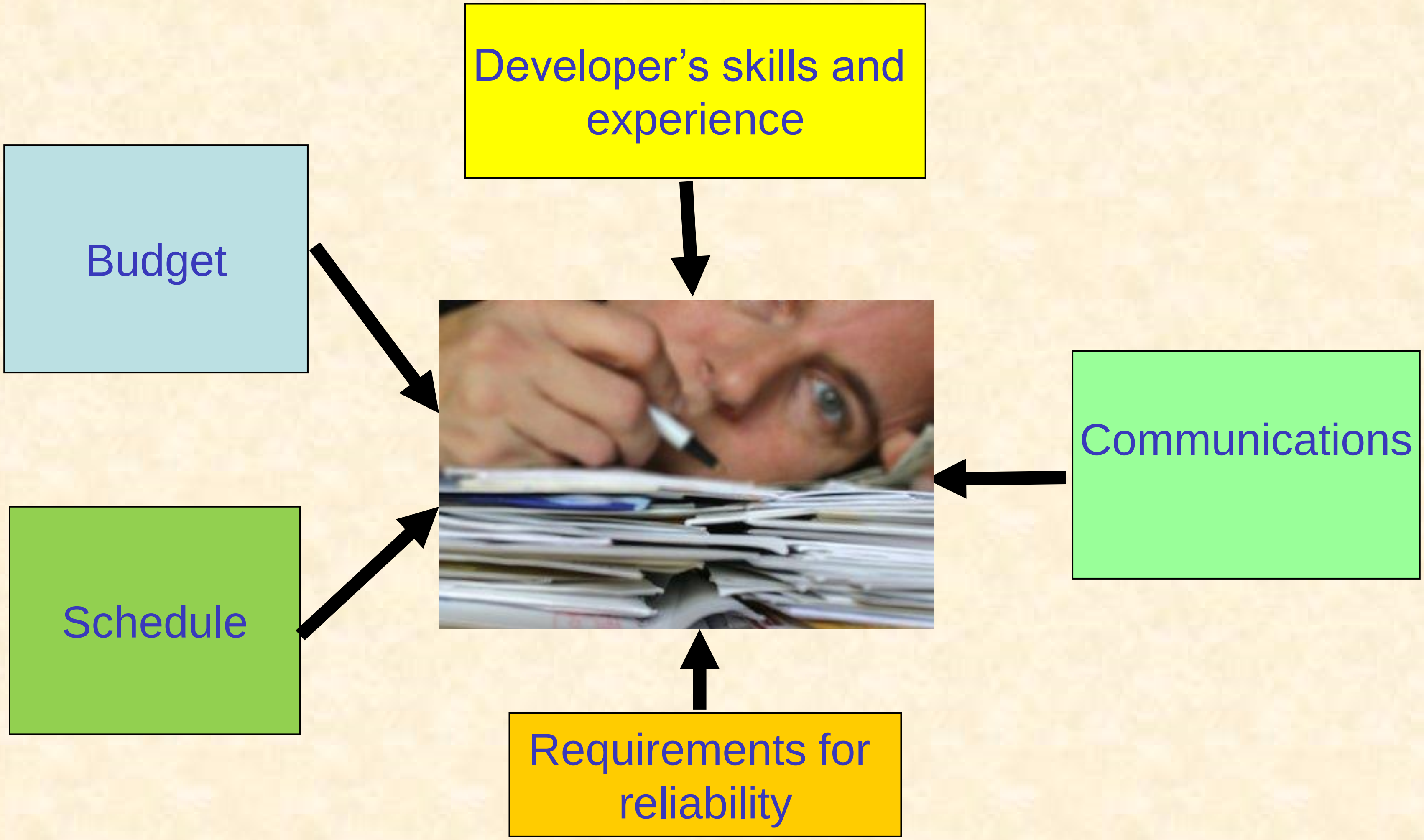
2. Formal Methods for improvement

3. Formal Engineering Methods for practicality

4. SOFL

I.1 Challenges in traditional Software Engineering

The challenges for software projects in traditional software engineering come from many aspects.



Developer’s skills and experience

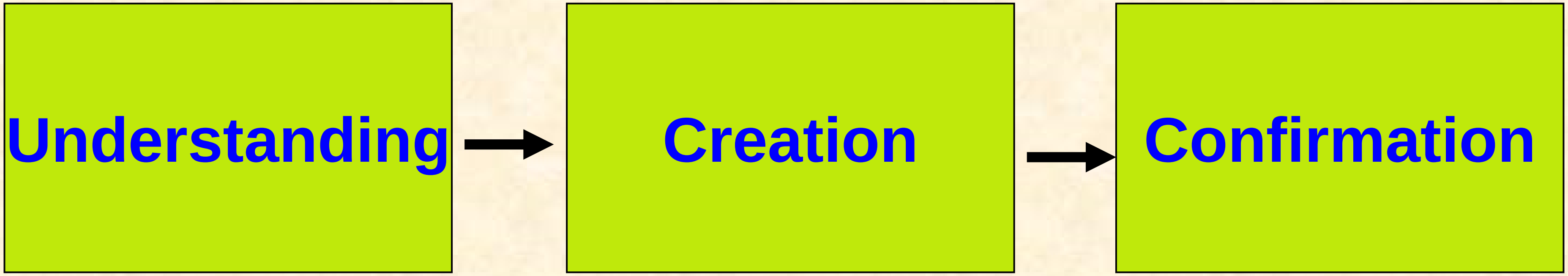
Budget

Communications

Schedule

Requirements for reliability

The major tasks in software development:



Understanding → Creation → Confirmation

Correctness &

Reliability!!!

What is a reliable system?

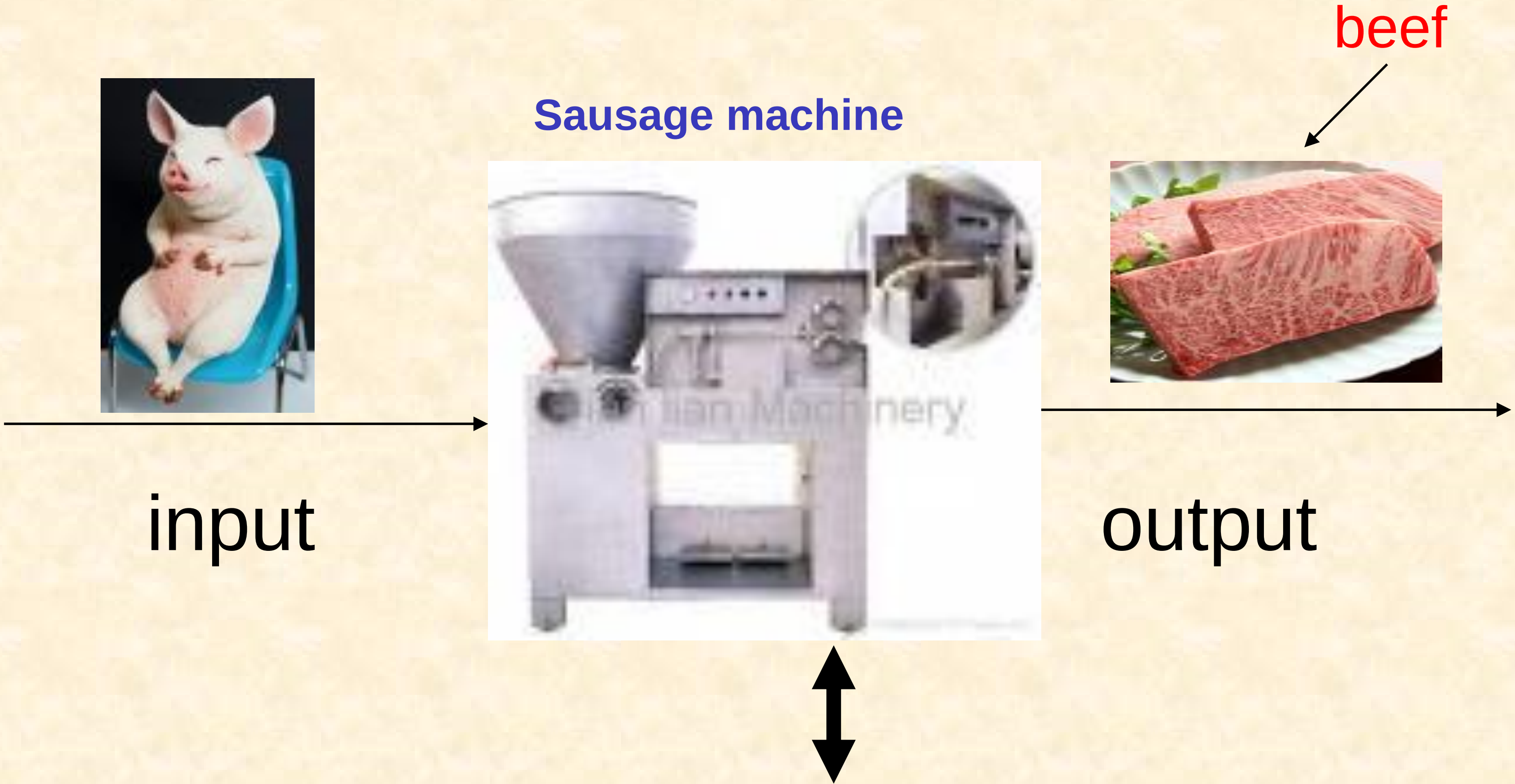
Sausage machine

input

output

Requirement: given a piglet as input, the machine will produce sausages automatically.

An example of unreliable system:



beef

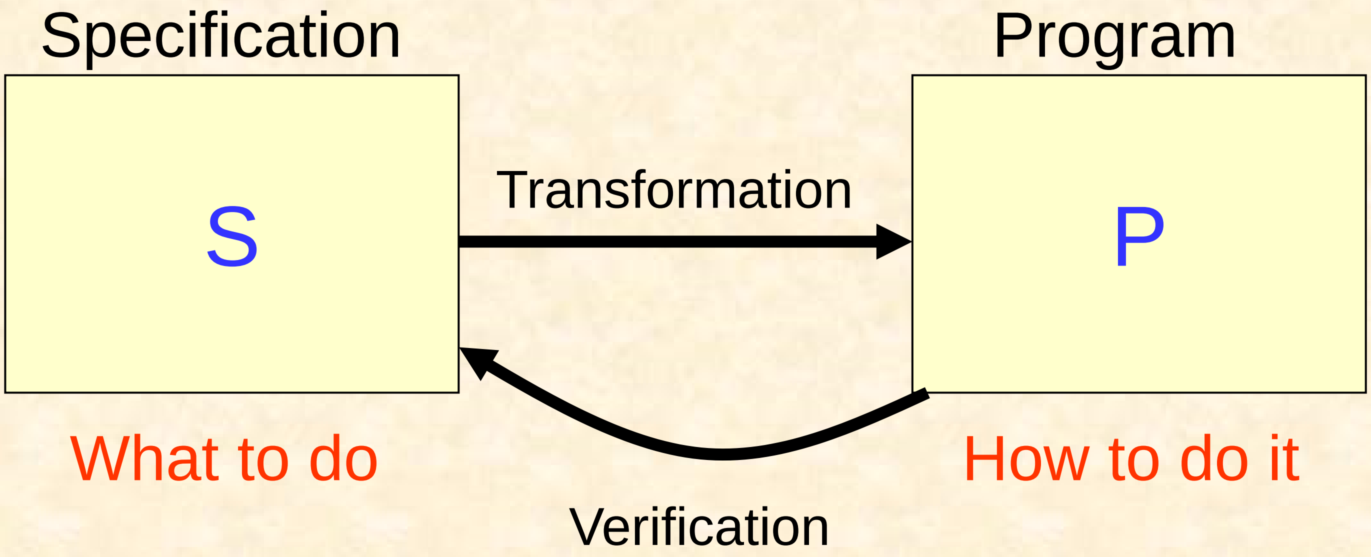
Sausage machine

input

output

Requirement: given a piglet, the machine will produce sausages automatically.

An abstract process model



- How to write S so that it can effectively help the developer understand the user’s requirements completely and accurately?
- How to ensure that S is not ambiguous so that it can be correctly understood by all the developers involved?
- How can S be effectively used for the construction of program P, and for the verification (e.g., proof, inspection, and testing) of P?
- How can software tools effectively support the analysis of S, transformation from S to P, and verification of P against S?

An example of informal specification:

“A software system for an Automated Teller Machine (ATM) is required to provide services on various accounts. The services include operations on current account, operations on savings account, transferring money between accounts, managing foreign currency account, and change password. The operations on a current or savings account include deposit, withdraw, show balance, and print out transaction records.”

A better way to write the same specification:

“A software system for an automated teller machine (ATM) is required to provide services on various accounts.

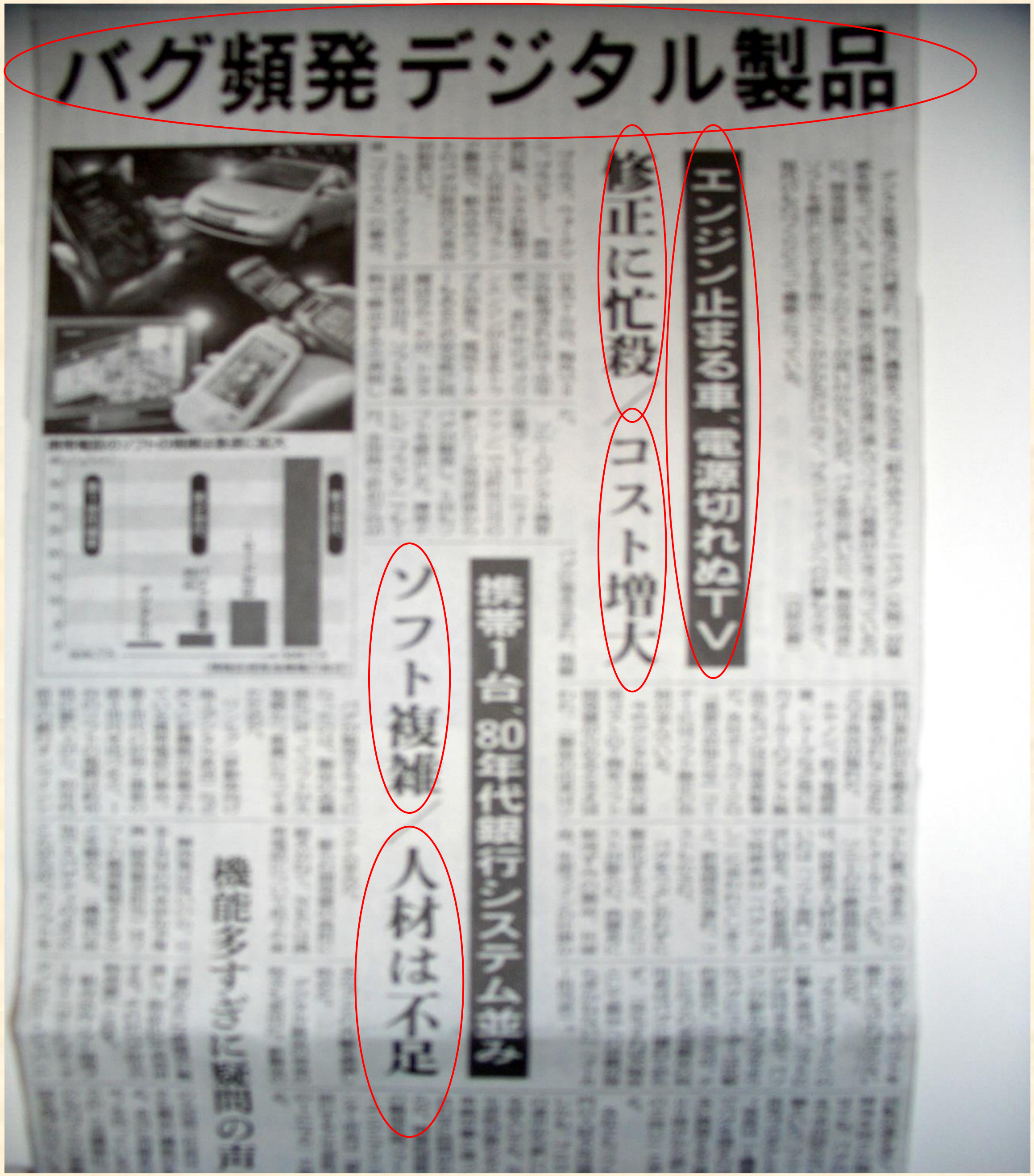
The services include
① operations on current account
② operations on savings account
③ transferring money between accounts
④ managing foreign currency account,
⑤ change password.

The operations on a current or savings account include
① deposit
② withdraw
③ show balance
④ print out transaction records.”

The major problems with informal specifications:

- Informal specifications are usually ambiguous, which is likely to cause misinterpretations.
- Informal specifications are difficult to be used for implementation and for inspection and testing of programs because of the big gap between the functional descriptions in the specifications and the program structures.
- Informal specifications are difficult to be analyzed for their consistency and validity.
- Informal specifications are difficult to be supported by software tools in their analysis, transformation, and management (e.g., search, change, reuse).

2006年
4月9日の
朝日新聞



バグ頻発 デジタル製品

エンジン止まる車、電源切れぬTV

修正に忙殺／コスト増大

携帯1台、80年代銀行システム並み

ソフト複雑／人材は不足

機能多すぎに疑問の声

Reality of software development in practice

Manager:

Why is the project over budget and behind schedule?

Client:

Why does the software system behave differently from my requirements?

Programmer:

Why are there so many bugs remaining in the program?

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

A possible solution to these problems:

Formal Methods!!!

I.2 Formal methods for improvement

Formal methods mean **verified design**.

Formal methods =

Formal Specification (VDM, Z, B-Method)

+

Refinement (Back, Morgan)

+

Formal Verification (Floyd-Hoare, Dijkstra)

↑

Set theory, logics, algebra, etc.

The most commonly used formal notations

(1) VDM-SL (Vienna Development Method – Specification Language),
IBM Research Laboratory in Vienna

References:

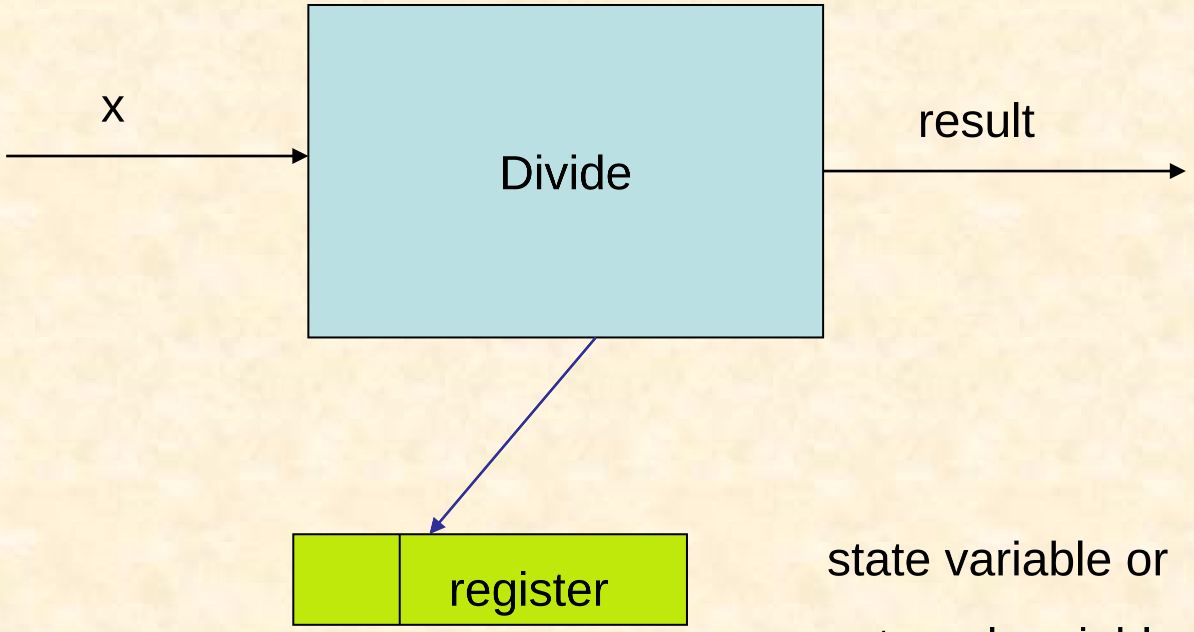
(1) "Systematic Software Development Using VDM'', by Cliff B. Jones, 2nd edition, Prentice Hall,1990.

(2) "Modelling Systems", by John Fitzgerald and Peter Gorm Larsen, Cambridge University Press,1998.

The major feature of VDM-SL is the technique for operation specification.

What is an operation?

An example of operation: Divide



x

Divide

result

register

state variable or

external variable

Operation specification:

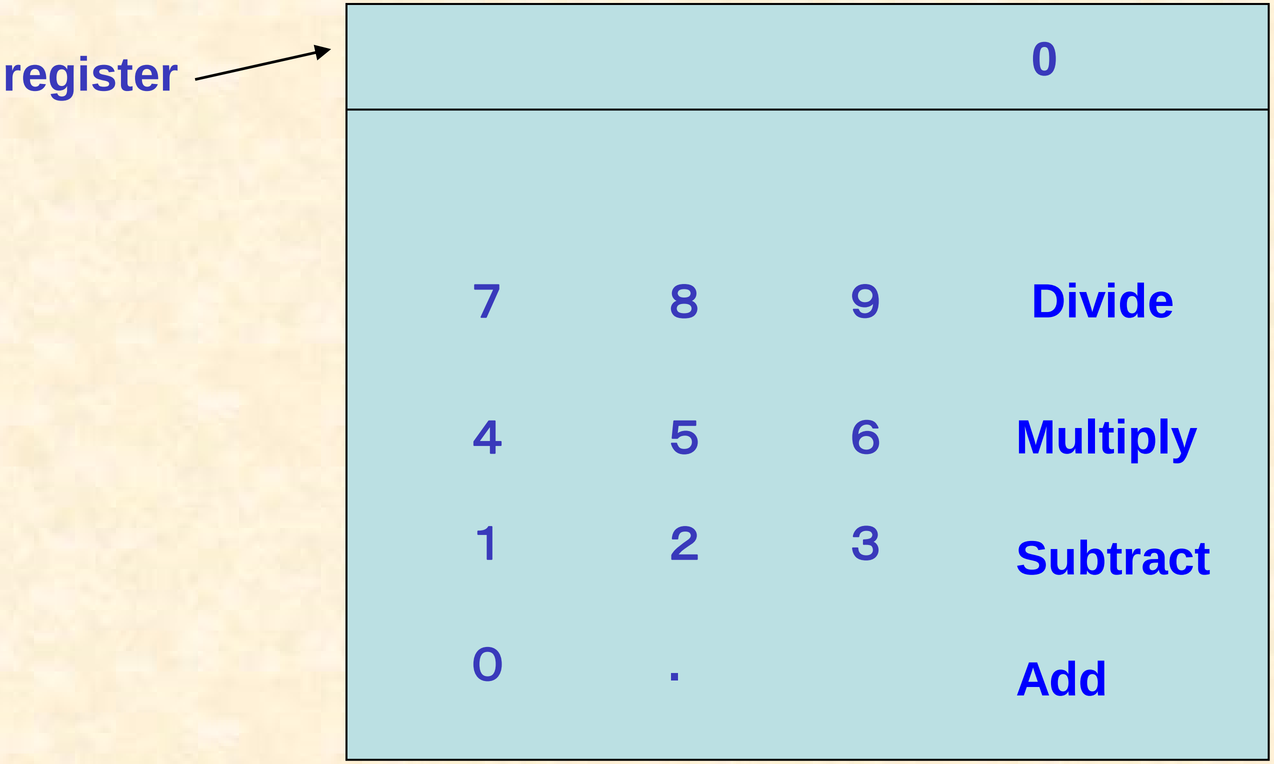
```
OperationName(input)output
ext State variables
pre   pre-condition
post  post-condition
```

Example:

```
Divide(x: int) result: real
ext wr register: real
pre x <> 0
post register = register~ / x
        and
        result = register
```

The difference and similarity between a VDM operation specification and program

An example of a simplified calculator:



register

0

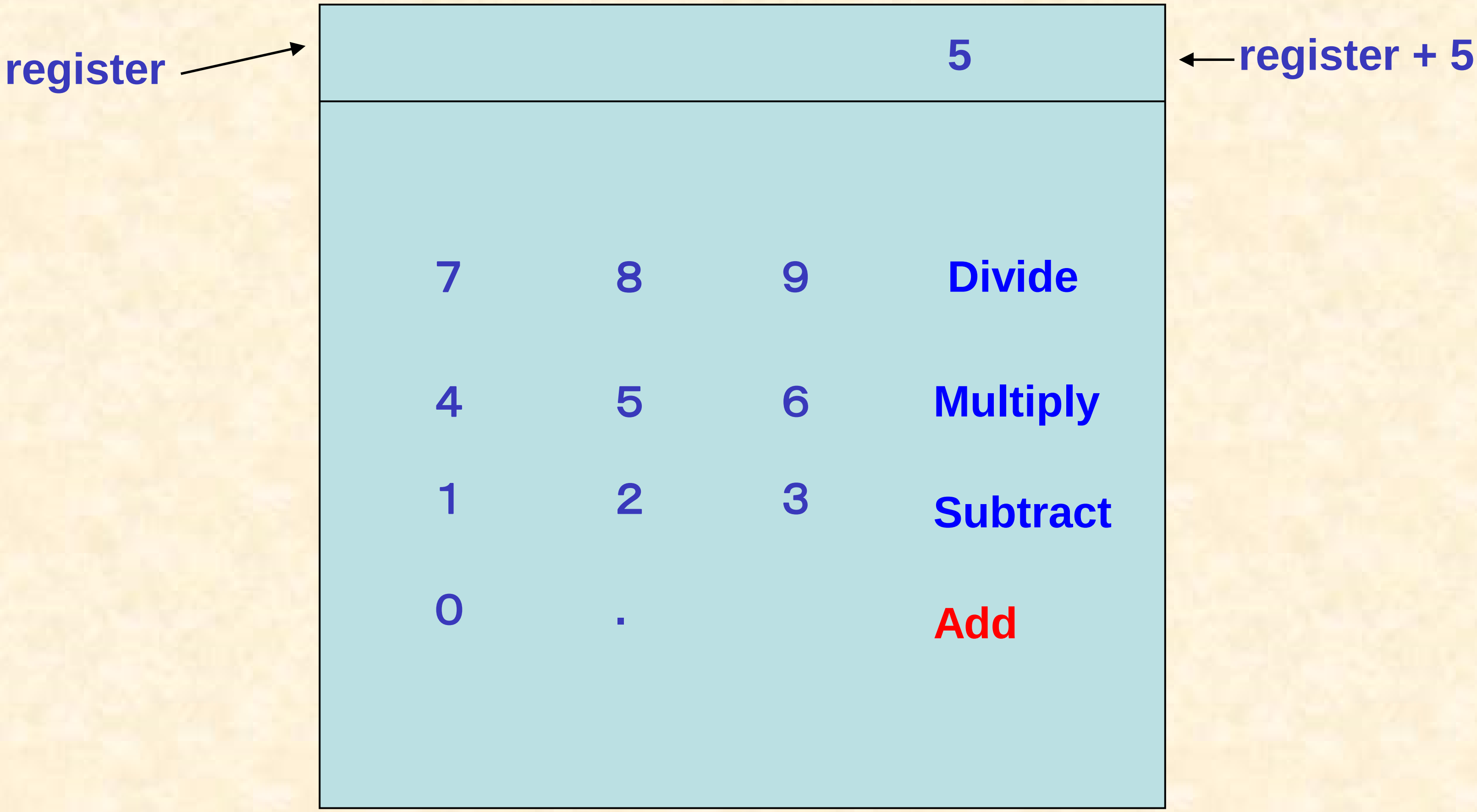
7 8 9 Divide

4 5 6 Multiply

1 2 3 Subtract

0 . Add

The difference and similarity between a VDM operation specification and program



register

5

register + 5

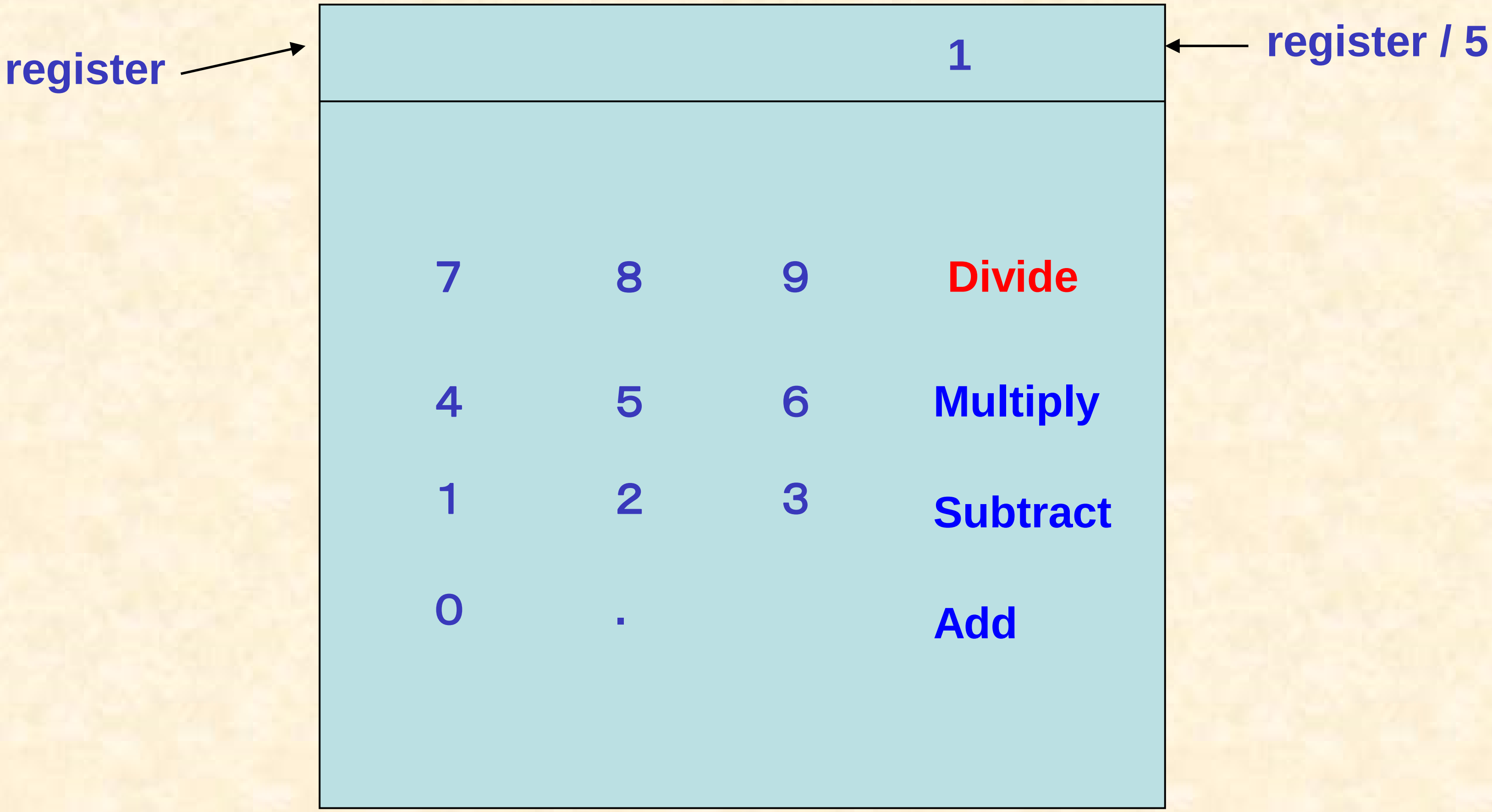
7 8 9 Divide

4 5 6 Multiply

1 2 3 Subtract

0 . Add

The difference and similarity between a VDM operation specification and program



register

1

register / 5

7 8 9 Divide

4 5 6 Multiply

1 2 3 Subtract

0 . Add

The difference and similarity between a VDM operation specification and program

Let’s consider the following program:

```
import …;

class Calculator {
 int  register := 0;

int Add(int x) {
 register := register + x;
 return register;
}

double Divide(int x) {
 register := register / x;
 return register;
}
……
}
```

```
main(String arg[]) {

 int input;

 double divisionResult;

 Calculator myCal :=

 new Calculator();

 read(input); //reading from GUI

 divisionResult :=

         myCal.Divide(input);

 System.out.println(“Division result:” +
divisionResult);

} //end of the main method
```

The difference and similarity between a VDM operation specification and program

The specification defines the relation between input and output, while the program defines the process of computing the output from the input.

```
import …;

class Calculator {
 int  register := 0;

……

double Divide(int x)
pre x != 0
{
 register := register / x;
 return register;
}
post register = register~ / x and
       result = register
}
```

```
main(String arg[]) {
 int input;
 double divisionResult;
 Calculator myCal :=
 new Calculator();
 read(input); //reading from GUI
 divisionResult :=
     myCal.Divide(input);
 System.out.println(“Division result:” +
divisionResult);
} //end of the main method
```

The difference and similarity between a VDM operation specification and program

Omit the program and only use the specification to define the function of the program.

```
import …;

class Calculator {
 int  register := 0;


……

double Divide(int x)
pre x != 0
post register~ / x = register
       and
       register = result
}
```

```
main(String arg[]) {
 int input;
 double divisionResult;
 Calculator myCal :=
 new Calculator();
 read(input); //reading from GUI
 if (input != 0) {
 divisionResult := myCal.Divide(input);
 System.out.println(“Division result:” + divisionResult);}
 else
 System.out.println(“The denominator is zero.”);
} //the end of the main method.
```

The difference and similarity between a VDM operation specification and program

Specification in VDM-SL:

```
module Calculator

……

Divide(x: int) result: real
ext wr register: real
pre x <> 0
post register = register~ / x
       and
       result = register
```

```
main(String arg[]) {
 int input;
 double divisionResult;
 Calculator myCal :=
 new Calculator();
 read(input); //reading from GUI
 if (input != 0) {
 divisionResult := myCal.Divide(input);
 System.out.println(“Division result:” +
divisionResult);}
 else
 System.out.println(“The denominator is zero.”);
} //the end of the main method.
```

(2) Z, PRG (Programming Research Group),
Oxford University, UK

A Z specification is composed of a set of schemas and possibly their sequential compositions. A schema can be used to define global variables, state variables, and operations.

References:

(1) "The Z Notation", by J.M. Spivey,
Prentice Hall, 1989.

(2) "Using Z: Specification, Refinement, and Proof'', by Jim Woodcock and Jim Davies,
Prentice Hall, 1996.

(3) B-Method,

Jean-Raymond Abrial, France

Reference:

(1) "The B-Book: Assigning Programs to Meanings",
by J-R Abrial, Cambridge University Press,1996,

A B specification is composed of a set of related abstract machines. Each abstract machine is a module that contains a set of operation definitions. Each operation is defined using pre- and post-conditions.

Class dicsussion

(1) Tell the difference between the assignment

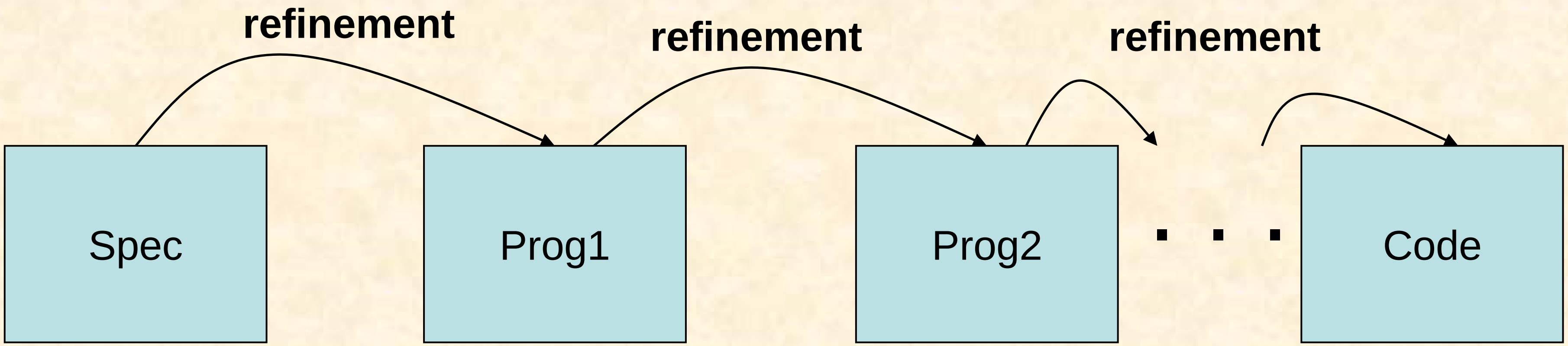
$x := y + z$ (or $x = y + z$ in Java)

in a program and the equality in mathematics:

$x = y + z$

(2) What is the relation between them?

Software development by refinement



refinement

refinement

refinement

Spec

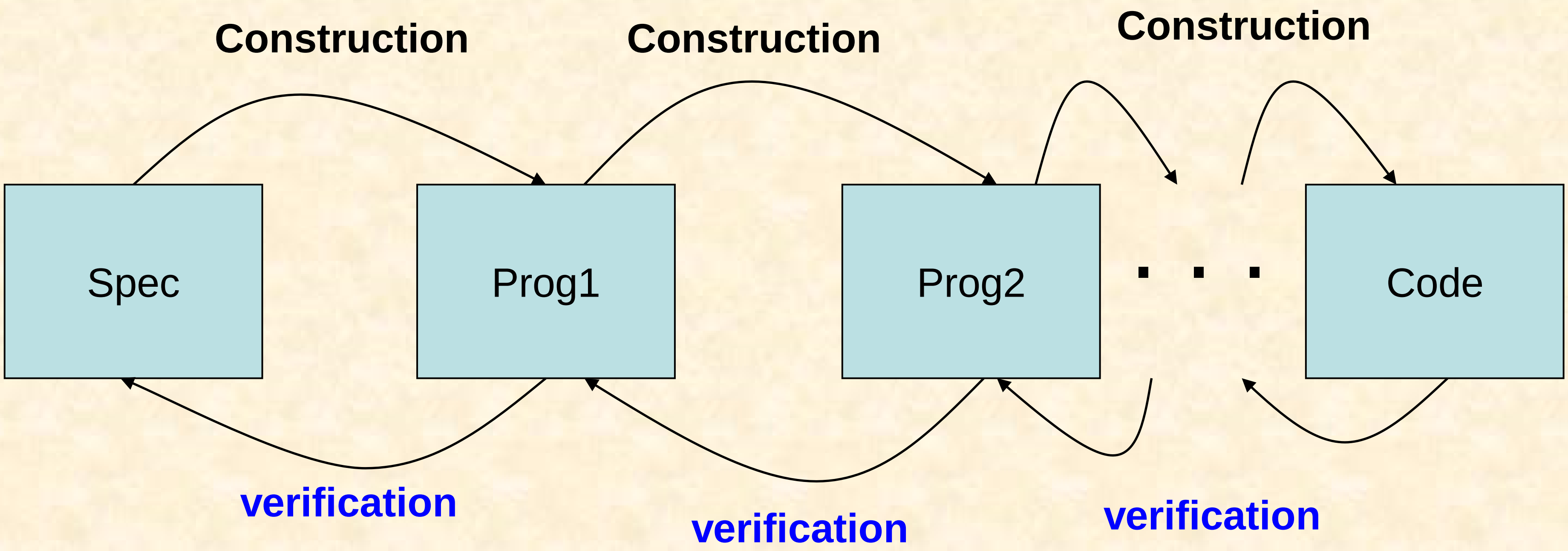
Prog1

Prog2

. . .

Code

Software development based on formal verification



Application examples of formal methods

1The company Praxis in the UK has used VDM, CSP, and Z to develop a toolset for the **SSADM** development method.

2. The IBM Hursley in the UK has collaborated with the PRG of Oxford university to apply Z to the development of a **Customer Information Control System**（CICS）.

3. The Darlington Nuclear Generator System (Ontario Hydro) in Canada has used Parnas’ SCR (Software Cost Reduction) to verify the Shutdown System.

4. The company CSK in Japan has used VDM to develop a stock processing system.

5. The company FeliCa has applied VDM++ to the development of a mobile phone IC chip.

Challenges for formal methods

- The complexity of formal specifications grows rapidly as their scale increases. This makes evolution of specifications, which is an necessary activity in real projects, very hard and costly.
- Formal methods only offer notations and rules, but not tell how each kind of technique (e.g., specification, refinement, verification) can be effectively applied in the context of practical software development.

- Formal verification for program correctness can only be applied to small and simple programs. There is a lack of effective techniques for dealing with large scale systems.
- The tool support is not good enough to make formal methods practical in industry.

- There are many practical constraints as well.

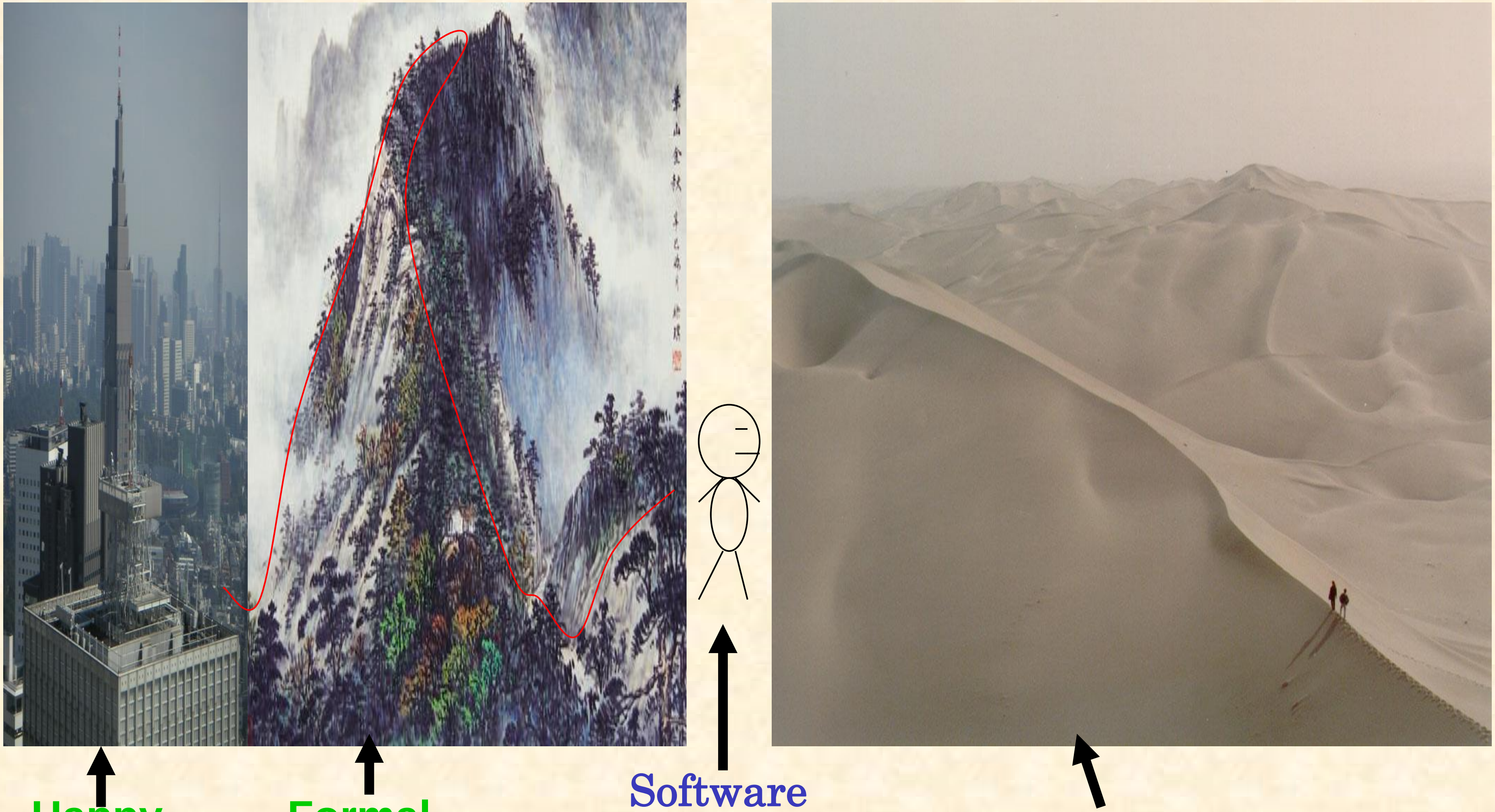
Examples:

(1) Practitioners may lack the skills for abstraction in writing formal specifications.

(2) Managers may not want to introduce formal methods before seeing hard evidence of the effectiveness of formal methods.

(3) Budget, schedule, and company’s environment may not allow practitioners to use formal methods.

Software Practitioners’ dilemma



Happy world

Formal Methods

Software practitioner

Traditional SE

I.3 Formal Engineering Methods for Practicality

Formal Engineering Methods (FEM) provide ways to integrate Formal Methods into the commonly used software engineering methods and approaches to improve their rigor, comprehensibility, effectiveness, and tool supportability for software productivity and quality.

Formal Methods

Application of Formal Methods in Software Engineering

Formal Engineering Methods

The difference between FM and FEM

FM answers the question:

what should we do and why?

FEM answers the question:

what can we do and how?

I.4 SOFL

SOFL stands for Structured Object-Oriented
Formal Language

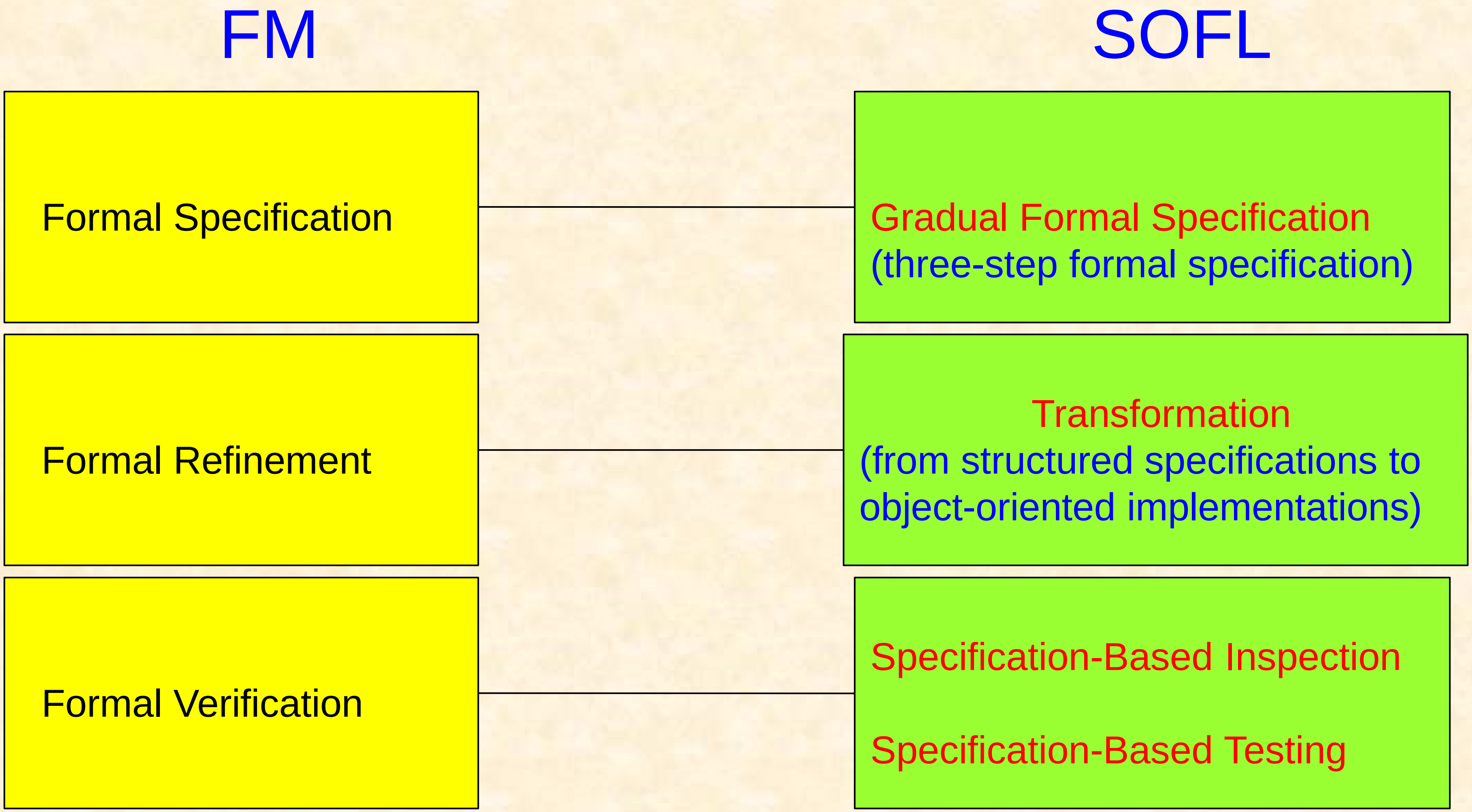
SOFL method is a specific and representative formal engineering method. The research on it started at the University of Manchester, UK in 1989.

Completed at Hiroshima City University in 1998.

Finalized at Hosei University in 2002.

Further developments of SOFL technologies after 2002.

Comparison between FM and the SOFL method



FM

SOFL

Formal Specification

Gradual Formal Specification
(three-step formal specification)

Formal Refinement

Transformation
(from structured specifications to object-oriented implementations)

Formal Verification

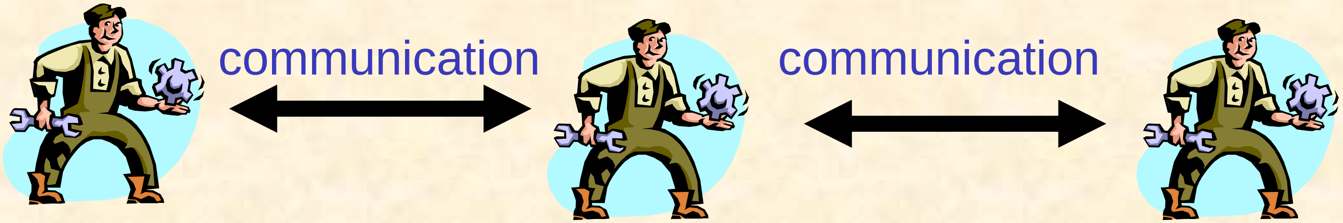
Specification-Based Inspection

Specification-Based Testing

The feature of SOFL

The SOFL method strikes a good balance among the three qualities:

Simplicity, Visualization, Precision

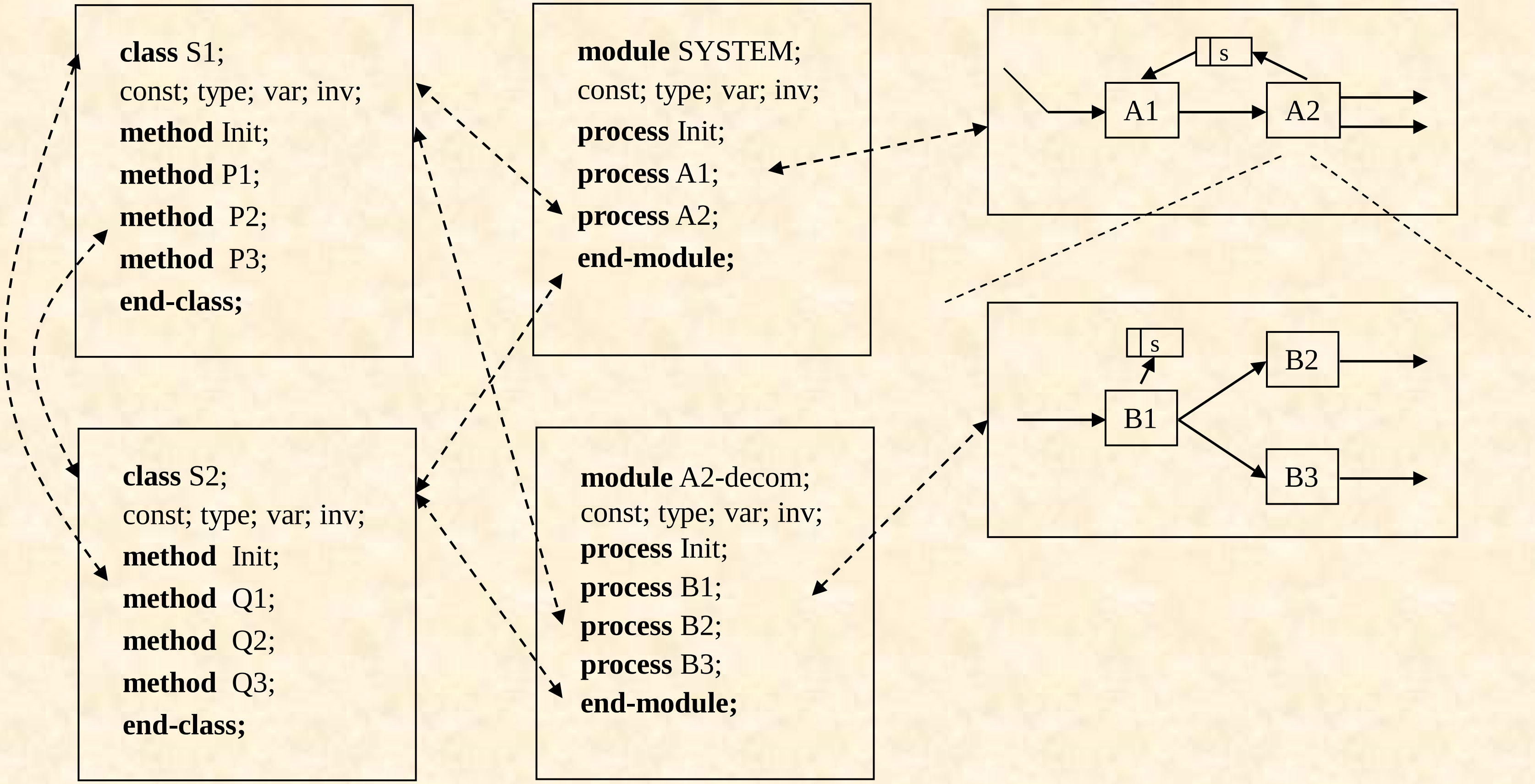


communication

communication

The structure of a SOFL specification:

CDFDs + modules + classes

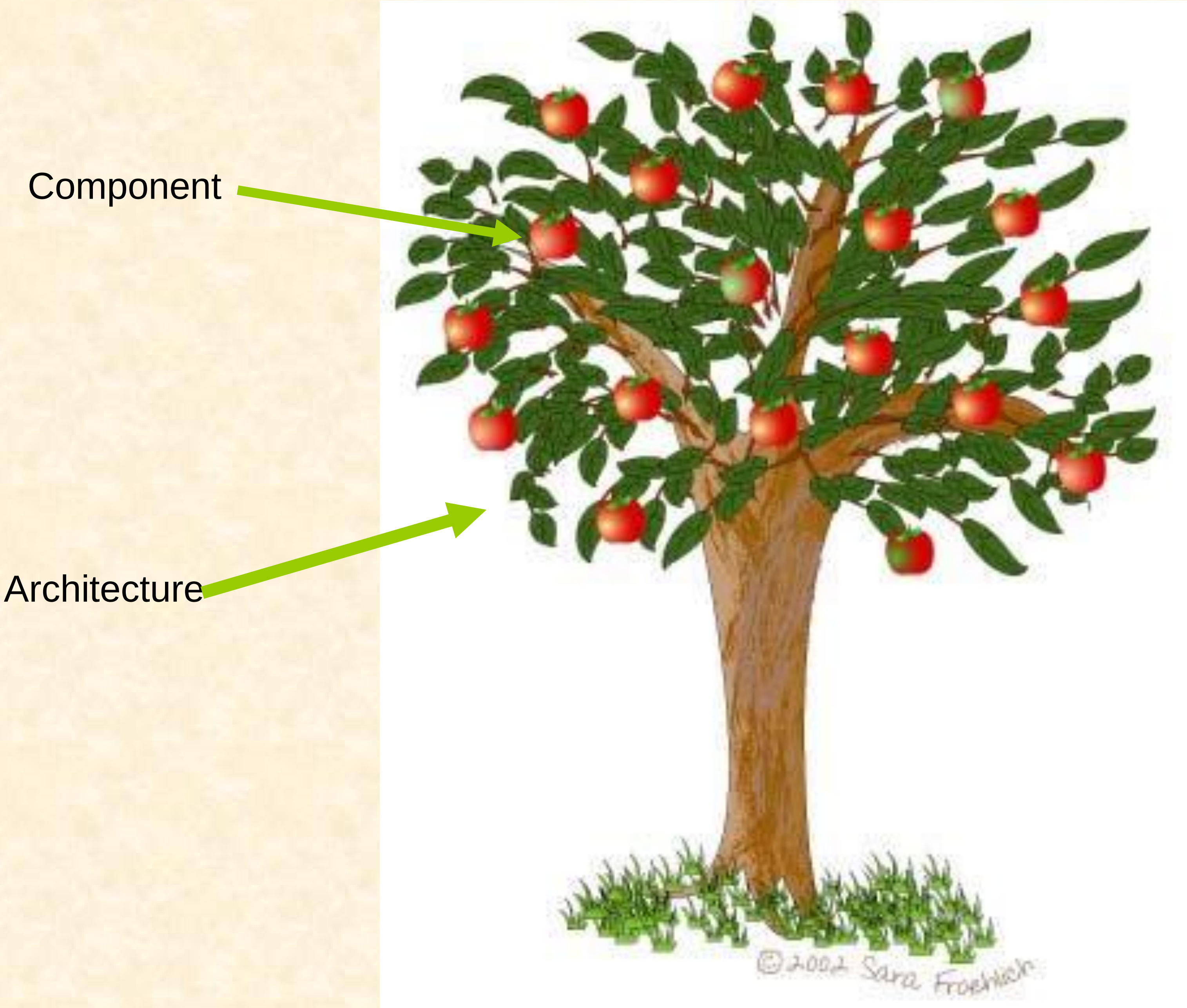


class S1;
const; type; var; inv;
method Init;
method P1;
method P2;
method P3;
end-class;

module SYSTEM;
const; type; var; inv;
process Init;
process A1;
process A2;
end-module;

class S2;
const; type; var; inv;
method Init;
method Q1;
method Q2;
method Q3;
end-class;

module A2-decom;
const; type; var; inv;
process Init;
process B1;
process B2;
process B3;
end-module;



Component

Architecture

Basic Components of the SOFL Specification Language

- The SOFL logic
- Module
- Condition Data Flow Diagrams
- Process specification
- Function definition and specification
- Data types
- Process decomposition
- Other issues

II.1 SOFL logic

SOFL logic is an extension of classical propositional logic and predicate logic; it allows “undefined” as a logical value

(SOFL adopts the three-value logic used in VDM).

II.1.1 Propositional logic

Definition: A proposition is a statement that is either true or false.

For example, the following statements are propositions:

(1) Tiger is animal (true)

(2) Apple is fruit (true)

(3) $3 + 5 > 10$ (false)

In contrast, the following statements are not propositions:

(1) Are you happy?

(2) Let's go swimming

(3) x := y + 3 (assignment statement)

Definition: The value true and false are called truth value.

In SOFL we use bool to represent the boolean type that contains the truth values, that is:

bool = {true, false}

Propositions are represented by symbols:

(1) P: Tiger is animal.
(2) Q: Apple is fruit.
(3) R: $3 + 5 > 10$.

Such a proposition is called atomic proposition (which cannot be decomposed).

Propositions can be connected using logical operators to form propositional expressions (or compound propositions) that describe more complex propositions.

Propositional operators

operator	read as	priority
not	not	highest
and	and	
or	or	
=>	implies	
<=>	is equivalent to	lowest

Conjunction

Definition: A conjunction is a propositional expression whose principal operator is and.

For example:

$x > 5$ and $x < 10$

Question: How to decide the truth value of a conjunction?

Truth table for conjunction

P1	P2	P1 and P2
true	true	true
true	false	false
false	true	false
false	false	false

Examples:

true and true <=> true
false and true <=> false
false and false <=> false

Disjunction

Definition: A disjunction is a propositional expression whose principal operator is or.

P1 or P2

For example:

$x > 5$ or $x < 3$

P1	P2	P1 or P2
true	true	true
true	false	true
false	true	true
false	false	false

Negation

Definition: A negation is a propositional expression whose principal operator is not.

not P1

Example:

not x > 5

P1	not P1
true	false
false	true

Implication

Definition: An implication is a propositional expression whose principal operator is =>.

P1 => P2

P1	P2	P1 => P2
true	true	true
true	false	false
false	true	true
false	false	true

Example:

x > 10 => x > 5

In this case we can also say that x > 10 is stronger than x > 5.

Equivalence

Definition: An equivalence is a propositional expression whose principal operator is <=>.

P1 <=> P2

P1	P2	P1 <=> P2
true	true	true
true	false	false
false	true	false
false	false	true

Examples:

(1)John is Chris' friend <=> Chris is John's friend

(2) x > 10 <=> not x = 10 and not x < 10

The use of parentheses

An expression is interpreted by applying the operator priority order unless parentheses are used.

For example: the expression

not p and q or r <=> p => q and r

is equivalent to the expression:

(((not p) and q) or r) <=> (p => (q and r))

Parentheses can be used to change the precedence of operators in expressions. For example, the above expression can be changed to:

not (p and ((q or (r <=> p))=> q) and r)

Tautology, contradiction, and contingency

Definition: A tautology is a proposition that evaluates to true in every combination of the truth values of its constituent propositions.

Examples:

(1) P or not P
(2) $x > 10$ or $x <= 10$

Definition: A contradiction is a proposition that evaluates to false in every combination of the truth values of its constituent propositions.

In other words, a contradiction is a negation of a tautology.

Examples:

(1) P and not P

(2) $x > 10$ and $x < 10$

Definition: A contingency is a propositional expression that is neither a tautology nor a contradiction.

In other words, a contingency can evaluate to either true or false.

Examples:

(1) P and Q (P and Q are not related with each other)

(2) $x > 5$ or $x < -5$

Normal forms

Definition: A disjunctive normal form is a special kind of disjunction in which each constituent propositional expression, is a conjunction of atomic propositions or their negations.

Form:

P_1 or P_2 or ... or P_n

and

P_i = Q_i_1 and Q_i_2 and … and Q_i_m

where i = 1,…,n and Q_i_j (j = 1, …., m) is an atomic proposition or negation of an atomic proposition.

The characteristic of a disjunctive normal form:

It evaluates to true as long as one of the constituent expression evaluates to true.

II.1.2 Predicate logic

The propositional logic only allows to make statements about specific objects, but it does not allow us to make universal statements and existential statements.

For example, the following are universal statements:

(1) Every student of Hiroshima University is happy.

(2) Nobody knows what to happen tomorrow.

The following are existential statements:

(1) One of my classmates received an award.

(2) Some students in my class do not like mathematics.

Predicates

Definition: A predicate is a truth-valued function.

In other words, a predicate is a function from a set X to the boolean type bool:

$$p: X \rightarrow \text{bool}$$

For example:

$x > 10$ is a predicate, but not proposition.

$5 > 10$ is a proposition, derived from the predicate $x > 10$ by substituting 5 for x.

where x is an integer variable.

Basic types (sets) in SOFL

The following are basic types in SOFL:

nat0: 0, 1, 2, 3, 4, … (natural numbers including 0)

nat: 1, 2, 3, 4, 5, … (natural numbers)

int: … -2, -1, 0, 1, 2, …(integers)

real: …-2.5, -1.4, 0, 1.4, 2.5,(real numbers)

char: ‘a’, ‘b’, ‘x’, ‘%’, … (characters)

bool: true, false (boolean values)

Example: a predicate ``is_big” is defined as follows:

```
is_big(x: int): bool
== x > 10
```

Then, the following propositions can be formed:

is_big(10)   (false)

is_big(15)   (true)

is_big(9)   (false)

Quantifiers

(1) Universal quantifier

For example:

is-big(x) == x > 10 (is-big is a predicate)

Then we can write the conjunction

is-big(12) and is-big(15) and is-big(20)

as

forall[x: {12, 15, 20}] | is-big(x)

In general, the universally quantified expression has the form:

forall[x1: X1, …, xn: Xn] | p(x1, x2, …, xn)

forall --- universal quantifier
xi: Xi (i =1,...,n) --- bindings
x1, x2, …, xn --- bound variables
X1, X2, …, Xn --- the ranges (sets or types) of the bound variables
p(x1, x2, …, xn) --- predicate

(2) Existential quantifier

For example, we can write the disjunction

is-big(5) or is-big(12) or is-big(15)

as

exists[x: {5, 12, 15}] | is-big(x)

We call such an expression existentially quantified expression.

exists![x: T] | p(x)

means that there exists a unique x in T that satisfies condition p(x).

In general, the existentially quantified expression has the form:

exists[x1: X1, …, xn: Xn] | p(x1, x2, …, xn)

exists --- existential quantifier

xi: Xi (i=1,...,n) --- bindings

x1, x2, …, xn --- bound variables

X1, X2, …, Xn --- the ranges (sets or types) of the bound variables

p(x1, x2, …, xn) --- predicate

(3) The convention

The body of a quantified expression is considered to extend as far as the right possible.

Example: the quantified expression

forall[x: nat] | (x > z and (exists[y: nat] | y > x))

is equivalent to the expression:

forall[x: nat] | x > z and exists[y: nat] | y > x

Multiple quantifiers

Examples:

```
forall[x: X] | forall[y: Y] | p(x,y)
```

can be written as:

```
forall[x: X, y: Y] | p(x, y)
```

forall[x: X] | exists[y: Y] | p(x, y)

can be written as:

forall[x: X]exists[x: Y] | p(x, y)

----

exists[x: X] | exists[y: Y] | p(x, y)

can be written as:

exists[x: X, y: Y] | p(x, y)

Examples of multiple quantifiers

```
forall[i: nat] exists[j: nat] | j  > i
```

This predicate is true, but the inversion of the universal quantifier and the existential quantifier will change the truth of the expression. Consider the following quantified expression, which is not true.

```
exists[j: nat] forall[i: nat] | j > i
```

Treatment of partial predicates

Definition: If a predicate may not yield a truth value for some values bound to its free variables, we call the predicate partial predicate.

For example,

$$x / 0 > 5$$

is a partial predicate.

The problem is that predicate logic does not allow undefined “value” to join evaluation of predicates.

One way to deal with this problem is to extend the truth tables of all the logical operators (i.e., and, or, not, =>, <=>) to allow the special value "undefined" to participate in evaluations of predicates. We use nil to denote "undefined".

The extension is made in the way that a result is given whenever possible, according to the predicate logic.

(and)	true	nil	false
true	true	nil	false
nil	nil	nil	false
false	false	false	false

(or)	true	nil	false
true	true	true	true
nil	true	nil	nil
false	true	nil	false

(not)	
true	false
nil	nil
false	true

Examples:

nil and false <=> false

nil and true <=> nil

nil and nil   <=> nil

(=>)	true	nil	false
true	true	nil	false
nil	true	nil	nil
false	true	true	true

(<=>)	true	nil	false
true	true	nil	false
nil	nil	nil	nil
false	false	nil	true

Class exercise 1

Use predicate expressions to describe the following statements:

(1) Every integer is greater than 0, equal to 0, or less than 0.

(2) For any three real numbers *a*, *b*, and *c*, if *a* is greater than *b* and *b* is greater than *c*, then a will be greater than *c*.

(3) For any natural number *a* there must exist another natural number *b* such that *b* is greater than *a*.

II.2 The Module

Definition: A module is a functional abstraction: it has a behavior represented by a condition data flow diagram (CDFD), and a structure to define data items and processes occurring in the condition data flow diagram. Each data item is defined with an appropriate type and each process is defined with a formal, textural notation using the SOFL logic.

Module for abstraction

An effective way to gain the understanding of system function is

abstraction and decomposition

Definition: Abstraction is a principle of extracting the most important information from implementation details.

The result of an abstraction is usually a concise specification of the system reflecting all the primarily important functions without unnecessary details.

Example of an ATM functional abstraction

(1) Provide the functions of showing balance and withdraw for selection.

(2) Insert a cash-card and supply a password.

(3) If showing balance is selected, the current balance of the bank account is given.

(4) If withdraw is selected, the requested amount of money is properly provided.

Abstraction may have different levels:

For example, if we refine function (4) in the previous abstraction, we get a refinement (concrete version):

(4’) If withdraw is selected and the password is correct, the requested amount of the money is provided; otherwise, if the password is wrong, a message for reentering the correct password is given.

We can refine (4’) further to get the following concrete version of the functional description by considering how to deal with the situation that the requested amount to be withdrawn is greater than the balance of the account:

(4’’) If withdraw is selected, the password is correct, and the requested amount is less than the balance of the account, the money of the requested amount will be provided. Otherwise, if either the password is wrong or the requested amount is greater than the balance, an appropriate message is provided.

Question?

How to express functional abstractions so that they are precise, comprehensible, easy to be verified and validated, and easy to be transformed into programs?

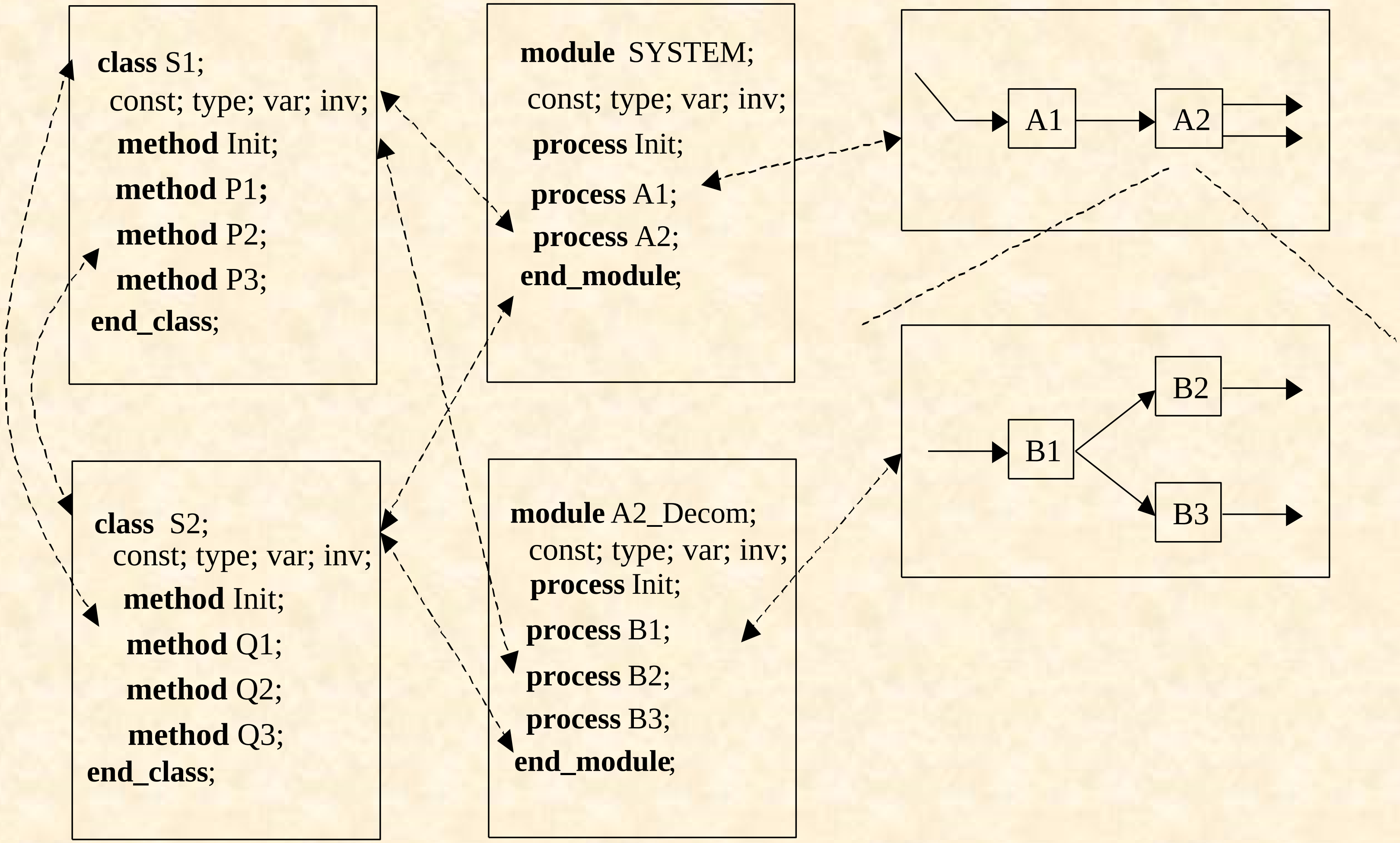
In SOFL we use module for functional abstraction.

Conceptually a module has the following structure:

ModuleName
condition data flow diagram
Specification of the components

Specifically, a module has the following structure in general:

The general structure of a SOFL specification



class S1;
const; type; var; inv;
method Init;
method P1**;**
method P2;
method P3;
end_class;

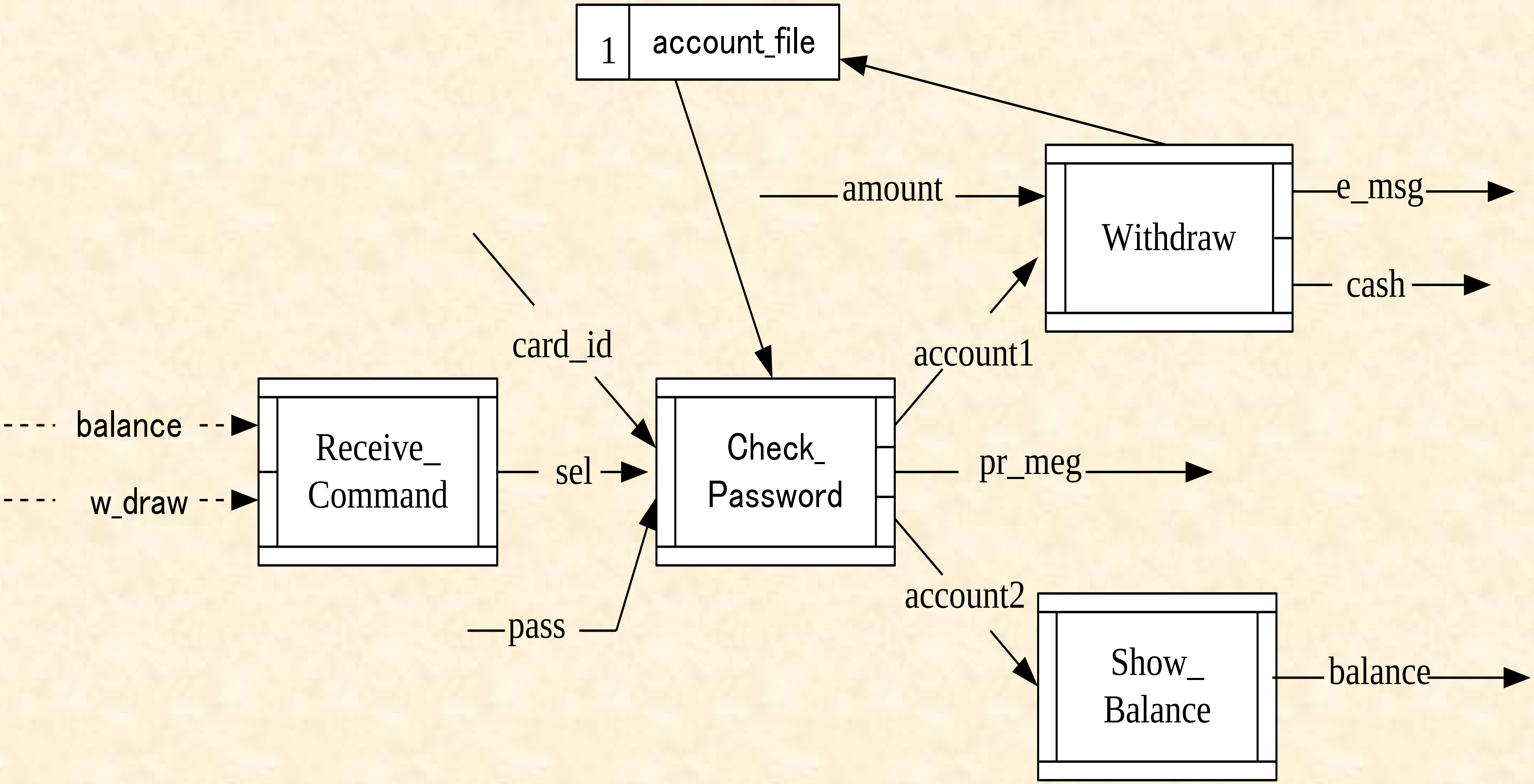
module SYSTEM;
const; type; var; inv;
process Init;
process A1;
process A2;
end_module;

class S2;
const; type; var; inv;
method Init;
method Q1;
method Q2;
method Q3;
end_class;

module A2_Decom;
const; type; var; inv;
process Init;
process B1;
process B2;
process B3;
end_module;

```
module ModuleName / UpperLevelModule;
  const ConstantDefinition;
  type   TypeDefinition;
  var     VariableDefinition;
  inv      TypeAndStateInvariants;
  behav    CDFD_Figure No.;

  process Init(); /* initialize the local store variables of the
    module. This process can be omitted if there is no local state
    variable defined in the var section.*/
  process_1;
  process_2;
  …
  process_n;
  function_1;
   …
  function_m;
 end-module;
```

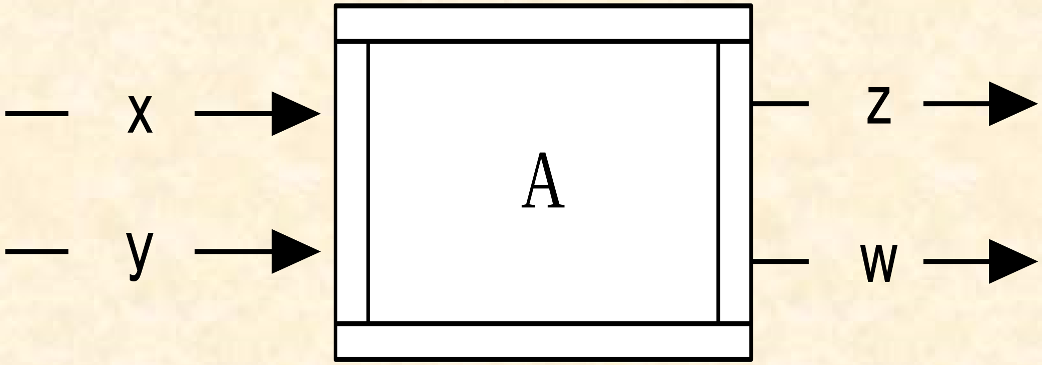
Condition Data Flow Diagrams (CDFD)



Process

A process models a transformation from input to output. It is similar to a VDM Operation, a procedure in Pascal, or a method in Java.

Graphical representation:



The components of a process:

name (A)
input port (receiving x and y)
output port (sending z and w)
precondition (indicated by the narrow rectangle at the top)
postcondition (indicated by the narrow rectangle at the bottom)

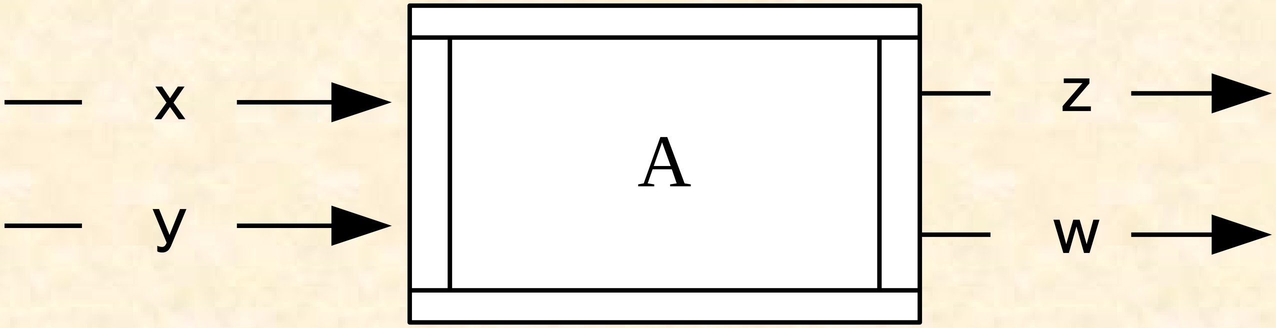
The meaning of process A:

1. when both the input data flows x and y are available, the process is enabled, but it will not execute until the output data flows z and w become unavailable.

2. the execution of the process consumes the input data flows x and y, and generates the output data flows z and w.

The formal specification of process A:

```
process A(x: Ti_1, y: Ti_2) z: To_1, w: To_2
pre P(x, y)
post Q(x, y, z, w)
end_process
```



A concrete specification of process A can be:

```
process A(x: int, y: int) z: real, w: int
pre x >= y
post z**2 = x – y and w > z
comment
z is a square root of x – y and w is greater than z.
end_process
```

or

```
process A(x, y: int) z: real, w: int
pre x >= y
post z**2 = x – y and w > z
end_process
```

A process specification with no specific precondition or postcondition:

```
process A(x, y: int) z, w: int
pre true
post z = x + y and w = x - y
end_process
```

```
process A(x, y: int) z, w: int
pre x > 0 and y > 0
post true
end_process
```

A process specification with no specific requirements (we call it choose):

```
process A(x, y: int) z, w: int
pre  true
post true
end_process
```

or with the simplified expression by omitting the pre and postconditions:

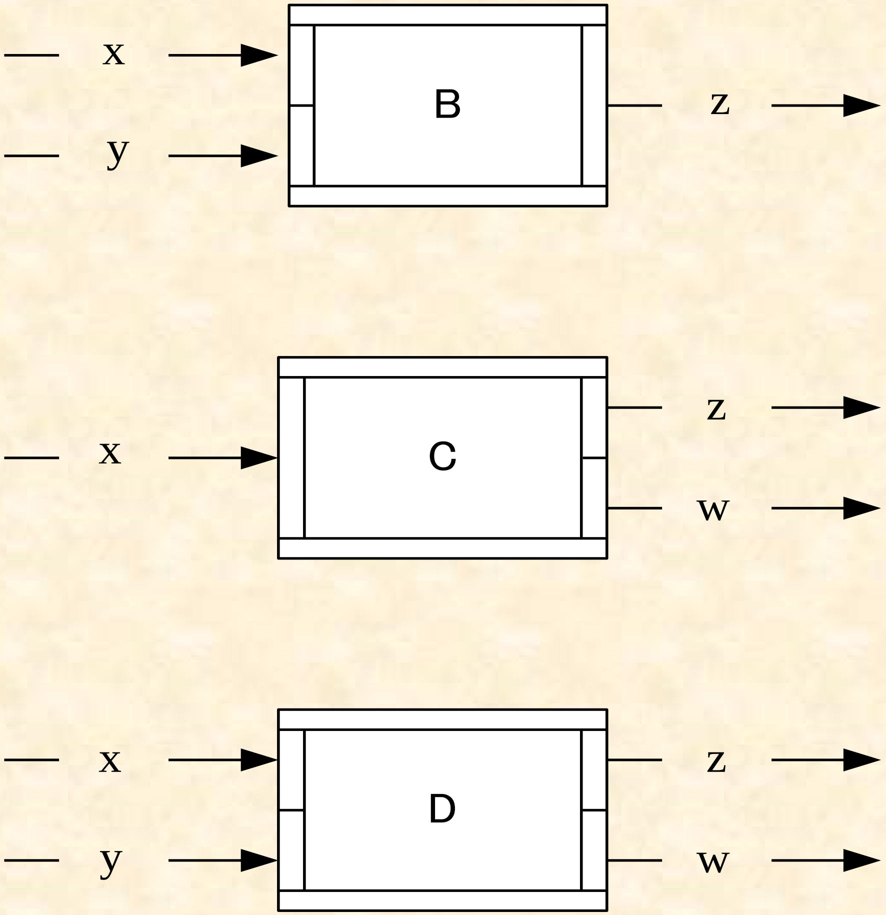
```
process A(x, y: int) z, w: int
end_process
```

Class discussion

When a programmer is required to implement the following specification, what do you think the programmer should do?

```
process A(x, y: int) z, w: int
 pre  true
 post true
 end_process
```

Processes with multiple ports



Specifications of process B

```
process B(x: int | y: int) z: real
pre x <> 0 or y >= 0
post z >= (x**2 + 1) / x        or
        z**2 >= y and z >= 0
end_process
```

The following specification is inappropriate:

```
process B(x: int | y: int) z: real
pre x <> 0 and y >= 0
post z >= (x**2 + 1) / x and
        z**2 >= y and z >= 0
end_process
```

Another possibility of process B

```
process B(x: int | y: int) z: int
pre  x > 0 or bound(y)
post z = x + 1 or z = y - 1
end_process
```

where bound(y) is a predicate (not a truth value) defined as follows:

bound(y) = true if y is available (i.e., y <> nil).

bound(y) = false if y is unavailable (i.e., y = nil).

Specifications of process C

```
process C(x: int) z: real | w: int
pre  x > 0
post z = (x**2 + 1) / x     or
        w**2 >= x and w > 0
end_process
```

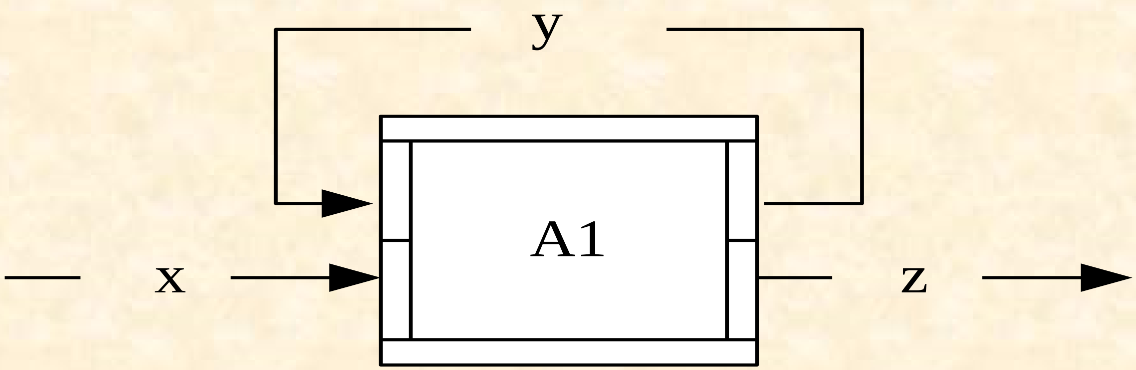
This specification does not tell exactly which of z and w will be generated as the result of an execution of process C. A more deterministic specification is:

```
process C(x: int) z: real | w: int
pre   x > 0
post x < 10 and z = (x**2 + 1) / x or
        x >= 10 and w**2 >= x and w > 0
end_process
```

The specification of process D

```
process D(x: Ti_1 | y: Ti_2) z: To_1 | w: To_2
 pre  P1(x) or P2(y)
 post bound(x)  and Q_1(z, x) or
        bound(y) and  Q_2(y, w)
 end_process
```

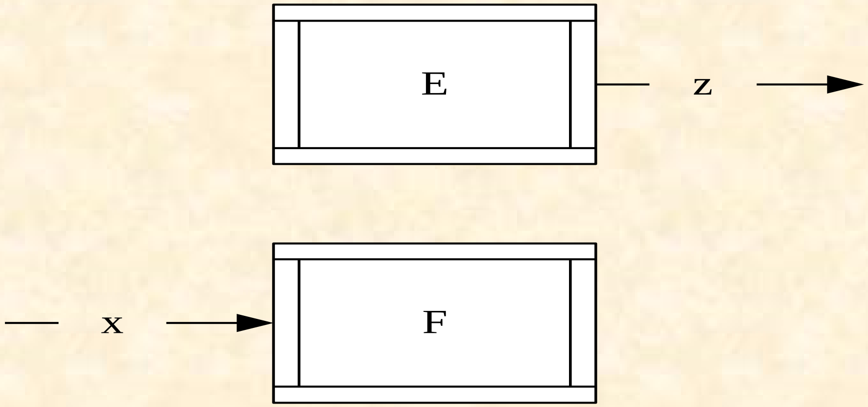
The specification of a process with a data flow loop



```
process A1(y: nat0 | x: nat0) y: nat0 | z: nat0
pre  x = 0 or bound(y)
post y = x + 1 or
       ~y < 100 and y = ~y + 1 or ~y >= 100 and z = ~y
end_process
```

In the postcondition, the decorated variable ~y denotes the input data flow y, while y denotes the output data flow y.

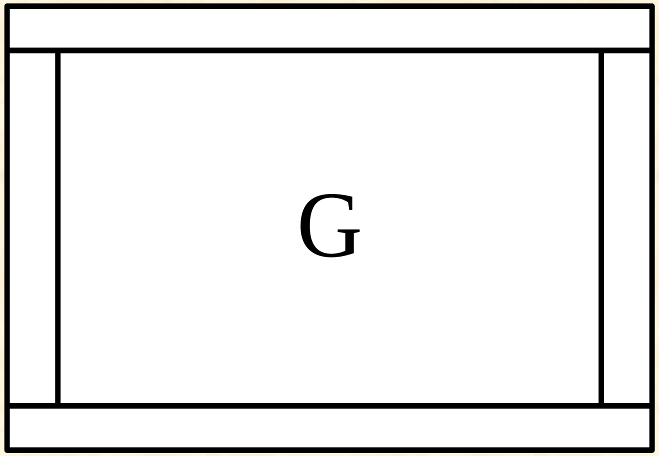
A process may have no input or output data flow



```
process E() z: nat0
pre  true
post z > 10
end_process
```

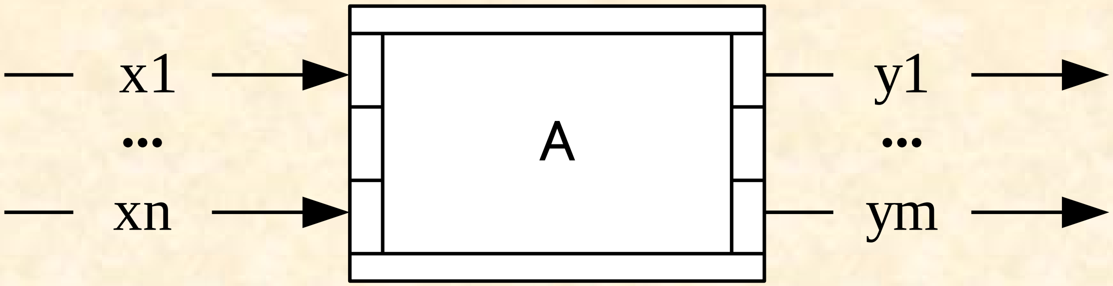
```
process F(x: nat0)
pre  x > 5
post true
end_process
```

A process with no input and output data flows is illegal.



The reason is that such a process does not provide any useful functionality.

The general form of a process



```
process A(x1_dec | x2_dec | ... | xn_dec)
                  y1_dec | y2_dec | ... | ym_dec
pre      P(x1, x2, ..., xn)
post     Q(x1, x2, ..., xn, y1, y2, ..., ym)
end_process
```

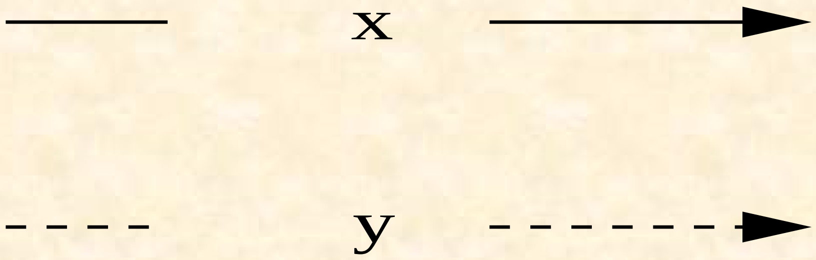
Each xi_dec (i =1..n) is a set of input variable declarations separated by comma, such as:

xi_1: Ti_1, xi_2: Ti_2, ..., xi_n: Ti_n

where xi_1, xi_2, ..., xi_n are the data flow variables connecting to input port xi, and Ti_1, Ti_2, ..., Ti_n are their types, respectively.

Data flows

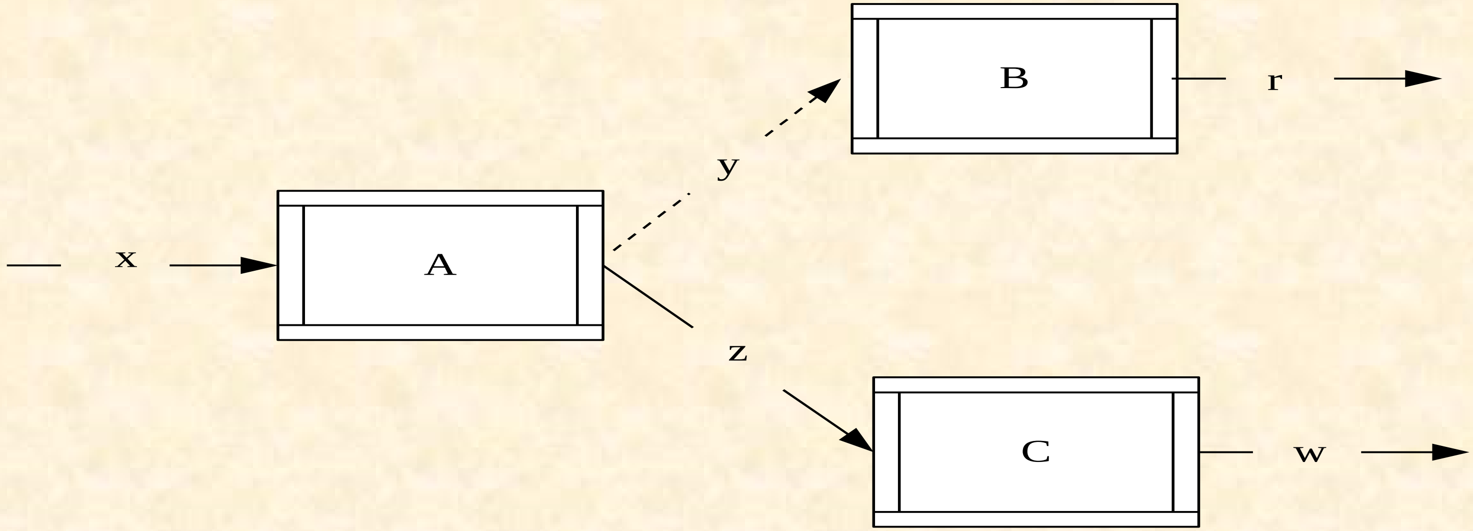
A data flow represents a data transmission from one process to another.



A data flow has a name, denoted by an identifier, and indicates the direction in which the data are transmitted.

Two kinds of data flows are available for use. One is called active data flow, such as x, and another is called control data flow, such as y.

An example showing the necessity of the two kinds of data flows

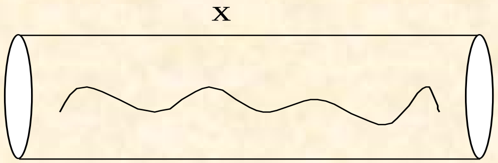


Active data flow: (1) provide useful value, (2) enable processes.
Control data flow: (1) enable processes.

In fact, a data flow name is a variable, not necessarily represents a specific value. When it is bound to a value, we say the variable is defined or available.

Data flow availability

Definition: Let x be a data flow variable of type T. Then, x is defined or available if a value of T is bound to x. Otherwise, x is undefined or unavailable.



In general, a data flow variable is declared with a type in the form:

x: T

Special type for a control data flow variable

A control data flow variable must be declared with the special type: sign, which means signal.

sign = {!}

An active data flow must not be declared with the type sign.

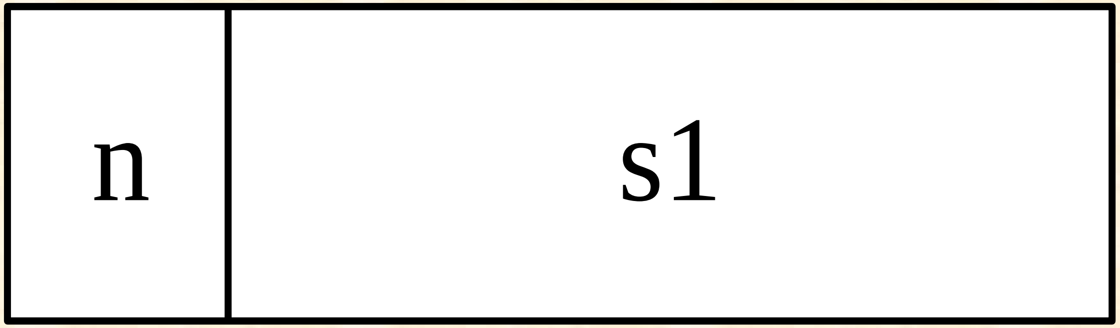
Expression of an available data flow

Let x be a data flow variable. Then, that x is available can be expressed using any one of the following two expressions:

- bound(x)

Data stores

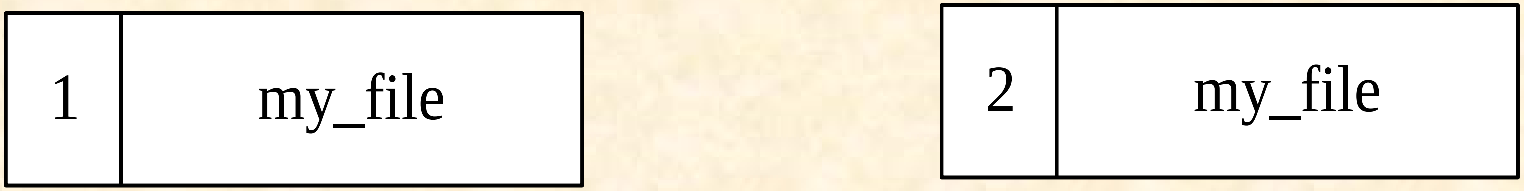
Definition: A data store, or store, is a variable holding data in rest.



s1 is the name of the store.

n is the number of the store, which may be useful in distinguishing stores with the same name.

For example, suppose the following two stores are designed by different persons, but they are used in the same specification.



1 my_file

2 my_file

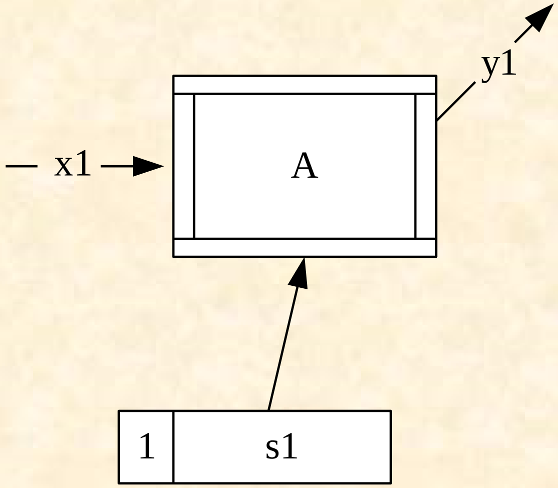
To distinguish them, we may use the following names to represent these two stores in the formal specification:

my_file_1 --- the store on the left
my_file_2 --- the store on the right

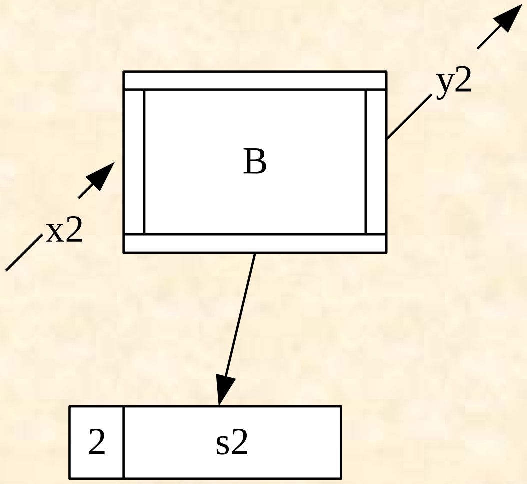
The characteristics of stores

- A store is passive; it does not actively send any data item to any process, but always makes its value ready for any related process to read and write.
- A store can only be connected, by directed lines, to processes. Syntactically, the directed lines from or to a store can only connected to either the bottom or top edge of the graphical symbol of a process. It cannot be connected to data flows or other data stores.
- A store can be either read or written (updated) by a process, which is represented by a directed line pointing to the process from the store or pointing to the store from the process.

For example,



(1)

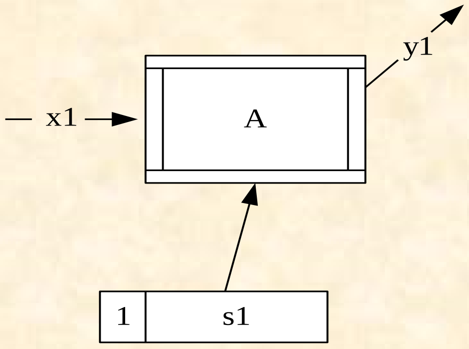


(2)

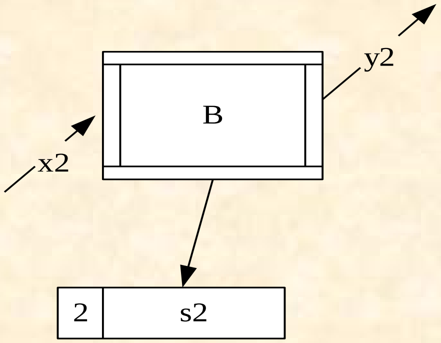
Process A reads data from store s1, which is called an external variable of process A.

Process B writes data to store s2, which is an external variable of process B.

Formal specification of a process connecting to a store



(1)



(2)

```
process A(x1: int) y1: int
ext rd s1: int
pre  x1 > 0 and s1 > x1
post y1 = s1 - x1
end_process
```

```
process B(x2: int) y2: int
ext wr s2: int
pre  x2 > 0
post y2 = ~s2 + x2 and
        s2 = ~s2 - x2
end_process
```

Decorated state variables in the postcondition:

~s2 denotes the value of variable s2 before the execution of process B. Such a value is known as the initial value of variable s2.

s2 denotes the value of variable s2 after the execution of process B. Such a value is called the final value of variable s2.

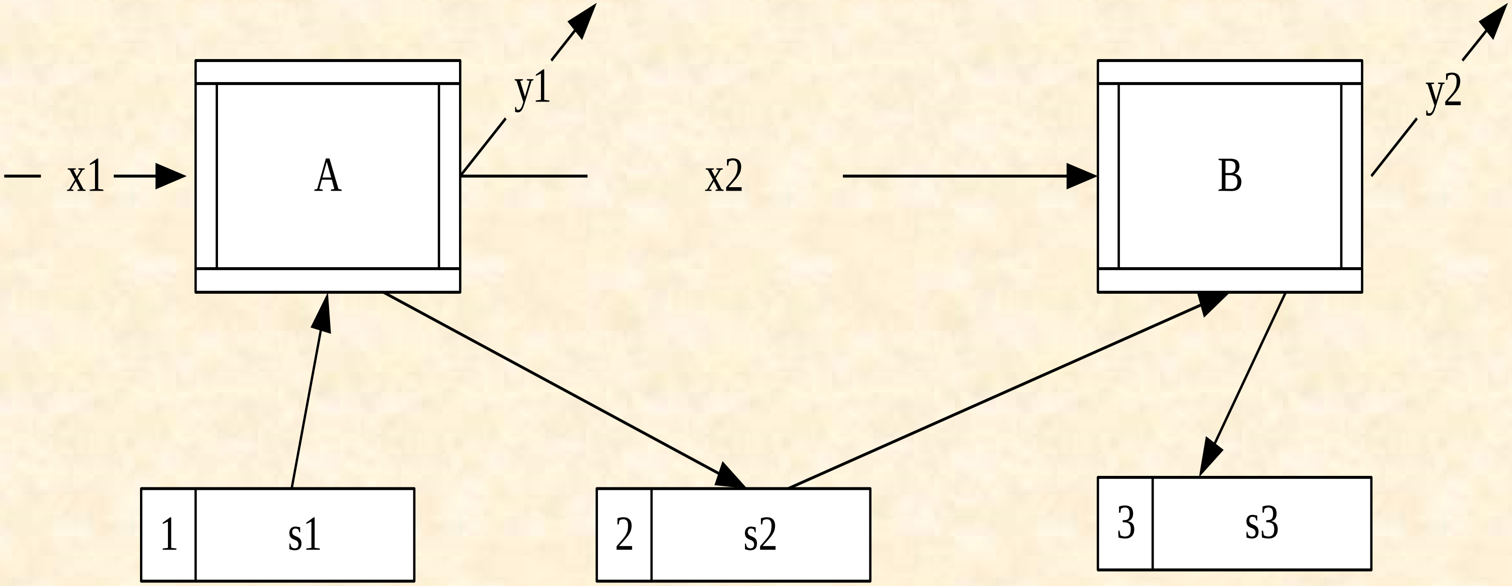
Convention: if a state variable is rd type of variable, then in the postcondition we use the non-decorated variable to denote both the initial value and final value of the variable, because they are the same in this case.

Class discussion

In the following specification, is it possible for external variable s2 to be involved in the pre-condition?

```
process B(x2: int) y2: int
ext wr s2: int
pre  x2 > 0
post y2 = ~s2 + x2 and
        s2 = ~s2 - x2
end_process
```

Multiple connections between processes and stores



The general structure of a process specification

```
process A(x_1: Ti_1 | x_2: Ti_2 | ... | x_n: Ti_n)
              y_1: To_1| y_2: To_2 | ... | y_m: To_m
 ext acc_1 z_1: Te_1
     acc_2 z_2: Te_2
       ...
     acc_q z_q: Te_q
 pre      P(x_1, x_2, ..., x_n, z_1, z_2, ..., z_q)
 post     Q(x_1, x_2, ..., x_n, y_1, y_2, ..., y_m,
              ~z_1, ~z2, ...,  ~z_q, z_1, z_2, ..., z_q)
 end_process
```

Convention for names

The names of processes, data flows, and stores are denoted by identifiers that should indicate their potential meanings for readability.

An identifier is a string of

- English letters
- digits
- underscore mark

but the first character must be a letter.

An identifier is case sensitive, so
Student_1 is different from student_1.

- The name of a process is usually written with an upper case letter for the first character of each English word and lower case letters for the rest of characters. If more than one English word are involved in a name, those words are separated by the underscore mark.

  Example: Receive_Command, Check_Password

- The name of a data flow or store is usually written using lower case letters for all the characters.

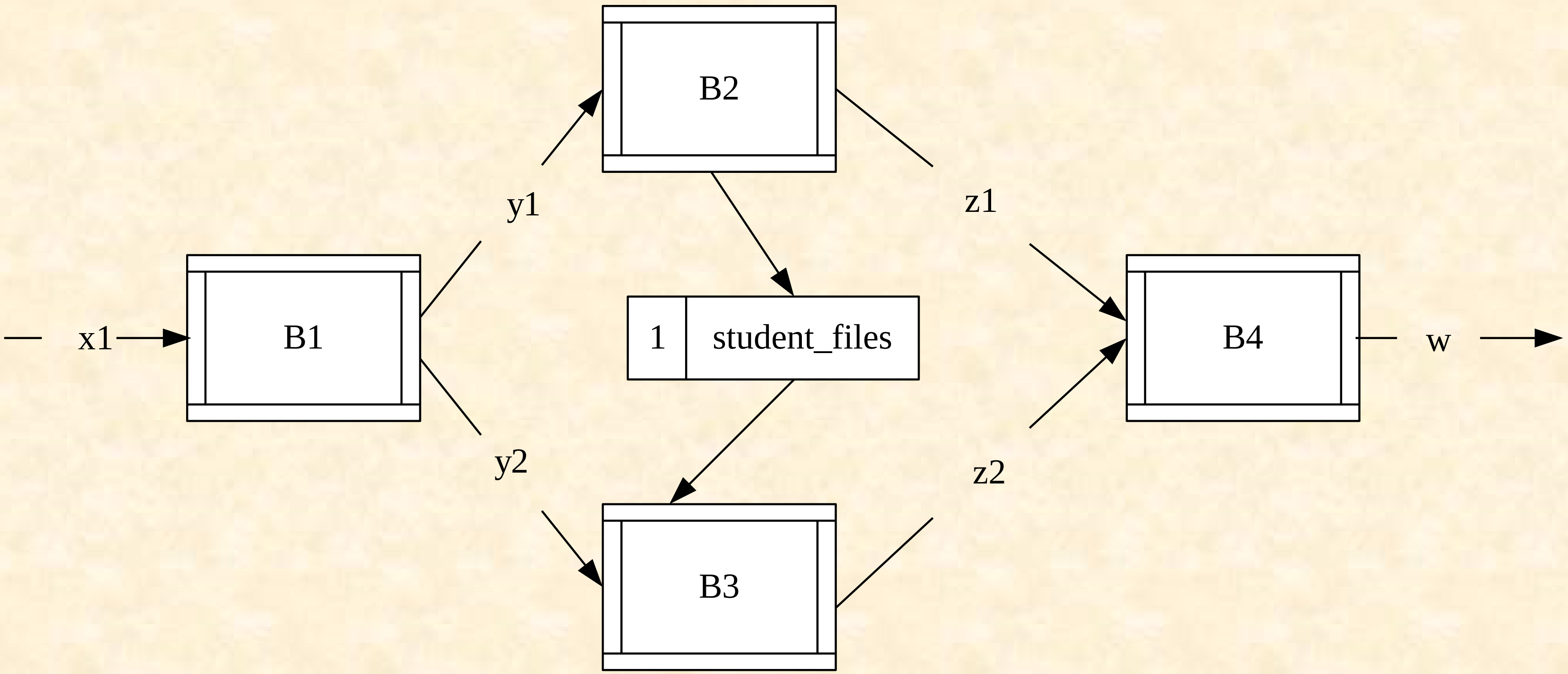
  Example: card_id, pass, w_draw

Restriction on parallel processes

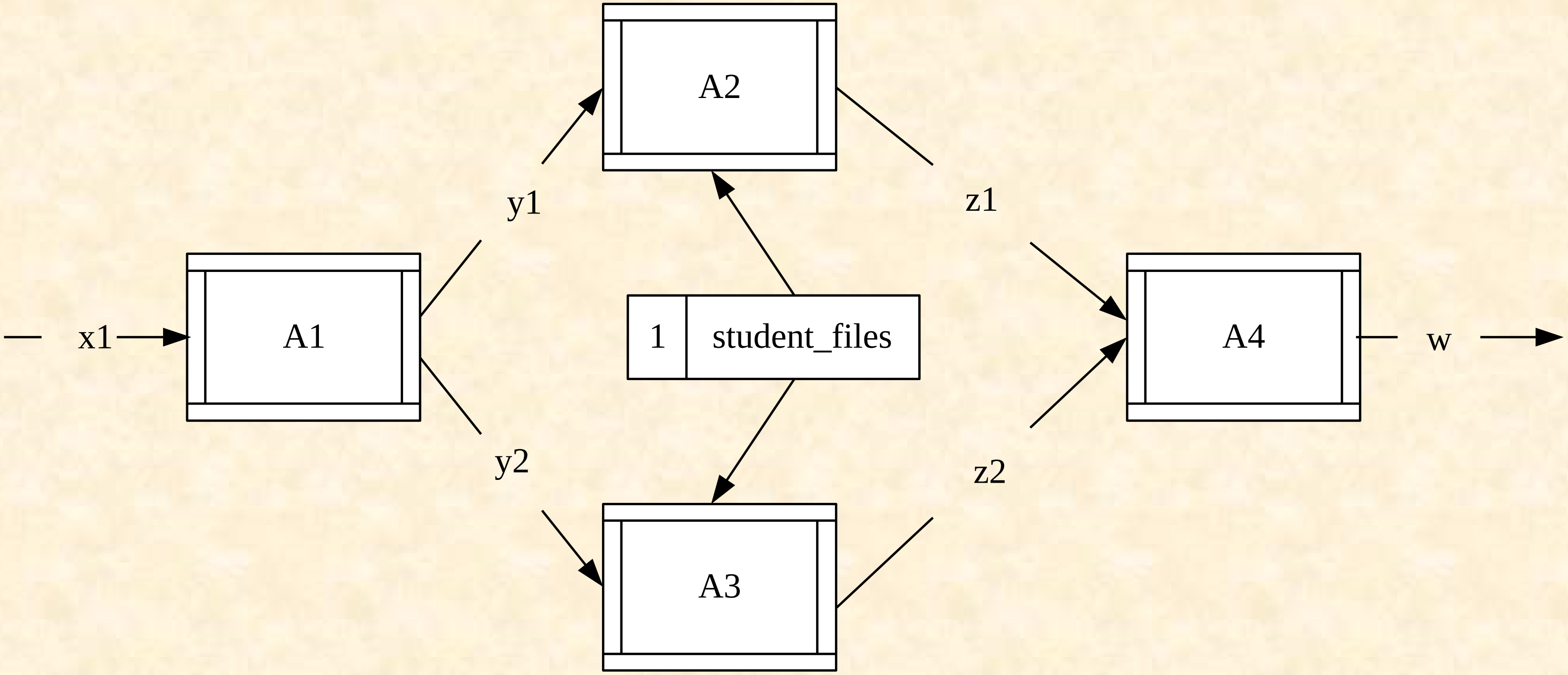
Two parallel processes cannot read from and write to the same data store. Thus, we can avoid possible confusion in operation on the data store.

However, this does not disallow two parallel processes to read from the same data store.

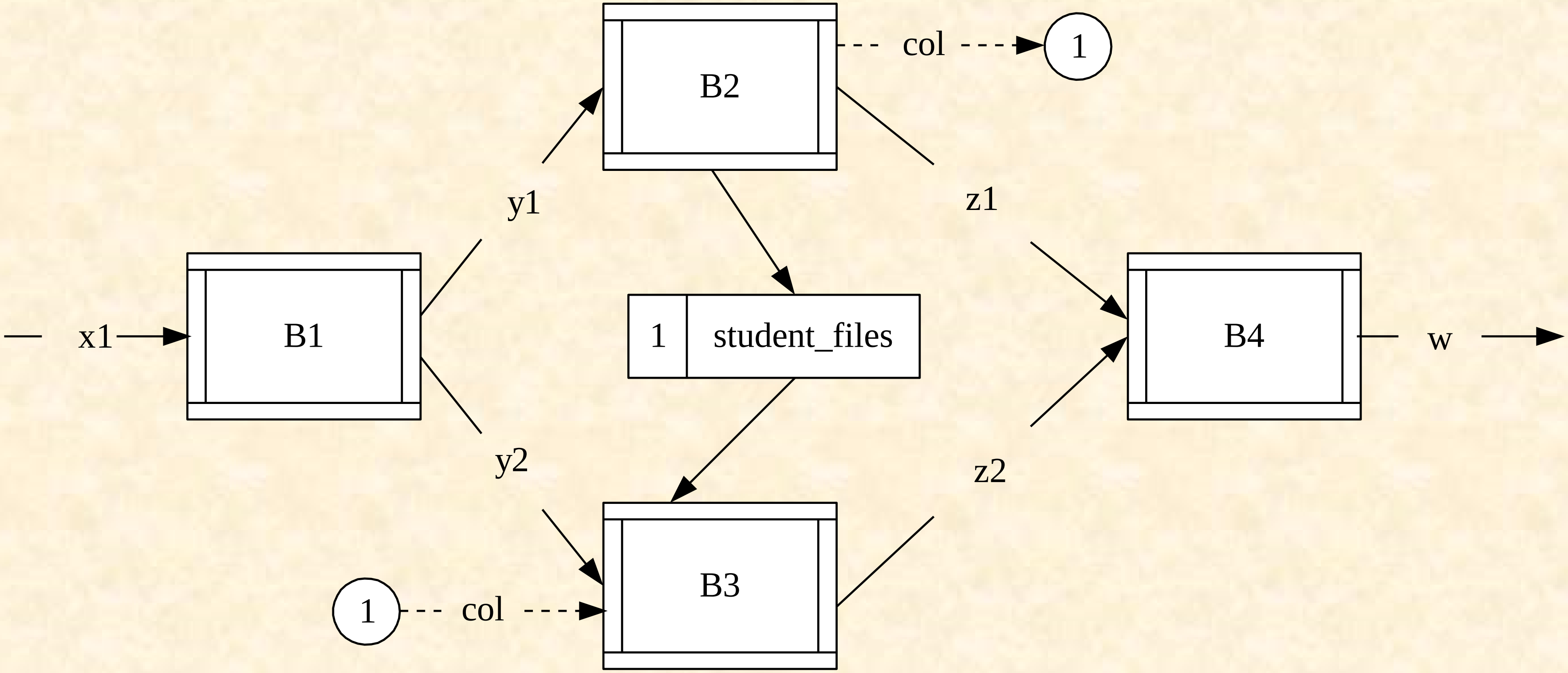
Example: the CDFD below is not allowed.



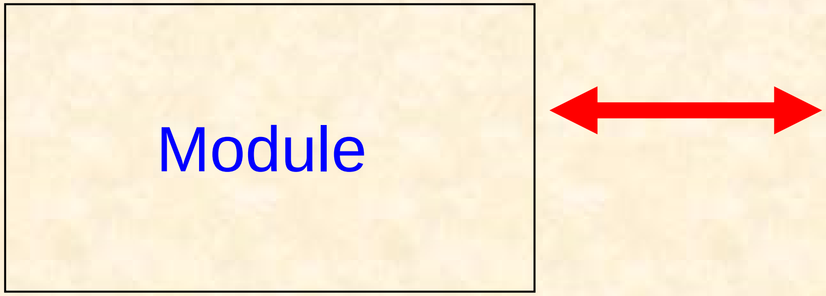
Example: the CDFD below is allowed.

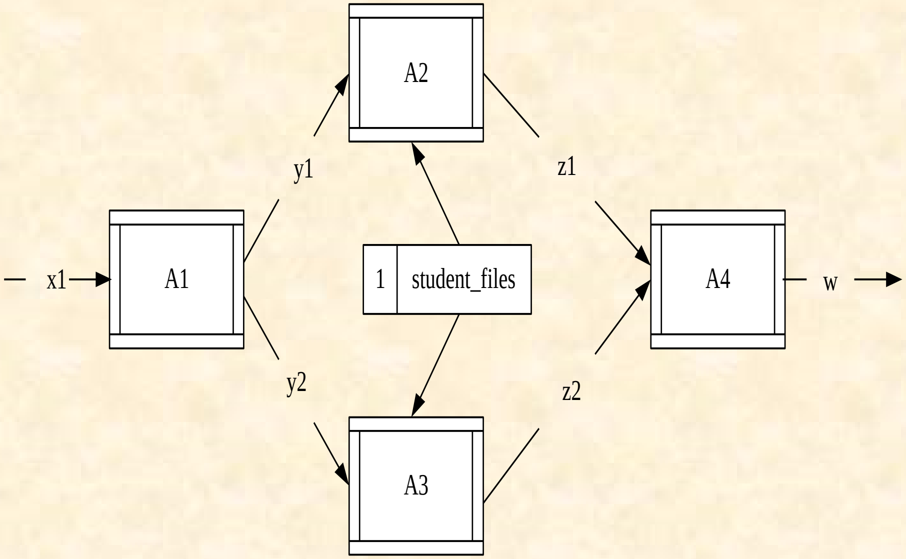


Example: if we really want to describe that process B2 first writes to store student_files and then B3 reads from the same store, we can draw a control data flow from B2 to B3, as shown in the CDFD below.



Associating CDFD with Module





Define all the components of the CDFD, such as processes, data flows, and stores.

```
module ModuleName / ParentModuleName;
 const ConstantDeclaration;
 type TypeDeclaration;
 var VariableDeclaration;
 inv TypeandStateInvariants;
 behav CDFD_no;
 Process Init;    /*for initialization of the local state variables */
 Process_1;
 Process_2;
   ...
 Process_n;
 Function_1;
 Function_2;
  ...
 Function_m
 end_module
```

Constant declaration

A constant with a special meaning may be frequently used in process specifications, but it may subject to change for whatever reason (e.g., to fit requirements changes or module version changes for different systems).

The form of constant declaration:

```
ConstIdentifier_1 = Constant_1;
ConstIdentifier_2 = Constant_2;
     ...
ConstIdentifier_q = Constant_q;
```

Example:

```
const
  age = 20;
```

Type declaration

The form of type declaration:

```
 type
   TypeIdentifier_1 = Type_1;
   TypeIdentifier_2 = Type_2;
             ...
   TypeIdentifier_w = Type_w;
```

Example:

```
 type
    Address = string;
    Employee = given;  /*Employee is treated as a set
                         of values that are not defined
                         precisely, because it is unnecessary at
                         this stage */
```

Variable declaration

All the variables declared in the var section are data store variables occurring in the associated CDFD.

The form of variable declaration:

```
 var
   Variable_1: Type_1;
   Variable_2: Type_2;
           ...
   Variable_u: Type_u;
```

Example:

```
var
   x1, x2, x3: int; /* local stores */
   student_files: set of Account; /*local
                                    store */
   ext x1, x2 : int;  /*external stores passed
                        over from the high
                        level CDFD */
   ext x1, #x2 : int; /*x1 is an external store
                        passed from the high level
                        CDFD, while x2 is an external store
                        exists independently of the system
                        under construction, e.g., file,
                        database. */
```

Type and state invariant

A type invariant is a predicate (usually a quantified predicate expression) that defines a constraint on the type and must be sustained throughout the entire system operation.

A state invariant is also a predicate that defines a constraint on the current state (i.e., on store variables).

The form of invariants:

```
 inv
    Invariant_1;
    Invariant_2;
         ...
    Invariant_v;
```

Example:

```
 inv
   forall[x: Address] | len(x) <= 50;
   card(student_files) <= 1000;
```

Thus, any variable declared with type Address must be constrained by the type invariant. For example,

```
    place: Address;
```

Then, “place” can only hold an address with up to 50 characters.

The behavior of the module

The behavior of a module is defined by the associated CDFD.

The expression that indicates the association between a module and its CDFD is:

```
behav CDFD_10;   /* assuming that the
                    associated CDFD is
                    numbered 10 */
```

Process specification

The general form of a process specification:

```
 process ProcessName(input) output
 ext ExternalVariables
 pre PreCondition
 post PostCondition
 decom LowerLevelModuleName
 explicit ExplicitSpecification
 comment InformalExplanation
 end_process
```

We will focus on decom, explicit, and comment sections.

decom section

decom ProcessName_decom;

ProcessName_decom is the name of a lower level module that is a decomposition of the current process. ProcessName is the name of the current process, while decom is a conventional word, indicating the related module is the decomposition of the process ProcessName.

explicit section

```
explicit
  local variable declaration;
  statement;
```

Example:

```
 explicit
  x: int, y: real;

  if x > 5
  then
     y := (x + 1) / 2;
  else
     y := x / 2;
```

More discussions on explicit specifications will be given later.

How to write comment

There are two kinds of comments. One is used to explain any necessary component in any place of a specification, such as a type, variable, and an invariant. Such a comment is written between a pair of slash-asterisk symbols /* ... */.

Example:

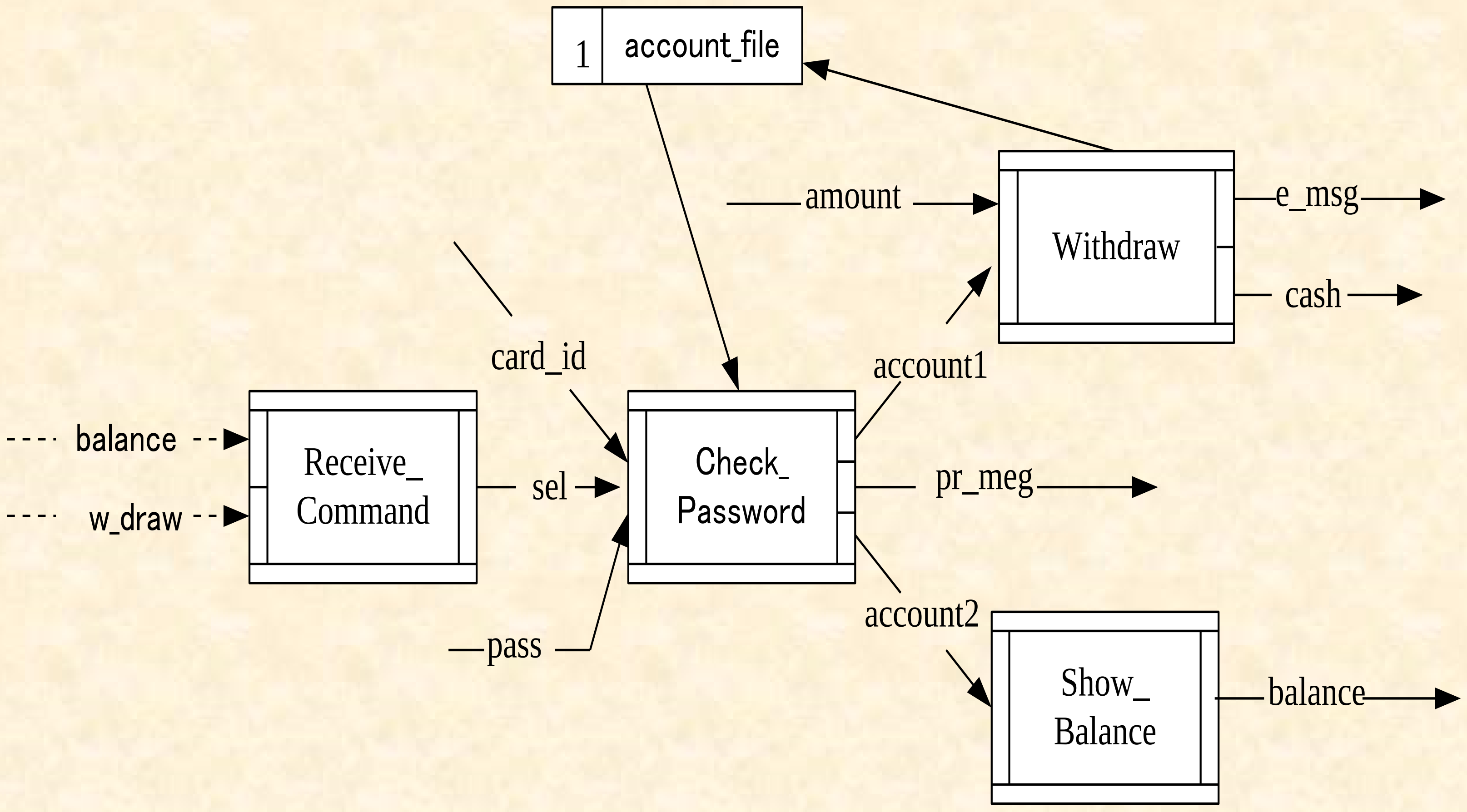
```
var
   student_files: set of Address; /*student_files is
                                    defined as a
                                    collection of
                                    home addresses,
                                    and each address is
                                    represented by a string. */
```

Another kind of comment is written after the keyword comment in a process specification, interpreting the meaning of the formal specification of the process.

Example:

```
process Add(x, y: int) z: int
post z = x + y
comment
 The precondition is true, while the postcondition
   requires that the output z be the sum of the inputs x
   and y.
end_process
```

A module for the ATM



```
module SYSTEM_ATM  /* This module has no parent
  module.*/
 type    Account = composed of
                      account_no: nat
                      password: nat
                      balance: real
                      end
 var   ext #account_file: set of Account; /* the
          account_file is an external store that
          exists independently of the cash
          dispenser. */
 inv
   forall[x: Account] | 1000 <= x.password <= 9999;
      /* The password of every account must be a
          a restricted natural number. */
 behav CDFD_1;   /* Assume the ATM CDFD is numbered 1. */
```

```
process Init()
end_process;  /* The initialization process does
                 nothing because there is no
                 local store in the CDFD to initialize. */
process Receive_Command(balance: sign |
                             w_draw: sign) sel: bool
post bound(balance) and sel = true or bound(w_draw)
     and sel = false
comment
 This process recognizes the input command: show
  balance or withdraw cash. The output data flow sel
  is set to true if the command is showing balance;
  otherwise if the command is withdrawing cash, sel is
  set to false.
end_process;
```

```
process Check_Password(card_id: nat, sel: bool, pass: nat)
                                           account1: Account |
                                            pr_meg: string |
                                            account2: Account
ext rd account_file  /*The type of this variable is omitted because
                                   this external variable has been declared in
                                   the var section. */
post sel = false and
        (exists![x: account_file] | x.account_no = card_id and
                    x.password = pass and account1 = x)  or
         sel = true and
         (exists![x: account_file] | x.account_no = card_id and
                   x.password = pass and account2 = x)    or
         not (exists![x: account_file] | x.account_no = card_id and
                  x.password = pass) and pr_meg = "Reenter your password or insert the
    correct card"
comment
  If sel is false and the input card_id and pass are correct with respect to the exiting
    information in account_file, the account information is passed to the output
    account1. If sel is true and the input card_id and pass are correct, the account
    information is passed to the ouput account2. However, if neither the card_id nor
    pass is correct, a prompt message pr_meg is given.
end_process;
```

```
process Withdraw(amount: real, account1: Account)
                 e_msg: string | cash: real
ext wr account_file
pre account1 inset account_file    /*input account1 must exist in the
  account_file*/
post (exists[x: account_file] | x = account1 and
            x.balance >= amount and
            cash = amount)  and
            account_file = union(diff(~account_file, {account1}),
              {modify(account1, balance -> account1.balance - amount)})
            or
            not exists[x: account_file] | x = account1 and
                                          x.balance >= amount and
                                          e_meg = "The amount is too big")
comment
 The required precondition is that input account1 must belong to the
  account_file. If the request amount to withdraw is smaller than the
  balance of the account, the cash will be withdrawn. On the other hand, if
  the request amount is bigger than the balance of the account, an error
  message "The amount is too big" will be issued.
end_process;
```

```
process Show_Balance(account2: Account)
                     balance: real
post balance = account2.balance;
end_process;
end_module;
```

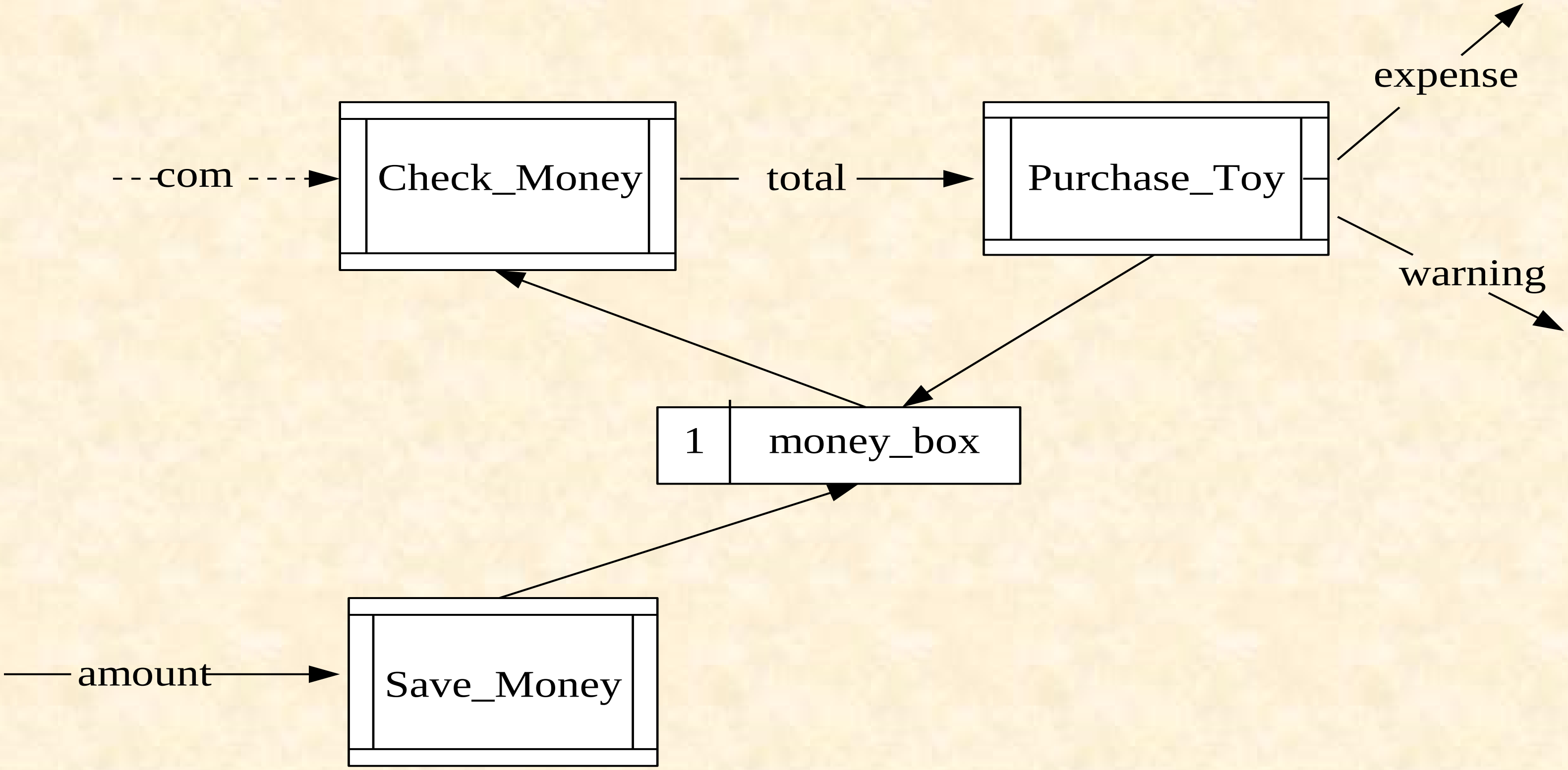
Class exercise 2

1. Define a calculator as a module. Assume that reg denotes the register that should be initialized to 0 and accessed by all the operations defined. The operations include Add, Subtract, Multiply, and Divide. Each operation is modeled by a process.

```
module Calculator;
var
 reg: real;

process Init()
ext ? reg:?
pre ?
post ?
end_process;

process Add(x:?)
ext ? reg: ?
pre ?
post ?
end_process;

process Multiply(x:?)
ext ? reg: ?
pre ?
post ?
end_process;

process Divide(x:?)
ext ? reg: ?
pre ?
post ?
end_process
end_module
```

Homework 1

Write a module to define all the data flows, stores, and processes of the CDFD in Figure 4, assuming all the data flows and stores are integers, and all the processes perform arithmetic operations.



com: command for checking the total amount of the money in the money-box

amount: the amount of money to be saved in the money_box

total: the total amount of the money in the money_box

expense: the sufficient amount for purchasing a toy

warning: a warning message for the shortage of the money in the money_box

Figure 4

```
module Money-Box;
  const
   toy_price = 1000;
  var
   money_box: ?

   process Save_Money(?)
   ext ?
   pre ?
   post ?
   end_process;
   process Check_Money (?) ?
   ext ?
   pre ?
   post ?
   end_process;
   process Purchase_Toy(?)
   ext ?
   pre ?
   post ?
   end_process
 end_module
```

Compound expressions for process specifications

1. The if-then-else expression

The general format is:

if B then E_1 else E_2

Let result denote the conditional expression.

Then it is equivalent to:

B and result = E_1 or not B and result = E_2

Example:

if x > 5 then x + z else z - x

is equivalent to

x > 5 and result = x + z or not x > 5 and result = z - x

2. The let expression

The let expression has the format:

```
    let v_1 = E_1, v_2 = E_2, ..., v_n = E_n
    in P(v_1, v_2, ..., v_n)
```

In this expression each v_i (i =1,...,n) is an identifier that serves as a pattern rather than a variable (whose value may change). This let expression is equivalent to the expression:

```
             P[E_1/v_1, E_2/v_2, ..., E_n/v_n]
```

Example:

```
let x1 = y + z * * 2, x2 = y - z * 5
in
 a* x1 ** 2 + b * x1 + c > a * x2 ** 2 + b * x2 + c
```

This expression is equivalent to:

```
a * (y + z * * 2) ** 2 + b * (y + z * * 2) + c >
a * (y - z * 5) ** 2 + b * (y - z * 5) + c
```

The case expression

A case expression is a multiple conditional expression. Its format is as follows:

```
case x of
 ValueList_1 -> E_1;
 ValueList_2 -> E_2;
            ...
 ValueList_n -> E_n;
 default -> E_n + 1
 end_case
```

Example:

```
case x of
 1, 2, 3 -> y + 1;
 4, 5, 6 -> y + 2;
 7, 8, 9 -> y + 3;
 default -> y + 10
end_case
```

Robust process specification

A process specification is robust if it can deal with any input value in the domain of the process. In other words, it defines a total relation rather than a partial relation.

For example, the process Get is not robust.

```
process Get(z : nat, a : nat) c : nat
ext wr mbox : nat
pre    z >= a
post  c = a and mbox = z – a
end_process
```

The reason why Get is not a robust specification is that Get may not deal with the inputs that do not satisfy the precondition: z >= a.

The robust specification of process Get is:

```
process Get(z : nat, a : nat) c : nat
ext wr mbox : nat
pre    true
post  if z >= a
          then c = a and mbox = z – a
          else c = 0 and mbox = ~mbox
end_process
```

Function definitions

A function provides a mapping from its domain to its range.

A function differs from a process in several ways:

- A function does not allow nondeterministic inputs and outputs whereas a process does.
- A function yields only one group of output whereas a process allows many groups.
- A function does not access to external variables (denoting stores in CDFDs) whereas a process may do so.

Example:

```
process P(x1, x2, x3: int) y1: int | y2: int
 pre is_greater(x1, x2)
 post if is_greater(x1, x3)
         then y1 = x1 * double(x3, x2)
         else y2 = x2 * Increase(x3, x1)
 end_process;
```

Function definitions：

```
function is_greater(a, b: int): bool
 == a > b
 end_function

function double(a, b: int): int
 == 2 * (a + b)
 end_function;

 function Increase(a, b: int): int
 pre true
 post Increase = a + b + a * b
 end_function;
```

There are two kinds of specifications for functions: explicit and implicit specifications.

1. Explicit specification:

```
    function Name(InputDeclaration) : Type
    == E
    end_function
```

Example:

```
  function add(x, y: int) : int
   == x + y
  end_function
```

2. Implicit specification

```
   function Name(InputDeclaration) : Type
   pre Pre
   post Post(Name)
   end_function
```

Example:

```
    function add(x, y: int) : int
    pre true
    post add > x + y
    end_function
```

Undefined function

If function A cannot be defined for some reason, it can be written as:

```
function A(x, y: int) : int
 == undefined
 end_function
```

This means that function A will be defined later in the development process (e.g., implementation).

Recursive functions

A recursive function is a function that applies itself during the computation of its body.

When writing a specification for a recursive function, two points are important:

- the body of the function (for explicit specification) or the postcondition of the function (for implicit specification) must contain an application of the same function.
- an exit is necessary to ensure that any application of the function terminates.

Example: the factorial function is:

$n! = n * (n - 1) * (n - 2) * ... * 3 * 2 * 1$

Let fact denote n!. Then its explicit specification is:

```
function fact(n: nat) : nat
== if n = 1
     then n
     else n * fact(n - 1)
end_function
```

The implicit specification of fact is:

```
function fact(n: nat) : nat
post if n = 1
        then fact = n
        else fact = n * fact(n - 1)
end_function
```

Class exercise 3

Write both the explicit and implicit specifications for the function Fibonacci:

Fibonacci(0) = 0;

Fibonacci(1) = 1;

Fibonacci(n) = Fibonacci(n - 1) + Fibonacci(n - 2)

where n is a natural number of type nat0.

II.3 Data types

A data type (or simply type) consists of a set of values and a set of operators:

Type = a set of values + a set of operators

Basic data types

This part explains all the basic data types available in SOFL.

The outline:

- The numeric types
- The character type
- The enumeration types
- The boolean type
- An example of using basic types

Numeric types

The numeric types include:

nat0 -- {0, 1, 2, 3, …} naturals containing zero.
nat -- {1, 2, 3, …} naturals
int -- {…, -2, -1, 0, 1, 2, …} integers
real -- {…, -2.5, -1.4, 0.0, 1.4, 2.5, …}
real numbers

The operations on the numeric types are given on the next slide.

Operator	Name	Type
- x	Unary minus	real --> real
abs(x)	Absolute value	real --> real
floor(x)	Floor	real --> int
x + y	Addition	real * real --> real
x - y	Subtraction	real * real --> real
x * y	Multiplication	real * real --> real
x / y	Division	real * real --> real
x div y	Integer division	int * int --> int
x rem y	Remainder	int * int --> nat0
x mod y	Modulus	nat0 * nat0 --> nat0
x ** y	Power	real * real --> real

Examples: let x = 9, y = 4.5, z = 3.14, a = - 4, b = 3. Then

- z = - 3.14

abs(a) = 4

floor(y) = 4

x + z = 12.14

x - y = 4.5

a * b = - 12

x / y = 2.0

a div b = -2

a rem b = 2 (quotient = -2)

x mod b = 0

x ** b = 729

The relational operators on numeric types are:

Operator	Name	Type
x < y	Less than	real * real --> bool
x > y	Greater than	real * real --> bool
x <= y	Less or equal	real * real --> bool
x >= y	Greater or equal	real * real --> bool
x < y < z	Less-between	real * real * real -->bool
x <= y <= z	Less-equal-between	real * real * real --> bool
x >= y >= z	greater-equal-between	real * real * real --> bool
x = y	Equal	real * real --> bool
x <> y	Unequal	real * real --> bool

Character type

char

A value of char type: 'x'

Examples:

'a' 'B' '|' ')' ':' '@' '7'

All the characters:

English letters:

a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Other characters:

, . : ; * + - / _ ~| ¥ ( ) [ ] { } @ ^` ' & % $ # " ! < > = ?

Newline

White space

Two characters can only be compared to see if they are the same (=) or different (<>).

Enumeration types

An enumeration type is a finite set of special values, usually with the feature of describing a systematic phenomena.

For example:

Week = {<Monday>, <Tuesday>,
<Wednesday>, <Thursday>,
<Friday>, <Saturday>, <Sunday>}

Except that two values of an enumeration type can be compared to be the same (=) or different (<>), there is no other operator on the enumeration type.

If we declare a variable weekday with the type Week as

```
 weekday: Week;
```

then the variable can take any value of the type, that is, weekday can take <Monday>, <Tuesday>, <Wednesday>, and so on.

```
<Tuesday> = <Tuesday> <=> true
<Tuesday> <> < Wednesday > <=> true
```

Boolean type

bool = {true, false}

Operator	Name
and	and
or	or
not	not
=>	implies
<=>	is equivalent to

These operators also apply to undefined value nil.

An example of using basic types

A simple process telling fares of railway tickets
for different kinds of passengers:

```
process Tell_Fare(passenger: {<STUDENT>,
     <ORDINARY>, <PENSIONER>}) fare: real
ext rd normal_fare: real
post fare = case passenger of
 <STUDENT> —> normal_fare - 0.25 * normal_fare;
 <ORDINARY> —> normal_fare;
<PENSIONER> —> normal_fare - 0.30 * normal_fare
                  end_case
end_process;
```

Class exercise 4

Assume that the courses to teach on weekdays are: "Software Engineering" on Monday, "Program Design" on Tuesday, "Discrete Mathematics" on Wednesday, "Programming Language" on Thursday, and "Formal Engineering Methods" on Friday. Write a formal specification for the process that gives the corresponding course title for an input weekday.

(Hint: define a type Course as an enumeration type)

Set types

The set types are one of the compound types available in SOFL, and usually used for the abstraction of data items that have a collection of elements.

The outline of this part:

- What is a set?
- Set type constructors
- Constructors and operators on sets
- Specification with set types

What is a set?

A set is an unordered collection of distinct objects where each object is known as an element of the set.

For example:

(1) A class is a set of students.
(2) A car park is a set of cars.

A set of values is enclosed with braces. For example,

(1) {5, 9, 10}
(2) {"John", "Chris", "David", "Jeff"}
(3) {"Java", "Pascal", "C", "C++", "Fortran"}

Notice: {a, a, b} is not a legal set.

Set type declaration

Let T be an arbitrary type, A be a set type to be defined. Then, the declaration of A has the form:

A = set of T

where T is called “element type”.

Formally, A is the power set of T:

$$A = \{x \mid \text{subset}(x, T)\}$$

where subset(x, T) means that x is a subset of T.

For example: let A be defined as follows:

```
type
A = set of {<DOG>, <CAT>, <COW>}
```

This means:

```
A = {{ }, {<DOG>}, {<CAT>}, {<COW>},
        {<DOG>, <CAT>}, {<DOG>, <COW>},
        {<CAT>, <COW>}, {<DOG>, <CAT>, <COW>}}
```

Set variable declaration:

Let s be a variable of type A, which is declared as:

s: A;

then, s can take any value of A:

s = { } (empty set) or
s = {<DOG>} or
s = {<CAT>} or
s = {<COW>} or
s = {<DOG>, <CAT>} or
s = {<DOG>, <COW>} or
s = {<CAT>, <COW>} or
s = {<DOG>, <CAT>, <COW>}

Constructors and operators on sets

1. Constructors

A constructor of set types is a special operator that constitutes a set value from the elements of an element type.

There are two set constructors:

set enumeration and set comprehension.

A set enumeration has the format:

{e_1, e_2, ..., e_n}

where e_i (i=1..n) are the elements of the set {e_1, e_2, ..., e_n}.

Examples:

{5, 9, 10, 50}

{‘a’, ‘t’, ‘l’}

A set comprehension has the form:

{e(x_1, x_2, ..., x_n) | x_1: T_1, x_2: T_2, ..., x_n: T_n & P(x_1, x_2, ..., x_n)}

or

{e(x_1, x_2, ..., x_n) | P(x_1, x_2, ..., x_n)}

where n >= 1.

The set comprehension defines a collection of values resulting from evaluating the expression e(x_1, x_2, ..., x_n) (n>=1) under the condition that the involved variables x_1, x_2, ..., x_n take values from sets (or types) T_1, T_2, ..., T_n, respectively, and satisfies property P(x_1, x_2, ..., x_n).

Examples:

{x | x: nat & 1 < x < 5} = {2, 3, 4}

{y | y: nat0 & y <= 5} = {0, 1, 2, 3, 4, 5}

{x + y | x: nat0, y: nat0 & 1 < x + y < 8} =
{2, 3, 4, 5, 6, 7}

{i | i inset {1, 3, 5, 7}} = {1, 3, 5, 7}

We can also use the following special notation to represent a set containing an interval of integers:

{i, ..., k} = {j | j: int & i <= j <= k}

Thus:

{1, ..., 5} = {1, 2, 3, 4, 5}

{-2, ..., 2} = {-2, -1, 0, 1, 2}

2. Operators

2.1 Membership (inset)

```
inset: T * set of T --> bool
```

Examples:

```
7 inset {4, 5, 7, 9} <=> true
3 inset {4, 5, 7, 9} <=> false
```

2.2 Non-membership (notin)

notin: T * set of T --> bool

Examples:

7 notin {4, 5, 7, 9} <=> false

3 notin {4, 5, 7, 9} <=> true

2.3 Cardinality (card)

card: set of T --> nat0

Examples:

card({5, 7, 9}) = 3

card({'h', 'o', 's', 'e', 'i'}) = 5

2.4 Equality and inequality (=, <>)

=: set of T * set of T --> bool
s1 = s2
== forall[x: s1] | x inset s2 and card(s1) = card(s2)

“==“ means “defined as”.

<>: set of T * set of T --> bool
s1 <> s2
== (exists[x: s1] | x notin s2) or (exists[x: s2] | x notin s1)

Examples:

{5, 15, 25} = { 25, 15, 5} <=> true
{5, 15, 25} <> {5, 20, 30} <=> true

2.5 Subset (subset)

subset: set of T * set of T --> bool

subset(s1, s2) == forall[x: s1] | x inset s2

Examples:

Let s1 = {5, 15, 25}, s2 = {5, 10, 15, 20, 25, 30}.
Then:

subset(s1, s2) <=> true   subset(s2, s1) <=> false
subset({ }, s1) <=> true   subset(s1, s1) <=> true

2.6 Proper subset (psubset)

psubset: set of T * set of T --> bool

psubset(s1, s2) == subset(s1, s2) and s1 <> s2

Examples:

let s1 = {5, 15, 25} and s2 = {5, 10, 15, 25, 30}.

Then:

psubset(s1, s2) <=> true

psubset(s1, s1) <=> false

psubset(s2, s1) <=> false

psubset({ }, s1) <=> true

2.7 Member access (get)

get: set of T --> T

get(s) == if s <> { } then x else nil

where x inset s.

Examples: assume s = {5, 15, 25}, then

get(s) = 5 or

get(s) = 15 or

get(s) = 25

And s still remains the same as before:

s = {5, 15, 25}.

2.8 Union (union)

```
union: set of T * set of T --> set of T
union(s1, s2) == {x | x inset s1 or x inset s2}
```

Examples:

```
union({5, 15, 25}, {15, 20, 25, 30}) =
                          {5, 15, 25, 20, 30}
union({15, 20, 25, 30}, {5, 15, 25}) =
                          {5, 15, 25, 20, 30}
```

The union operator is commutative. Thus,

union(s1, s2) = union(s2, s1).

It is also associative, that is,

union(s1, union(s2, s3)) =
    union(union(s1, s2), s3).

Due to these properties, the operator union can be extended to deal with more than two sets:

union: set of T * set of T * ... * set of T --> set of T

union(s1, s2, ..., sn) == {x | x inset s1 or x inset s2 or ... or x inset sn}

2.9 Intersection (inter)

```
inter: set of T * set of T --> set of T
inter(s1, s2) == {x | x inset s1 and x inset s2}
```

For example, let s1 = {5, 7, 9},
s2 = {7, 10, 9, 15},
s3 = {8, 5, 20}.

Then

```
inter(s1, s2) = {7, 9}
inter(s1, s3) = {5}
inter(s2, s3) = { }
```

The inter operator is commutative and associative. That is,

```
inter(s1, s2) = inter(s2, s1),
inter(s1, inter(s2, s3)) = inter(inter(s1, s2), s3).
```

We can also extend the inter operator to deal with more than two operands:

```
inter: set of T * set of T * ... * set of T --> set of T
inter(s1, s2, ..., sn)
== {x | x inset s1 and x inset s2 and ... and x inset sn}
```

2.10 Difference (diff)

diff: set of T * set of T --> set of T
diff(s1, s2) == {x | x inset s1 and x notin s2}

For example, let s1 = {5, 7, 9}
s2 = {7, 10, 9, 15}
s3 = {8, 12}.

Then

diff(s1, s2) = {5}
diff(s1, s3) = {5, 7, 9}
diff(s2, s1) = {10, 15}
diff(s1, { }) = s1

2.11 Distributed union (dunion)

A set can be a set of sets, and the distributed union of such a set is an operation that obtains the union of all the member sets of the set.

```
 dunion: set of set of T --> set of T
 dunion(s) == union(s1, s2, ..., sn)
```

where s = {s1, s2, ..., sn}.

Example:

Let s1 = {{5, 10, 15}, {5, 10, 15, 25}, {10, 25, 35}}
Then
   dunion(s1) = union({5, 10, 15}, {5, 10, 15, 25}, {10, 25, 35}}
   = {5, 10, 15, 25, 35}

2.12 Distributed intersection (dinter)

dinter: set of set of T --> set of T

dinter(s) == inter(s1, s2, ..., sn)

where s = {s1, s2, ..., sn}.

For example, let

s = {{5, 10, 15}, {5, 10, 15, 25}, {10, 25, 35}}.

Then

dinter(s) = inter({5, 10, 15}, {5, 10, 15, 25},
{10, 25, 35}} = {10}

2.13 Power set (power)

Given a set, we can apply the operator power to yield its power set that contains all the subsets of the set, including the empty set.

```
power: set of T --> set of set of T
power(s) == { s1 | subset(s1, s)}
```

Example: let s = {5, 15, 25}. Then

```
  power(s) = {{ }, {5}, {15}, {25}, {5, 15},
                     {15, 25}, {5, 25}, {5, 15, 25}}
```

Specification with set types

An Email_Address_Book

```
module Email_Address_Book;
 type
  EmailAddress = given;
 var
  email_book: set of EmailAddress;
 behav: CDFD_1;
```

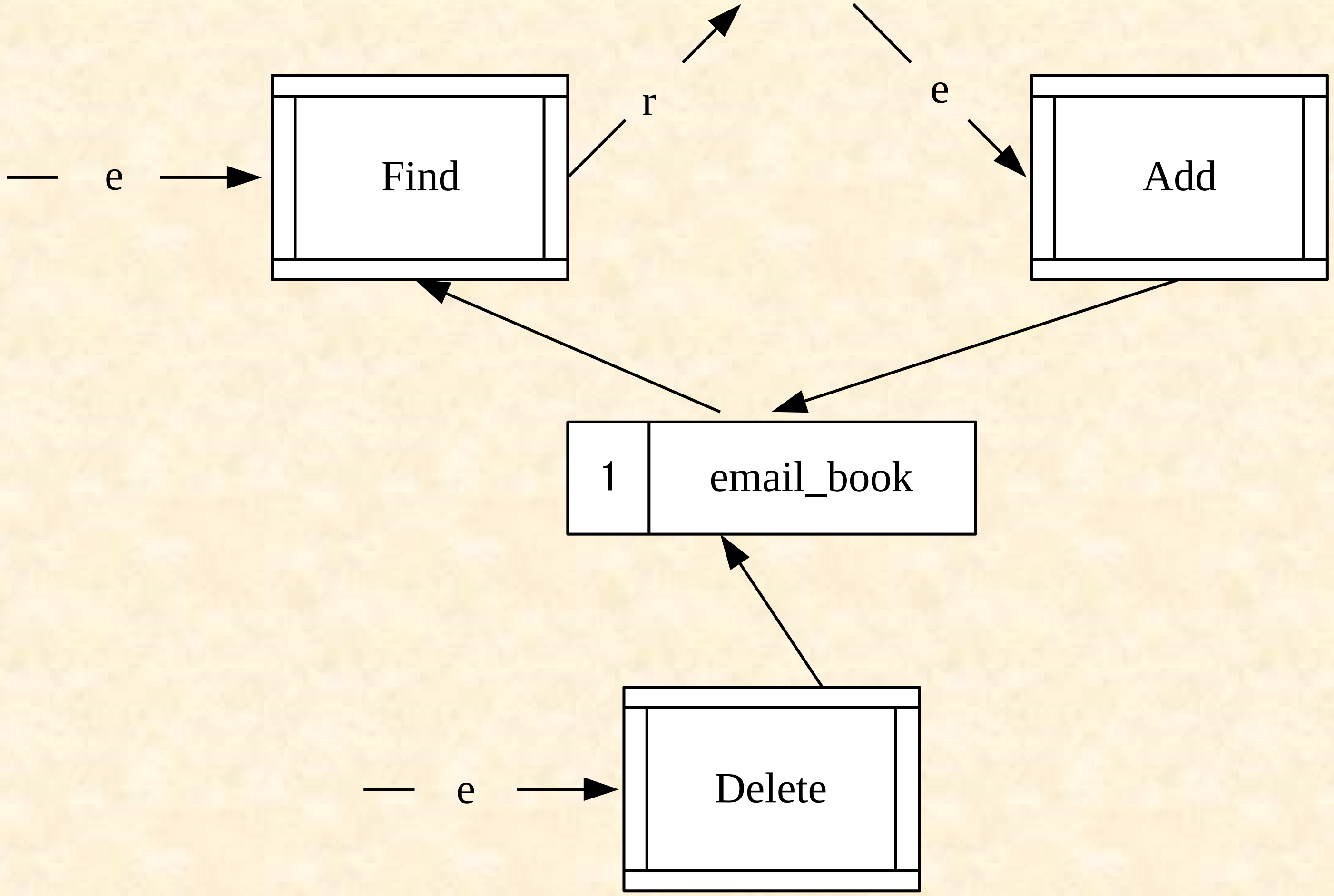


Figure 1

```
process Find(e: EmailAddress) r: bool
ext rd email_book
post r = (e inset email_book)
end_process;

process Add(e: EmailAddress)
ext wr email_book
pre e notin email_book
post email_book = union(~email_book, {e})
end_process;

process Delete(e: EmailAddress)
ext wr email_book
post email_book = diff(~email_book, {e})
end_process;
end_module;
```

Class exercise 5

1. Let s1 = {5, 15, 25}, s2 = {15, 30, 50}, s3 = {30, 2, 8}, and s = {s1, s2, s3}.

Evaluate the expressions:

a. card(s1)

b. card(s)

c. union(s1, s2)

d. diff(s2, s3)

e. inter(union(s2, s3), s1)

f. dunion(s)

g. dinter(s)

h. inter(union(s1, s3), diff(s2, union(s1, s3))

2. Construct a module to model a telephone book containing a set of telephone numbers. The necessary processes are Add, Find, Delete, and Update. The process Add adds a new telephone number to the book; Find tells whether a given telephone number is available or not in the book; Delete eliminates a given telephone number from the book; and Update replaces an old telephone number with a new number in the book.

3. Write a specification for a process Merge. The process takes two groups of students and merge them into one group. Since the merged group will be taught by a different professor, the students from both groups may drop from the merged group (but exactly which students will drop is unknown).

Sequence and string types

Contents:

- What is a sequence?
- Sequence type declarations
- Constructors and operators on sequences
- Specifications using sequences

What is a sequence?

A sequence is an ordered collection of objects that allows multiple occurrences of the same object. As with sets, the objects are known as elements of the sequence.

Examples:

(1) [5, 15, 15, 5, 35]

(2) ['u', 'n', 'i', 'v', 'e', 'r', 's', 'i', 't', 'y']

(3) [20.5, 40.5, 85.5]

The “string” type

A string is a special sequence in the sense that its all elements are only characters. In other words, a string is a sequence of characters.

We use the keyword string to denote the type that contains all the possible strings.

Examples of strings:

"university"
"sofl@yahoo.ac.jp"
"Formal Engineering Methods"

Sequence type declarations

A sequence type A is declared based on an element type T in the following format:

A = **seq of** T

Example:

Ages = seq of nat

Then, we can declare a variable of type Ages:

student_ages: Ages

Constructors and operators on sequences

A sequence can be created using either sequence constructors or operators.

1. Constructors

There are two constructors: sequence enumeration and sequence comprehension.

(2.1) Sequence enumeration

A sequence enumeration has the format:

[a_1, a_2, ..., a_n]

where a_i (i=1..n) are the elements of the sequence.

Example:

[5, 9, 8, 9, 5]

The order and the occurrences of the elements are significant. Thus:

[5, 9] <> [9, 5] and

[5, 9, 5] <> [5, 9]

(2.2) Sequence comprehension

A sequence comprehension takes the format:

[e(x_1, x_2, ..., x_n) | x_1: T_1, x_2: T_2, ..., x_n: T_n
& P(x_1, x_2, ..., x_n)]

The sequence comprehension defines a sequence whose elements are derived from the evaluation of expression e(x_1, x_2, ..., x_n) under the condition that x_1 takes values from type T_1, x_2 from T_2, ..., x_n from T_n, and all of these values satisfy property P(x_1, x_2, ..., x_n).

Note that all the types T_i (i =1,...,n) are countable numeric types and the elements of the sequence must occur in the ascending order. For example,

[i * j | i: nat, j: nat &1 <= i + j <= 3] = [1, 2, 2]

As with the set notation, we also use the following special notation to represent a sequence of integer interval from i to j:

[i, ..., j] = [x | x: int & i <= x <= j]

Thus:

[3, ..., 6] = [3, 4, 5, 6]
[-2, ..., 2] = [-2, -1, 0, 1, 2]
[0, ..., 4] = [0, 1, 2, 3, 4]

However, if index j is smaller than i, [i, ..., j] will represents the empty sequence [ ]. For example,

[9, ..., 2] = [ ]

2. Operators

All the operators are applicable to variables of string type as well.

2.1 Length (len)

The length of a sequence means the number of its elements.

```
len: seq of T --> nat0
len(s) == the number of its elements.
```

Examples: let s1 = [4, 9, 10], s2 = [{3, 9}, {6}],
s3 = [10, 9, 4, 25], and s4 = "university".

Then:

```
len(s1) = 3
len(s2) = 2
len(s3) = 4
len(s4) = 10
```

2.2 Sequence application

A sequence can apply to an index, a natural number, to yield the element occurring at the position indicated by the index.

Let s be a sequence of type seq of T. Then, s can be regarded as a function from nat to T:

```
    seq of T * nat --> T
    s(i) == the ith element of sequence s
```

The precondition for applying s to an index i (i.e., s(i)) is that index i is within the range of 1 to len(s). Otherwise, if i is beyond this range, the sequence application s(i) is undefined.

Examples: let s1 = [4, 9, 10], s2 = [{3, 9}, {6}],

s3 = [10, 9, 4, 25], and s4 = "university".

Then:

s1(1) = 4

s1(2) = 9

s2(1) = {3, 9}

s3(4) = 25

s4(5) = 'e'

2.3 Subsequence

A subsequence of a sequence is part of the sequence.

Let s be a sequence of type seq of T, and i and j are two indexes. Then the subsequence of s that keeps the elements in the same order as they are in s is denoted as:

```
   seq of T * nat * nat --> seq of T
   s(i, j) == [s(i), s(i + 1), ..., s(j - 1), s(j)]
```

Examples:

```
   s1(2, 3) = [9, 10]     s1(1, 3) = s1
   s3(2, 4) = [9, 4, 25]     s4(2, 8) = "niversi"
```

2.4 Head (hd)

The head of a non-empty sequence is its first element.

```
    hd(s: seq of T) he: T
    pre   s <> []
    post  he = s(1)
```

If s is the empty sequence, hd(s) is undefined.

For example, let s1 = [4, 9, 10], s2 = [{3, 9}, {6}],
s3 = [10, 9, 4, 25], and s4 = "university".

hd(s1) = 4,   hd(s2) = {3, 9},   hd(s3) = 10,
hd(s4) = 'u'

2.5 Tail (tl)

The tail of a non-empty sequence is its subsequence resulting from eliminating its head.

```
tl(s: seq of T) ts: seq of T
pre s <> []
post ts = s(2, len(s))
```

The application of the operator tl to the empty sequence is undefined, that is, tl([]) = nil.

For example: let s1 = [4, 9, 10], s2 = [{3, 9}, {6}],
s3 = [10, 9, 4, 25], and s4 = "university".

tl(s1) = [9, 10], tl(s2) = [{6}],
tl(s3) = [9, 4, 25], tl(s4) = "niversity"

2.6 Elements (elems)

The operator for obtaining the set of all the elements of a sequence is elems that is defined as:

```
 elems: seq of T --> set of T
 elems(s) == {x | x: T &
                    (exists[i: {1, ..., len(s)}] | x = s(i)})
```

Since the result of elems(s) is a set, not a sequence,
no duplication of elements of s is allowed.

For example, let s1 = [4, 9, 10], s2 = [{3, 9}, {6}],
s3 = [10, 9, 4, 25], and s4 = "university".

Then,

elems(s1) = {4, 9, 10}   elems(s2) = {{3, 9}, {6}}
elems(s3) = {10, 9, 4, 25}
elems([5, 10, 5, 10, 15]) = {5, 10, 15}
elems(s4) = {'u', 'n', 'i', 'v', 'e', 'r', 's', 't', 'y'}
elems([ ]) = { }.

2.7 Indexes (inds)

A sequence corresponds to a set of natural numbers that indicates the positions of the elements of the sequence. Such a set is known as index set.

```
 inds: seq of T --> set of nat
 inds(s) == {i | i: nat & exists[ x: elems(s)] | s(i) = x}
```

It is obvious that the index set of the empty sequence is the empty set.

Furthermore, the cardinality of inds(s) is equal to the length of sequence s, but may be greater than the number of the elements of set elems(s) due to the possibility of having duplicated elements in s.

For example, let s1 = [4, 9, 10],
s2 = [{3, 9}, {6}],
s3 = [10, 9, 4, 25],
s4 = "university".

Then:

inds(s1) = {1, 2, 3}
inds(s2) = {1, 2}
inds(s3) = {1, 2, 3, 4}
inds(s4) = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

The index set is often used when describing a property of a sequence. Consider the example:

exists[i: inds(s)] | s(i) > 5

This quantified expression describes a property of sequence s, requiring that s has at least one element greater than 5.

2.8 Concatenation (conc)

Sequences can be concatenated to form another sequence.

```
conc(s_1: seq of T, s_2: seq of T) cs: seq of T
post (forall[i: inds(s_1)] | cs(i) = s_1(i)) and
        (forall[j: inds(s_2)] | cs(j + len(s_1)) = s_2(j)) and
        len(cs) = len(s_1) + len(s_2)
```

The concatenation of sequences s_1 and s_2 is formed by appending s_2 to the end of s_1.

Examples:

conc(s1, s3) = [4, 9, 10, 10, 9, 4, 25]

conc(s4, s4) = "universityuniversity"

The concatenation of sequences is not commutative. Thus:

conc(s1, s3) <> conc(s3, s1)

The concatenation operator conc can be extended to deal with more than two sequences. Thus:

conc(s_1, s_2, ..., s_n) = conc(s_1, conc(s_2, conc(s_3, ...)))

For example, let s1 = [5, 15, 25],
s2 = [10, 20, 30, 40],
s3 = [2, 4, 6, 8, 10].

Then:

conc(s1, s2, s3) = [5, 15, 25,
10, 20, 30, 40,
2, 4, 6, 8, 10]

2.9 Distributed concatenation (dconc)

Let S be a sequence of sequences:

S = [s_1, s_2, ..., s_n]

where each s_i (i =1,…,n) is a sequence.

dconc: seq of seq of T --> seq of T
dconc(S) == conc(s_1, s_2, ..., s_n)

Example: let S1 = [[5, 15, 25], [10, 20, 30, 40],
[2, 4, 6, 8, 10]]

Then

dconc(S1) = [5, 15, 25, 10, 20, 30, 40,
2, 4, 6, 8, 10]

2.10 Equality and inequality (= and <>)

Sequences can be compared to determine whether they are identical or not.

s_1 = s_2 <=> (len(s_1) = len(s_2) and
forall[i: inds(s_1)] | s_1(i) = s_2(i))

s_1 <> s_2 <=> not s_1 = s_2

Examples: let s1 = [5, 15, 25], s2 = [10, 20, 30, 40],
s3 = [2, 4, 6, 8, 10].

Then:

s1 = s1 <=> true   s1 <> s2 <=> true
s2 = s3 <=> false

Specifications using sequences

1. Sorting of an integer sequence

```
Example: ~list = [3, 5, 8, 5, 9, 3, 10, 5]
                   list = [3, 3, 5, 5, 5, 8, 9,10]
   module SortingOfIntegerSequence;
    var
      list: seq of int;
    process Sort()
    ext wr list: seq of int
    pre   true
    post  Is_Permutation(~list, list) and Is_Ordered(list)
    comment
    After the sorting, the final list must maintain the same
    elements including their all occurrences and keep their
    elements in the ascending order.
    end_process;
```

```
function Is_Permutation(l1, l2: seq of int): bool

== forall[e: union(elems(l1), elems(l2))] |

          card({i | i: inds(l1) & l1(i) = e}) =

          card({i | i: inds(l2) & l2(i) = e})

end_function;
```

```
function Is_Ordered(l: seq of int): bool
== forall[i,j: inds(l)] | i < j => l(i) <= l(j)
end_function;
```

The characteristic of this specification is that it says nothing about how to sort the integer sequence list, but focuses on the relation between the initial list (i.e., ~list) and the final list.

2. Membership Management System

```
module MembershipManagementSystem;
  type
    Member = string;  /* A member is denoted by its name
                                    which is a string of characters */
  var
    all_members: seq of Member;

  process Register(m: Member)
  ext wr all_members
  post  all_members = conc(~all_members, [m])
  comment
    The function of recording the member m in the member list
  all_members is specified by defining the all_members after
  the process as the concatenation of the all_members before
  the process and the sequence composed of member m.
  end_process;
```

```
 process Search(m: Member) pos: set of nat
 ext rd all_members
 post  pos = {i | i: nat & all_members(i) = m}
 comment
  Finding all the positions of member m in the
  member list all_members is modeled by a set
  comprehension.
end_process;
```

```
process Exchange(pos1, pos2: nat)
ext wr all_members
pre pos1 inset inds(all_members) and pos2
       inset inds(all_members)
post  all_members(pos1) = ~all_members(pos2) and
         all_members(pos2) = ~all_members(pos1) and
         forall[i | i: inds(all_members)] &
                      i <> pos1 and i <> pos2 =>
                       all_members(i) = ~all_members(i)
comment    This process only exchanges the
members at position pos1 and pos2, and keeps the
rest of the members unchanged in the list.
end_process;
end_module.
```

Class exercise 6

1. Given a set T = {1, 2, 5}, declare a sequence type based on T, and list up to 5 possible sequence values in the type.

2. Evaluate the sequence comprehensions:
   a. [x | x: nat & 3 < x < 8]
   b. [y | y: nat0 & y <= 3]
   c. [x * x | x: nat, y: nat & 1 <= x <= 3]

3. Let s1 = [5, 15, 25], s2 = [15, 30, 50], s3 = [30, 2, 8], and s = [s1, s2, s3]. Evaluate the expressions:
   a. hd(s1)
   b. hd(s)
   c. len(tl(s1)) + len(tl(s2)) + len(tl(s3))
   d. len(s1) + len(s2) - len(s3)
   e. union(elems(s1), elems(s2))
   f. inter(union({hd(s2)}, elems(s3)), elems(s1))
   g. union(inds(s1), inds(s2), inds(s3))
   h. elems(conc(s1, s2, s3))
   i. dconc(s)

4. Construct a module to model a queue of integers with the processes: Append, Eliminate, Read, and Count. The process Append adds a new element to the queue; Eliminate deletes the top element of the queue; Read tells what is the top element; and Count yields the number of the elements in the queue.

Composite and product types

Contents:

- Composite types
  - Constructing a composite type
    - Constructor
  - Operators
  - Comparison
- Product types
- Examples of specification

Composite types

A composite object usually contains several fields, each describing the different aspect of the object. A composite object is like a record in Pascal and structure in C. A composite type provides a set of composite objects.

A composite type is constructed using the type constructor: composed of ... end.

The general form of a composite type declaration:

```
 A = composed of
        f_1: T_1
        f_2: T_2
          ...
        f_n: T_n
      end
```

where f_i (i =1,...,n) are variables called fields and T_i are their types. Each field represents an attribute of the composite object of the type. A value of a composite type is called composite object (or composite value).

A variable co of composite type A can be declared in one of the following forms:

```
(1)  co: A

(2)  co: composed of
            f_1: T_1
            f_2: T_2
              ...
            f_n: T_n
            end
```

Example: type declaration:

```
Account = composed of
                account_no: nat
                 password: nat
                balance: real
               end
```

Variable declaration:

```
account: Account
```

(1) Constructor

Only one constructor known as make-function is available for composite types. The format:

mk_A(v_1, v_2, ..., v_n)

The make-function yields a composite value of composite type A whose field values are v_i (i =1,…,n) that corresponds to fields f_1, f_2, ..., f_n, respectively. For example,

account = mk_Account(1073548, 1234, 5000)

(2) Operators

(2.1) Field select

Let co be a variable of composite type A. Then, we use

co.f_i

to represent the field f_i (i =1,...,n) of the composite object co.

Examples:

account.password
account.balance

(2.2) Field modification (modify)

Given a composite value, say co, of type A, we can apply the field modification operator modify to create another composite value of the same type.

```
modify(co, f_1 -> v_1, f_2 -> v_2, ..., f_n -> v_n)
```

Example:
let account = mk_Account(1073548, 1234, 5000)
Then, we can have the expression:

```
   account1 = modify(account, password -> 4321)
```

(3) Comparison

Two composite values can be compared to determine whether they are identical or not.

Examples:

```
mk_Account(1073548, 1234, 5000) =
   mk_Account(1073548, 1234, 5000)
```

```
mk_Account(1073548, 1234, 5000) <>
   mk_Account(1073548, 4321, 5000)
```

Product types

A product type defines a set of tuples with a fixed length.

Let T_1, T_2, ..., T_n be n types.
Then, a product type T is declared as follows:

```
T = T_1 * T_2 * ... * T_n
```

A value of T is created using the make-function:

```
mk_T(v_1, v_2, ..., v_n)
```

Example:

Suppose type Date is declared as:

Date = nat0 * nat0 * nat0

Then

mk_Date(1999, 7, 25)
mk_Date(2000, 8, 30)
mk_Date(2001, 7, 10)

are the values of type Date.

Examples: the use of tuples:

```
d: Date;

d = mk_Date(1999, 7, 25)

d = mk_Date(2000, 8, 30)

d = mk_Date(2001, 7, 10)
```

There are two operators on tuples: tuple application and tuple modification.

(1) A tuple application yields an element of the given position in the tuple, whose general format is:

a(i): T * nat --> T_i

where a is a variable of product type T; i is a natural number indicating the position of the element referred to in tuple a; and T_i denotes the ith type in the declaration of T.

For example, let

date1 = mk_Date(1999, 7, 25)

date2 = mk_Date(2000, 8, 30)

Then, the following results can be derived:

date1(1) = 1999

date1(2) = 7

date1(3) = 25

date2(1) = 2000

date2(2) = 8

date2(3) = 30

A tuple can also be directly used in applications.

Examples:

```
mk_Date(2000, 8, 30)(2) = 8
mk_Date(2000, 8, 30)(3) = 30
```

(2) A tuple modification is similar to a composite value modification. The same operator modify is also used for tuple modification, but with slightly different syntax:

```
modify: T * T_1 * T_2 * ... * T_n -> T
modify(tv, 1 -> v_1, 2 -> v_2, ..., n -> v_n)
```

where T is a product type, T_i (i =1,…,n, n >= 1) are the element types. This operation yields a tuple of the same type based on the given tuple tv, with the first element being v_1, the second element being v_2, and so on.

Examples:

```
modify(mk_Date(2000, 8, 30), 1 --> 2001, 3 --> 20) =
    mk_Date(2001, 8, 20)

modify(mk_Date(2001, 8, 20), 2 --> 15) =
    mk_Date(2001, 15, 20)
```

An example of specification

Suppose we want to build a table to record students' credits resulting from two courses:

personal data	course1	course2	total
Helen,0001,A3	2	2	4
John, 0002,A2	0	2	2
...	...	...	...

We aim to build several processes on this kind of table.

This table can be perceived as a sequence of student data. That is,

T = [OneStudent1, OneStudent2, …,
OneStudentn]

```
module Students_Record;
 type
  CourseCredit = nat0;
  TotalCredit = nat0;
  PersonalData = composed of
                   name: string
                  id: nat0
                  class: string
                 end;
  OneStudent = PersonalData * CourseCredit * CourseCredit *
               TotalCredit;
  StudentsTable = seq of OneStudent;

 var
  students_table: StudentsTable;

 inv
  forall[i, j: inds(students_table)] | i <> j => students_table(i)(1).id <>
                               students_table(j)(1).id);
```

```
process Search(search_id: nat0)
                    info: OneStudent
ext rd students_table
pre exists[i: inds(students_table)] |
  students_table(i)(1).id = search_id
post exists[i: inds(students_table)] |
  students_table(i)(1).id = search_id and
  info = students_table(i)]
end_process;
```

```
process Update(one_student: OneStudent, credit1,
                  credit2: CourseCredit)
ext wr students_table
pre  exists[i: inds(students_table)] |
                      students_table(i) = one_student
post len(students_table) = len(~students_table) and
     forall[i: inds(~students_table)] |
       (~students_table(i) = one_student =>
        students_table(i) =
          modify(~students_table(i), 2 -> credit1,
                          3 -> credit2,
                          4 -> credit1 + credit2)) and
   (~students_table(i) <> one_student =>
   students_table(i) = ~students_table(i))
end_process;
end_module;
```

Class exercise 7

1. Let a = mk_Account(010, 300, 5000), where the type Account is defined as follows:

```
Account = composed of
              account_no: nat1
              password: nat1
              balance: real
              end
```

Then evaluate the expressions:

a. a.account_no = ?

b. a.password = ?

c. a.balance = ?

d. modify(a, password -> 250) = ?

e. modify(mk_Account(020, 350, 4050), account_no -> 100,
   balance -> 6000) = ?

3. Let x be a variable of the type Date defined as follows:

Date = nat0 * nat0 * nat0

Let x = mk_Date(2002, 2, 6).

Then evaluate the expressions:

a. x(1) = ?

b. x(2) = ?

c. x(3) = ?

d. modify(x, 1 -> 2003)

e. modify(x, 2 -> 5, 3 -> 29)

f. modify(x, 1 -> x(1), 2 -> x(2))

4. Define a composite type Student that has the fields: name, date_of_birth, college, and grade. Write specifications for the processes: Register, Change_Name, Get_Info. The Register takes a value of Student and adds it to the external variable student_list, which is a sequence of students. Change_Name updates the name of a given student with a new name in student_list. Get_Info provides all the available field values of a given student in student_list.

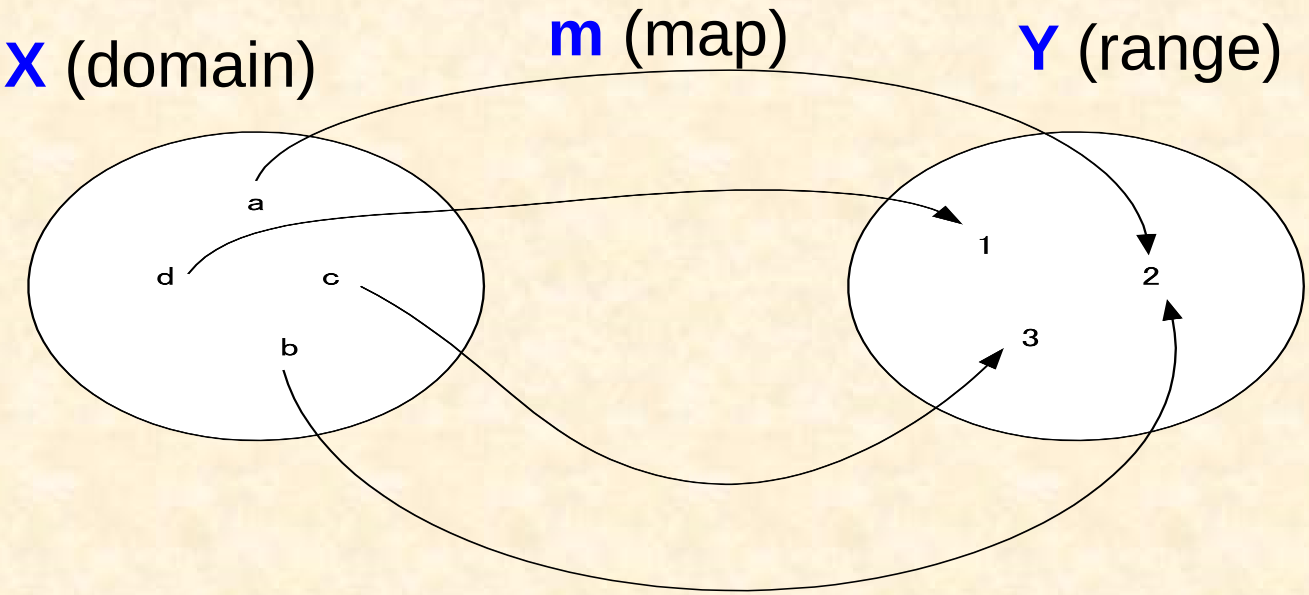
Map types

Contents:

- What is a map?
- The type constructor
- Operators
- Specification using maps

What is a map?

A map is a finite set of pairs, describing an association between two sets. It is a special function.



forall[x1, x2: X] | x1 <> x2 and m(x1) inset Y and m(x2) inset Y => m(x1) <> m(x2)

A map (or sometimes we call it "map value") is represented with a notation similar to the set notation:

{a_1 -> b_1, a_2 -> b_2, ..., a_n -> b_n}

Each a_i -> b_i (i =1,...,n) denotes a pair which is known as maplet.

For example, the map illustrated in the Figure on the previous slide is expressed as follows:

{a -> 2, b -> 2, c -> 3, d -> 1}

An empty map is expressed as:

{->}

Important property:

A map usually describes a many-to-one association: it allows the mapping from many elements in the domain to the same element in the range, but does not allow the mapping from the same element in the domain to different elements in the range.

The type declaration

A map type T is declared based on the domain type T1 and the range type T2 in the following format:

T = map T1 to T2

T contains all the possible maps that associate values in T1 with the values in T2.

Another example:

```
A = map nat to char
```

declares a map type A whose domain type is nat and range type is char.

Examples: possible maps (or map values) of type A:

```
{1 -> 'a', 2 -> 'b', 3 -> 'c', 4 -> 'd'}
{5 -> 'u', 15 -> 'v', 25 -> 'w'}
{10 -> 'x', 20 -> 'y'}
{50 -> 'r'}
{->}
```

Note that: domain type and range type of a map type can be an infinite set, although a concrete map value derived from the map type must contain only finite maplets (elements of a map).

Operators

(1) Constructors

Two constructors: map enumeration and
map comprehension.

(1.1) Map enumeration

The general format:

{a_1 -> b_1, a_2 -> b_2, ..., a_n -> b_n}

Examples:

{3 -> 'a', 8 -> 'b', 10 -> 'c'}

{“Beijing Jiaotong University" -> “China“, "Hosei University" -> "Japan", "University of Manchester" -> "U.K."}

{1 -> s(1), 2 -> s(2), 3 -> s(3)}

(1.2) Map comprehension

{a -> b | a: T1, b: T2 & P(a, b)} or

{a -> b | P(a, b)}

Example:
{x -> y | x: {5, 10, 15}, y: {10, 20, 30} & y = 2 * x} =
{5 -> 10, 10 -> 20, 15 -> 30}

defines a map.

The following map comprehension defines an illegal map:

{x -> y | x: {1, 2, 3}, y: {5, 10, 15, 20} & y > x * 5} =
{1 -> 10, 1 -> 15, 1 ->20, 2 -> 15, 2 -> 20, 3 -> 20}

Why?

(2) Other operators

(2.1) Map application

Let m be a map:

```
m: map T1 to T2;
```

Then, m can be applied to an element in its domain to yield an element in its range.

Example:

```
m(a)
```

denotes an application to element a in its domain.

Example: let

m1 = {5 -> 10, 10 -> 20, 15 -> 30}

Then

m1(5) = 10

m1(10) = 20

m1(15) = 30

Note that when m1 applies to number 2, for example, the result of the application is undefined:

m1(2) = undefined.

(2.2) Domain and range (dom, rng)

Let m be a map:

```
m: map T1 to T2;
```

Then, the domain of m is a subset of T1 and its range is a subset of T2, which can be obtained by applying the operators dom and rng, respectively.

```
dom: map T1 to T2 --> set of T1
```

Example:

let m1 = {5 -> 10, 10 -> 20, 15 -> 30}

Then

```
dom(m1) = {5, 10, 15}
```

The range operator rng yields, when applied to a map, the set of the second elements of all the maplets in the map.

```
rng: map T1 to T2 --> set of T2
rng(m) == {m(a) | a inset dom(m)}
```

Example: let m1 = {5 -> 10, 10 -> 20, 15 -> 30}.

Then,

```
rng(m1) = {10, 20, 30}
```

(2.3) Domain and range restriction to (domrt, rngrt)

Given a map and a set, sometimes we may want to obtain the submap of the map whose domain or range is restricted to the set. Such operations are known as domain restriction to and range restriction to, respectively.

```
domrt: set of T1 * map T1 to T2 --> map T1 to T2
domrt(s, m) == {a -> m(a) | a inset inter(s, dom(m))}

rngrt: map T1 to T2 * set of T2 --> map T1 to T2  rngrt(m, s)
 == {a -> m(a) | m(a) inset inter(s, rng(m))}
```

Examples: let

m1 = {5 -> 10, 10 -> 20, 15 -> 30}
s1 = {5, 10}.

Then,

domrt(s1, m1) = {5 -> 10, 10 -> 20}
rngrt(m1, s1) = {5 -> 10}

(2.4) Domain and range restriction by (domrb, rngrb)

In contrast to "domain restriction to" and "range restriction to" operations, sometimes we may want to derive a submap of a map whose domain or range is the subset of the domain or range of the map that is disjointed with a given set. Such operations are called domain restriction by and range restriction by, respectively.

```
    domrb: set of T1 * map T1 to T2 --> map T1 to T2
    domrb(s, m) == {a -> m(a) | a inset diff(dom(m), s)}

   rngrb: map T1 to T2 * set of T2 --> map T1 to T2
   rngrb(m, s) == {a -> m(a) | m(a) inset diff(rng(m), s)}
```

Examples: let

m1 = {5 -> 10, 10 -> 20, 15 -> 30}
s1 = {5, 10}.

Then,

domrb(s1, m1) = {15 -> 30}
rngrb(m1, s1) = {10 -> 20, 15 -> 30}

(2.5) Override (override)

Overriding is an operation for a union of two maps m1 and m2, denoted by override(m1, m2), with the restriction: if a maplet in map m2 shares the first element with a maplet in m1, the resulting map only includes the maplet in m2 as its element.

```
override: map T1 to T2 * map T1 to T2 -->
          map T1 to T2
override(m1, m2) == {a -> b |
    a: union(dom(m1), dom(m2)) &
    a inset dom(m2) => b = m2(a) and
    a notin dom(m2) => b = m1(a)}
```

Example: let

m1 = {5 -> 10, 10 -> 20, 15 -> 30},

m2 = {10 -> 5, 15 -> 50, 4 -> 20}.

Then,

override(m1, m2) =

{10 -> 5, 15 -> 50, 4 -> 20, 5 -> 10}

Notice: override is not commutative, that is,
override(m1, m2) <> override(m2, m1)

holds in general.

Example: compare override(m1, m2) to the following:

override(m2, m1) = {5 -> 10, 10 -> 20,
15 -> 30, 4 -> 20}

(2.6) Map inverse (inverse)

Map inverse is an operation that yields a map from a given map by exchanging the first and second elements of every maplet of the given map.

```
inverse: map T1 to T2 --> map T2 to T1
inverse(m) == {a --> b | a: rng(m), b: dom(m)
                 & a = m(b)}
```

Example: let m1 = {5 -> 10, 8 -> 20, 2 -> 30}

Then,

inverse(m1) = {10 -> 5, 20 -> 8, 30 -> 2}

However, if the map defines a many-to-one rather than one-to-one association between its domain and range, the application of the inverse operator is undefined.

(2.7) Map composition (comp)

Map composition is an operation that forms a another map from two given maps.

```
comp: map T1 to T2 * map T2 to T3 -->
      map T1 to T3
comp(m1, m2) == {a -> b | a: dom(m1),
                          b: rng(m2) &
                 exists[x: rng(m1)] |
                   x inset dom(m2) and
                   x = m1(a) and b = m2(x)}
```

Example: let

m1 = {5 -> 10, 8 -> 20, 2 -> 4},

m2 = {10 -> 5, 15 -> 5, 4 -> 20},

Then, the composition of m1 and m2 is:

comp(m1, m2) = {5 -> 5, 2 -> 20}

(2.8) Equality and inequality (=, <>)

We use m1 = m2 to mean m1 is identical to m2, and m1 <> m2 to mean m1 is different from m2. Formally,

```
m1 = m2 <=> dom(m1) = dom(m2) and
            rng(m1) = rng(m2) and
            forall[a: dom(m1), b: rng(m1)] |
                 b = m1(a) <=> b = m2(a)

m1 <> m2 <=> not m1 = m2
```

Specification using maps

Let us reconsider defining the type Account with a map type. Since every customer's account number is unique and it is common to allow one customer to have only one account of the same kind in the same bank, the customer account can be modeled as a map from the account number to the account data including password and balance.

```
Account = map AccountNumber to AccountData;

AccountNumber = nat;
AccountData = composed of
                  password: nat
                  balance: real
                  end;
```

We then redefine the processes Check_Password, Withdraw, and Show_Balance as follows:

```
process Check_Password(card_id: AccountNumber, pass: nat)
                                        confirm: bool
ext rd account_file: Account
post card_id inset dom(account_file) and
       account_file(card_id).password =  pass and
       confirm =  true
       or
       card_id notin dom(account_file) and
       confirm = false
comment
   If the given account number card_id and password pass
   matches with the account_file, the output confirm becomes
   true; otherwise, it becomes false.
end_process;
```

```
process Withdraw(card_id: AccountNumber, amount: real)
                              cash: real
 ext wr account_file: Account
 pre card_id inset dom(account_file) and amount <=
    account_file(card_id).balance
 post account_file = override(~account_file, {card_id ->
        mk_AccountData(~account_file(card_id).password,
             ~account_file(card_id).balance - amount)} and
    cash = amount
comment
   The precondition requires that the provided card_id be
    registered in the account_file and the requested amount to
    withdraw be less than or equal to the current balance. The
    updating of the current balance of the account with the
    account number card_id is expressed by a map overriding
    operation: the updated balance is the result of subtracting
    the requested amount from the current balance.
end_process;
```

```
process Show_Balance(card_id:
                        AccountNumber) bal: real
ext rd account_file: Account
pre card_id inset dom(account_file)
post bal = account_file(card_id).balance
comment
  The account number card_id must exist in the
   account_file before the execution of the
   process. The assignment of the current
   balance to the output variable bal is reflected
   by a map application in the postcondition.
end_process;
```

Class exercise 8

1. Let m1 and m2 be two maps of the map type from nat0 to nat0;

   m1 = {1 -> 10, 2 -> 3, 3 -> 30},

   m2 = {2 -> 40, 3 -> 1, 4 -> 80}, and s = {1, 3}.

Then, evaluate the expressions:

a. dom(m1) = ?

b. dom(m2) = ?

c. rng(m1) = ?

d. rng(m2) = ?

e. domrt(s, m1) = ?

f. domrt(s, m2) = ?

g. rngrt(m1, s) = ?

h. rngrt(m2, s) = ?

i. domrb(s, m1) = ?

j. domrb(s, m2) = ?

k. rngrb(m1, s) = ?

l. rngrb(m2, s) = ?

m. override(m1, m2) = ?

n. override(m2, m1) = ?

o. inverse(m1) = ?

p. inverse(m2) = ?

q. comp(m1, m2) = ?

r. comp(m2, m1) = ?

s. m1 = m2 <=> ?

t. m1 <> m2 <=> ?

2. Define BirthdayBook as a map type from the type Person (with the fields: id, name, and age) to the type Birthday, and specify the processes: Register, Find, Delete, and Update. All the processes access or update the external variable birthday_book of the type BirthdayBook. The process Register adds a person's birthday to birthday_book. Find detects the birthday for a person in birthday_book. Delete eliminates the birthday of a person from birthday_book. Update replaces the wrong birthday registered in birthday_book with a correct birthday.

The union types

A compound object may come from different types. For example, a component of a world wide web home page may contain normal text, pictures, audio data, and so on, each belonging to a different category. The union types allow us to define such compound objects.

The outline of this part:

- Union type declaration
- Is function
- A specification with union types

Union type declaration

Let T1, T2, ..., Tn be n types. Then, a union type T constituted from these types is declared in the format:

T = T1 | T2 | ... | Tn

A value of T can come from one of the types T1, T2, ..., Tn.

It is important to keep T1, T2, ..., Tn disjoint so that any value of type T can be precisely determined to belong to which constituent type.

Example:

```
Color = {<Red>, <Blue>, <Yellow>}
Key = char
Digits = set of nat
```

the union type Hybrid can then be declared as:

```
Hybrid = Color | Key | Digits
```

the following values belong to the type Hybrid:

```
<Red>
<Blue>
'b'
'5'
{10, 20, 100}
```

No operators can be built on a union type except the equality (=) and inequality (<>).

For example,

```
<Red> = <Blue> <=> false
<Red> <> {3, 5, 8} <=> true
'b' = 'b' <=> true
```

is function

When writing specifications there may be a situation that requires a precise type of a given value. Such a type can be determined by applying a built-in function known as is function:

is_T(x)

This function is a predicate that yields true when the type of value x is T (any type is possible); otherwise, it yields false.

Examples:

is_Color(<Red>) <=> true

is_Hybrid(<Red>) <=> true

Specification with a union type

Suppose we want to write a program that scans a a specification in SOFL and records different kinds of tokens in different tables. We first declare Token as a union type:

Token = EnglishLetter | Identifier |SpecialCharacter

where EnglishLetter, Identifier, and SpecialCharacter are supposed to have been declared before.

We then build a process Record_Token to record different tokens obtained by scanning the current text in different tables.

```
 process Record_Token(token: Token)
 ext wr english_char_table: seq of EnglishLetter
       wr identifier_table: seq of Identifier
       wr special_char_table: seq of SpecialCharacter
 post (is_EnglishLetter(token) =>
 english_char_table = conc(~english_char_table, [token])) and
         (is_Identifier(token) =>
 identifier_table = conc(~identifier_table, [token])) and
         (is_SpecialCharacter(token) =>
 special_char_table = conc(~special_char_table, [token]))
 comment
  The token is recorded in the corresponding table.
end_process
```

Class exercise 9

1. Define a union type School with the constituent types ElementarySchool, JuniorHighSchool, HighSchool, and University, assuming that all the constituent types are given types.

2. Let s1 and s2 be two variables of the type set of Hybrid. Let s1 = {<Red>, 3, 'b'} and s2 = {<Blue>, 'a', 'b', 9}. Evaluate the expressions:

   a. card(s1) = card(s2) <=> ?

   b. union(s1, s2) = ?

   c. inter(s1, s2) = ?

   d. diff(s1, s2) = ?

Hierarchical CDFDs and Modules

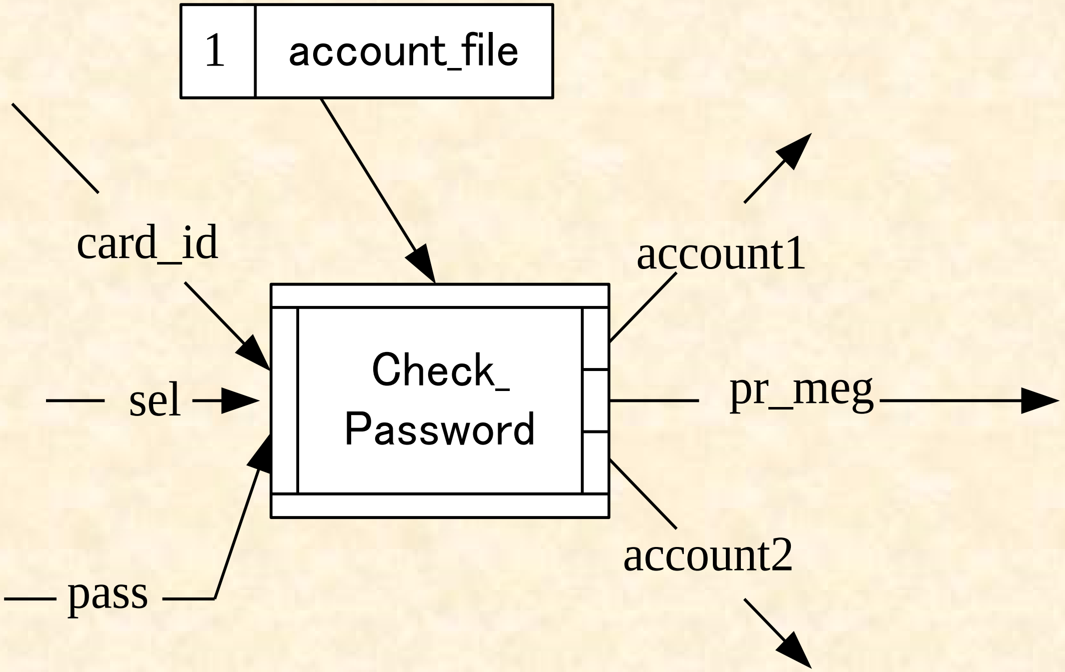
The motivation for building hierarchical CDFDs:

- It is almost impossible to construct only one level CDFD and module for a complex system.
- It is necessary to organize the developers within a team so that each of them can concentrate on a single part of the entire system independently and all the developers can conduct their activities concurrently.

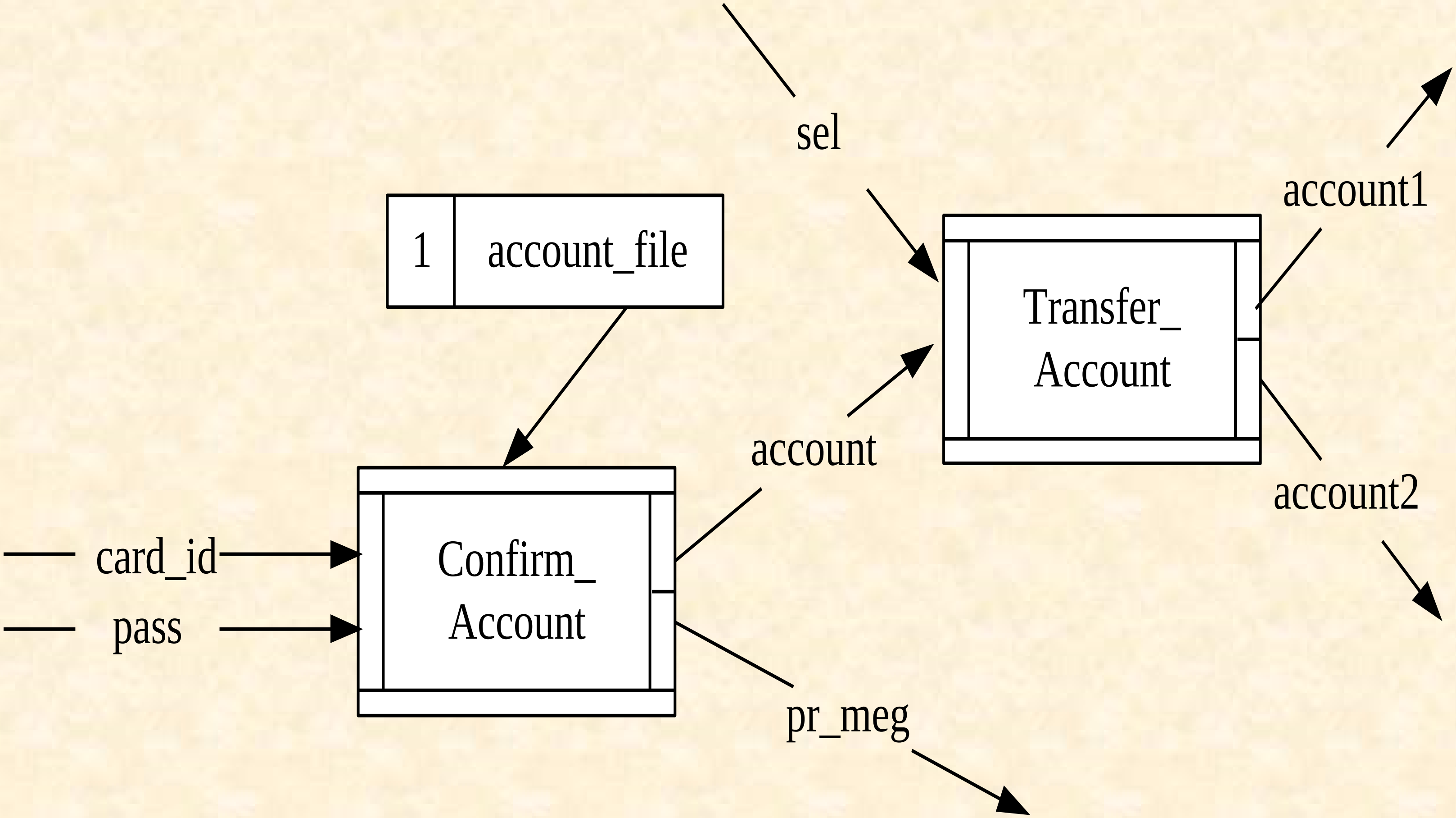
Process decomposition

Process decomposition is an activity to break up a process into a lower level CDFD.

Example:



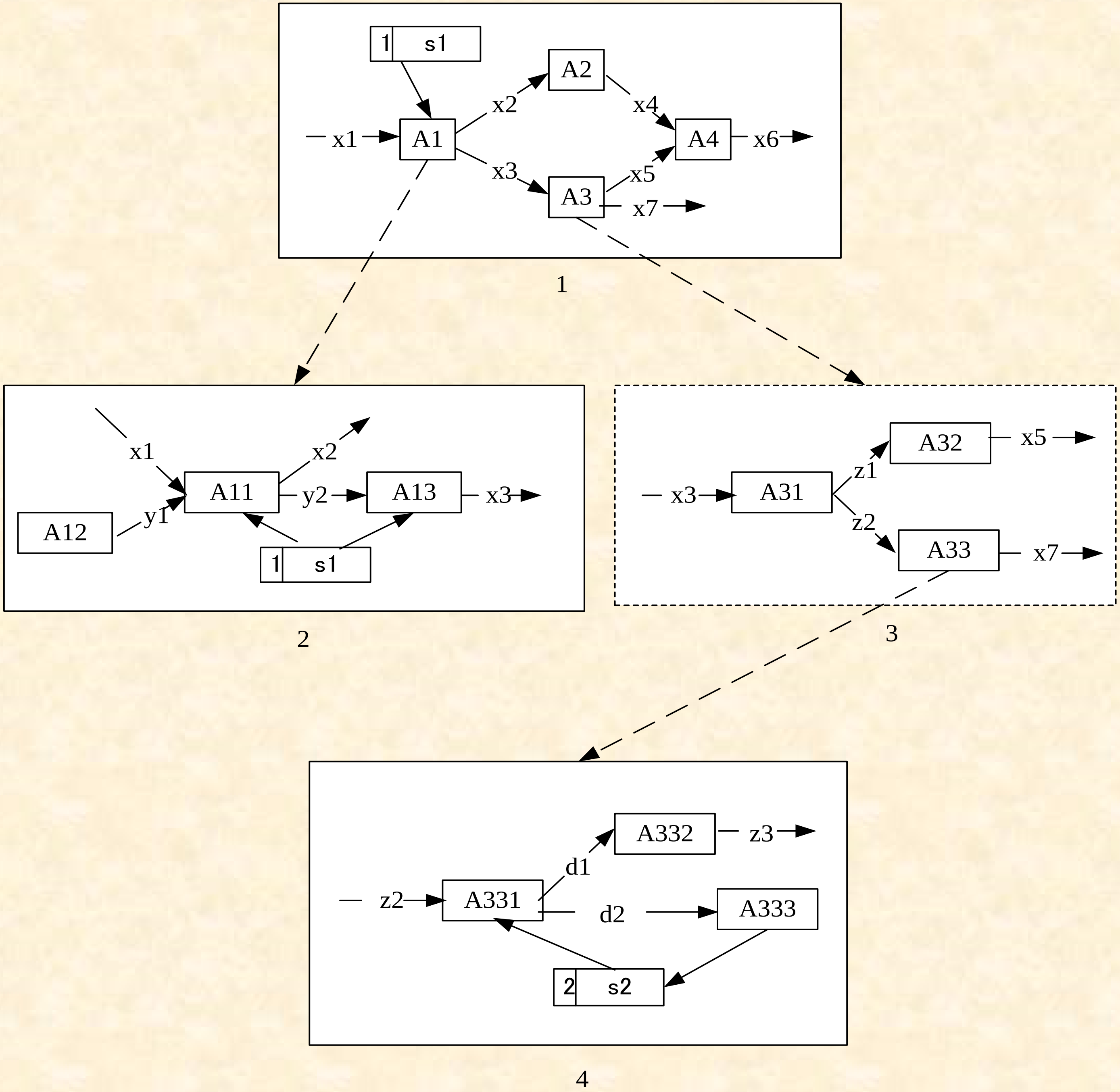
This process can be decomposed into the CDFD on the next slide:


sel
1
account_file
Transfer_ Account
account1
account2
account
Confirm_ Account
card_id
pass
pr_meg

```
module Check_Password_decom / SYSTEM_ATM;
var    ext  account_file: set of Account;
behav CDFD_2; /* Assume the CDFD in Figure 2 is numbered
   2. */
process Init()
end_process;

process Confirm_Account(card_id: nat0, pass: nat0)
                         account: Account | pr_meg: string
 ...
end_process;
process Transfer_Account(sel: bool, account: Account)
                          account1: Account | account2: Account
 ...
end_process;
end_module;
```

Definition If process A is decomposed into a CDFD, we call the CDFD the decomposition of process A and process A the high level process of the CDFD.

Decompositions can be carried out until all the lowest level processes are simple enough to be formally defined. As a result, a hierarchical CDFDs will be constructed.


1
s1
A2
x2
x4
x1
A1
A4
x6
x3
x5
A3
x7
1
2
x1
x2
A11
y2
A13
x3
A12
y1
1
s1
3
A32
x5
z1
x3
A31
z2
A33
x7
4
A332
z3
d1
z2
A331
d2
A333
2
s2

The associated module hierarchy of this CDFD hierarchy is outlined as follows:

```
module SYSTEM_Example;
 ...
var    s1: Type1;
behav CDFD_1;
process Init;
process A1

decom: A1_decom; /*Module A1_decom is associated with the
                                decomposition of A1. */
end_process;

process A2;

process A3
decom: A3_decom; /*Module A3_decom is associated with the
   decomposition                                of A3. */
end_process;

process A4;
end_module;
```

```
module A1_decom;
 ...
 var  ext s1: Type1;
 behav CDFD_2;
 process Init;
 process A11;
 process A12;
 process A13;
 end_module;
 module A3_decom;
   ...
 behav CDFD_3;
 process Init;
 process A31;
 process A32;
 process A33
  decom: A33_decom; /*Module A33_decom is associated with the
                                   decomposition of A33. */
 end_process;
 end_module;
```

```
module A33_decom;
   ...
 var  s2: Type1;
 behav CDFD_4;
 process Init;
 process A331;
 process A332;
 process A333;
end_module.
```

Handling stores in decomposition

Generally speaking, a store accessed by a high level process must also be drawn in the decomposition of the process, and must be accessed in the same way, probably by several processes.

Definition An external store of a CDFD is a store that is accessed by a high level process of the entire specification.

There are two kinds of external stores:

(1) a store local to a high level CDFD. Usually it is declared after the keyword ext.

(2) existing external store that is global to all the CDFD in the hierarchy. Usually such stores are declared with the sharp mark, such as

ext #file: int

Definition A local store of a CDFD is a store that is introduced for the first time in the CDFD.

Scope

When defining a module, we may need to refer to some type or function or any components defined in another module. In this case, we need a scope rule to constrain the reference of those components.

To give a formal definition of the scope rule, we need the following definitions.

Definition Let process A be defined in module M1. If A is decomposed into a CDFD associated with module M2, we say that process A is decomposed into module M2.

Definition If process A defined in module M1 is decomposed into module M2, M2 is called child module of M1, while M1 is called parent module of M2.

Definition Let A_1, A_2, ..., A_n be a sequence of modules where n >1. If A_1 is the parent module of A2, and A2 is the parent module of A_3, ..., and A_n - 1 is the parent module of A_n, then we call A1 ancestor module of A_n and A_n descendant module of A_1.

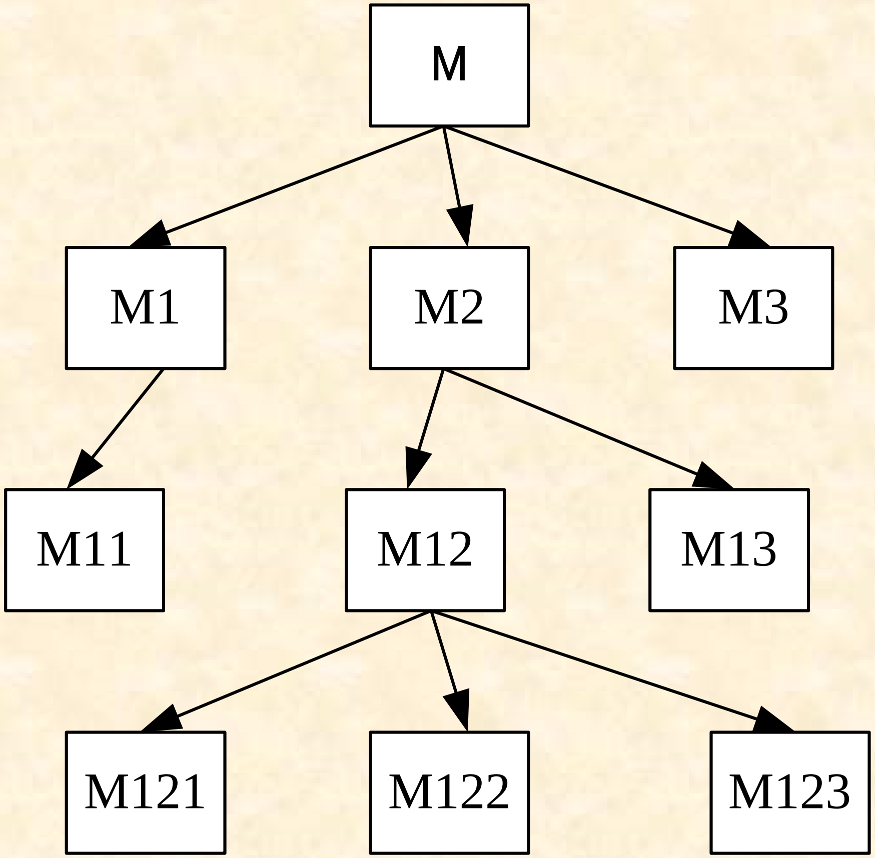
Definition If module A is neither ancestor module nor descendant module of module B, A and B are called relative modules.

Example:

(1) M2 is the parent module of M12.

(2) M2 is an ancestor module of M122.

(3) M11 and M13 are relative modules. M13 and M121 are relative modules.



Scope rules:

- Let M_c be a type or constant identifier declared in module M1. Then, the scope of the effectiveness of this declaration is M1 and its all descendant modules.
- Let M_c be declared in both module M1 and its ancestor module M. Then, M_c declared in M1 has higher priority in its scope than the same M_c declared in M.
- Let M_f be a function defined in M1. Then, the scope of M_f is M1 and its all descendant.
- Let M_c be a type, constant identifier, or function declared in module M1. Then, if M_c is used in its relative module M2, it must be used in the form:

M1.M_c

Approaches to constructing specifications

The outline of this part:

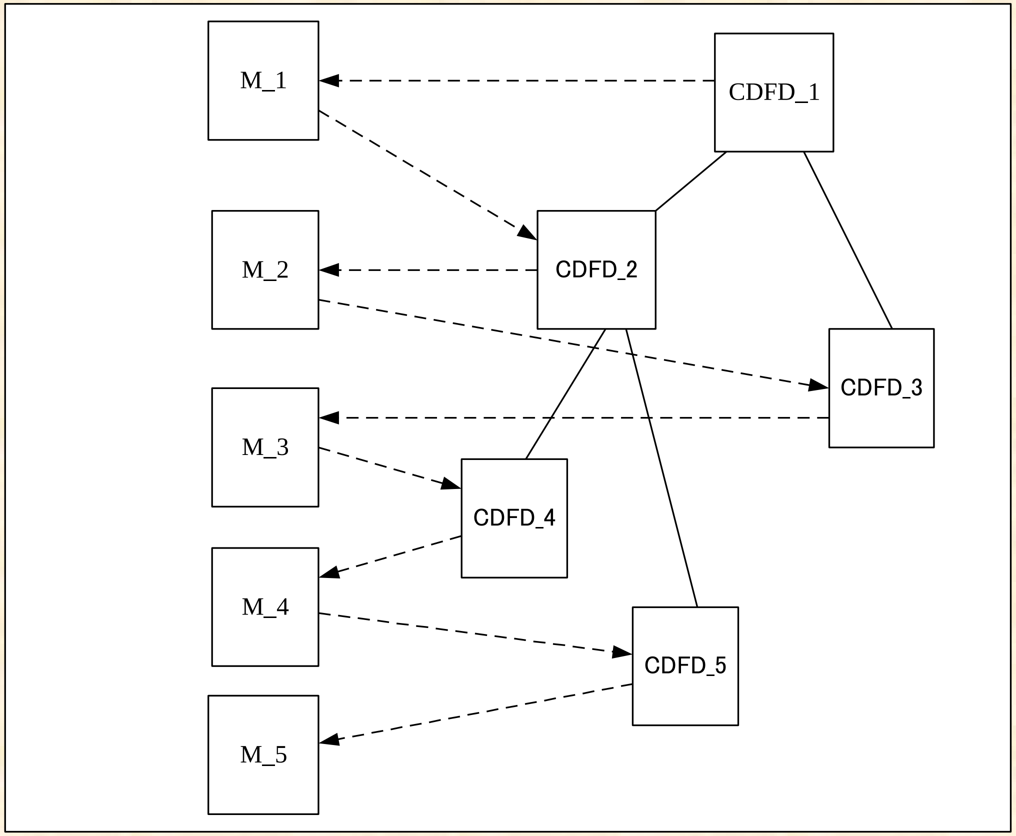
- The top-down approach
- The bottom-up approach
- The middle-out approach
- Comparison of the approaches

The top-down approach

Two strategies can be taken in the top-down approach: the CDFD-module-first strategy and
the CDFD-hierarchy-first strategy

(1) The CDFD-module-first strategy

The fundamental idea of this strategy is that after a CDFD is constructed, its associated module must be defined precisely, before any decomposition of processes in this CDFD takes place. After both the CDFD and module are finalized, another decomposition can take place. Such a process goes on until no process needs further decomposition.

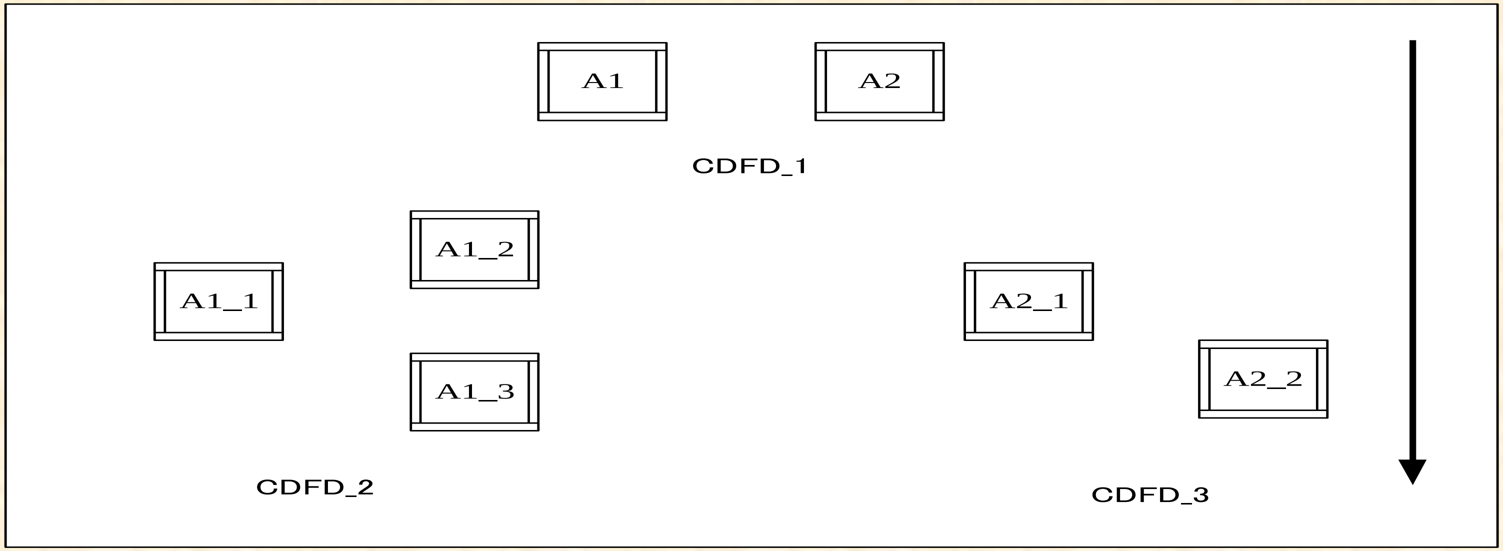


The following guidelines may be useful in determining whether a process needs a decomposition:

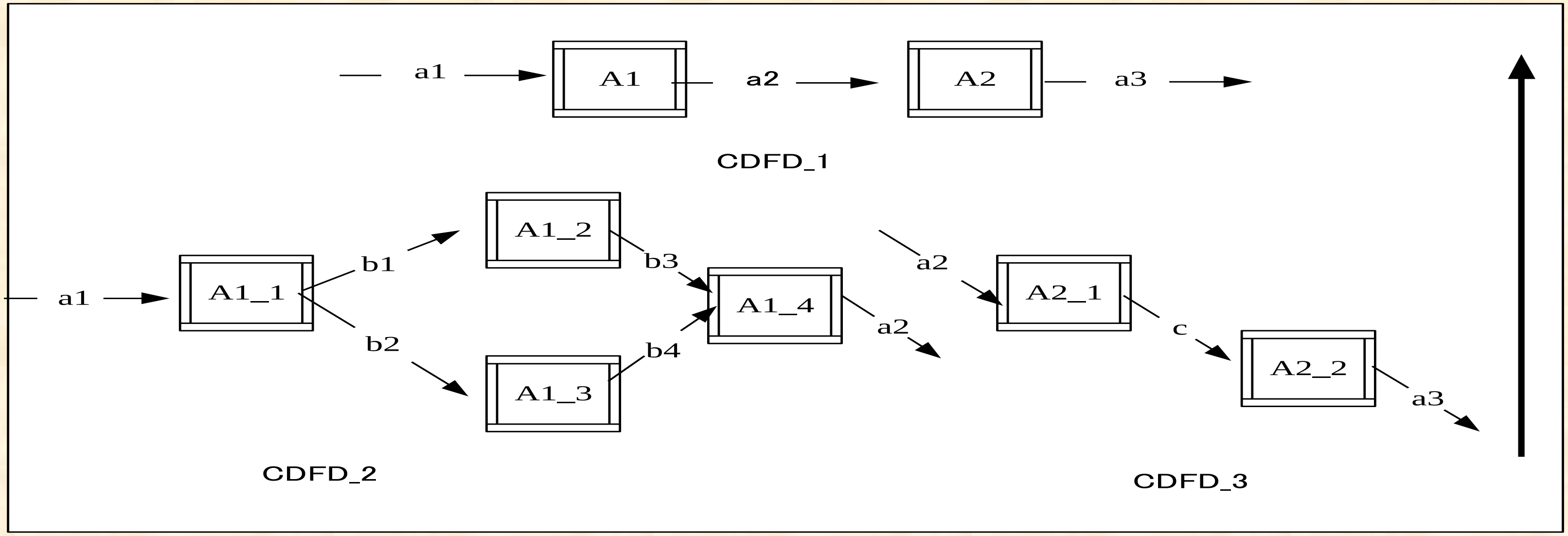
1. If the relation between the input and output data flows of a process cannot be expressed without further information, the decomposition of this process should be considered.
2. If the behavior of a process involves a sequence of actions, this process needs to be decomposed.
3. If the postcondition of a process is too complex to be written in a concise manner, it may need decomposition.

(2) The CDFD-hierarchy-first strategy

Building a specification using the CDFD-hierarchy-first strategy starts with the construction of the CDFD hierarchy by decomposition of processes, and then proceeds to define the modules of the CDFDs involved in the CDFD hierarchy, after it is completed.



(a) Top-dwon for processes

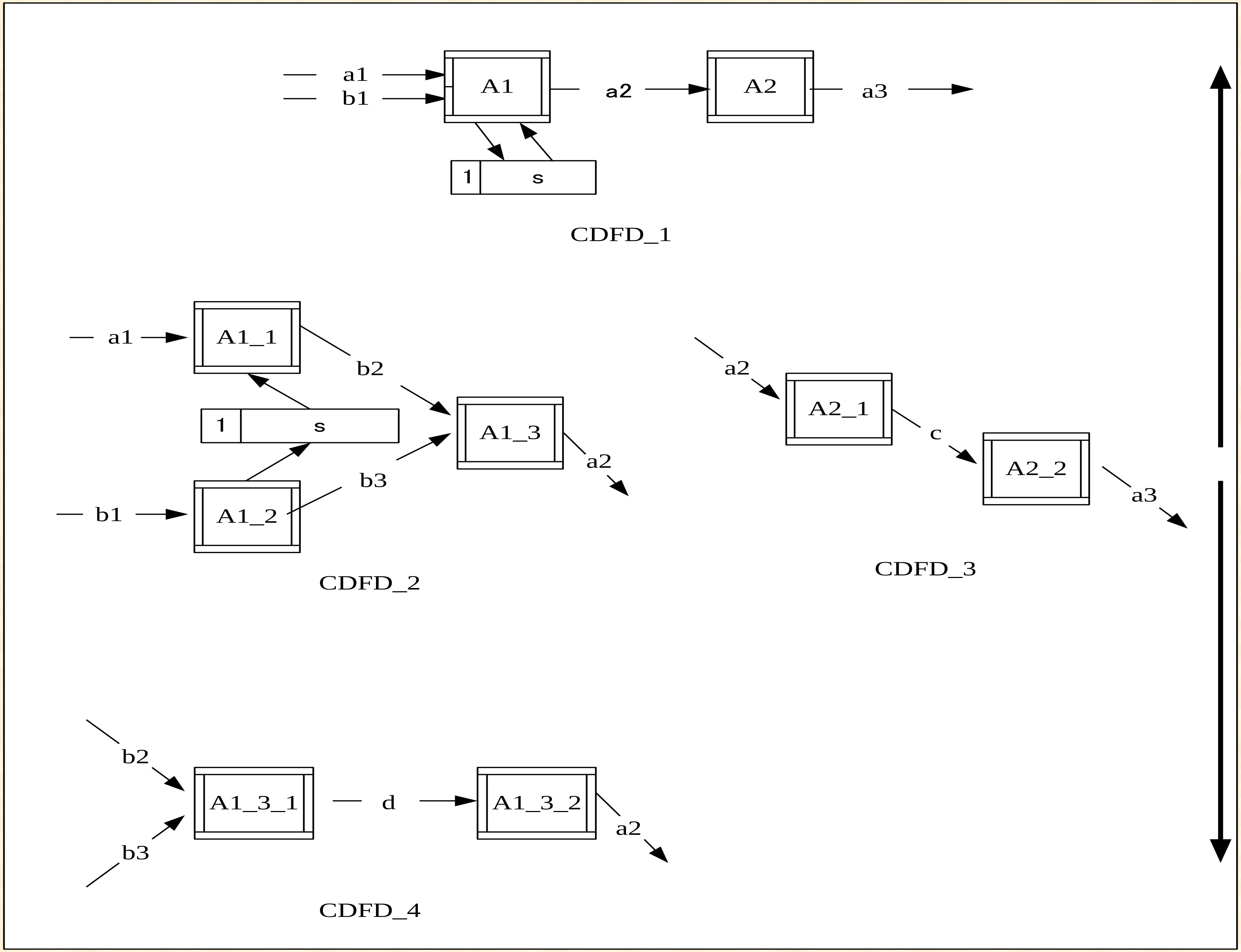


(b) Bottom-up for data flows and stores

The middle-out approach

Constructing a specification by the middle-out approach usually starts with the building of the CDFDs and modules modeling the functions that are most familiar to the developer and crucial to the system. Then, the system is built by

- decomposing some of the introduced processes
- synthesizing some of those processes to form high level processes.


a1
b1
A1
a2
A2
a3
1
s
CDFD_1
a1
A1_1
b2
1
s
A1_3
a2
b3
b1
A1_2
CDFD_2
a2
A2_1
c
A2_2
a3
CDFD_3
b2
A1_3_1
d
A1_3_2
a2
b3
CDFD_4

When synthesizing CDFDs into high level processes, we can follow the following guidelines:

- If there are more than two input data flows to different starting processes of a CDFD, the CDFD needs to be abstracted into a high level process that defines precisely the relationship among those input data flows.
- If two processes in a CDFD access the same store in both reading and writing manner, this CDFD needs to be considered for abstraction to define that concurrent execution of the two processes is impossible (e.g., processes A1_1 and A1_2 in CDFD_2).
- If two CDFDs have relations in terms of data flows, they need to be abstracted into high level processes and the connections between these processes need to be formed in the high level CDFD (e.g., process A1_3 in CDFD_2 and process A2_1 in CDFD_3 share the same data flow a2).

Comparison of the approaches

1. The advantages and weakness of the top-down approach:

(1) Advantages:

- It is effective and intuitive in providing sub-goals or sub-tasks to support the current goal or task, and in developing ideas with little information (abstraction) into ideas with more information (decomposition).
- It provides a good global view of data flows and stores that may be used across CDFDs at different levels, thus the consistency in using data flows and stores can be well managed during the decomposition of high level processes.

(2) Weakness:

It may cause frequent modifications of high level processes, data flows, stores, and even the entire CDFDs, as with the progress of decomposition of high level processes, due to the lack of sufficient knowledge about what data flows and stores will be used or produced by the processes in the lower level CDFDs.

2. The advantages and weakness of the middle-out approach

(1) Advantages:

- It may be more effective and natural than the top-down approach, because it always starts with modeling the most familiar and crucial functions.
- It also takes a flexible way to utilize the top-down and the bottom-up approaches. Taking which approach usually stems from natural demands during the construction of the entire specification.

(2) Weakness:

The developer may not be easy to take a global view of the specification in the early stage, thus data flows, stores, and processes created in different CDFDs may overlap or defined inconsistently.

3. How to use the top-down and the middle-out approach?

- Use the middle-out approach for requirements analysis and requirements specification constructions, especially for the semi-formal ones, because the most familiar and important functional requirements are often focused in the early stage of requirements analysis.
- Use the top-down approach for design, because the designer usually has a fair understanding of the functional requirements after studying the semi-formal requirements specification, and needs to take a global view in structuring the entire system.

Class exercise 10

1. Answer the questions:

a. what is a hierarchy of CDFDs?

b. what is a hierarchy of modules?

c. what is the relation between module hierarchy and CDFD hierarchy?

d. what is the relation between a CDFD and its high level process in a CDFD hierarchy?

e. what does it mean by saying that modules M1 and M2 are relative modules?

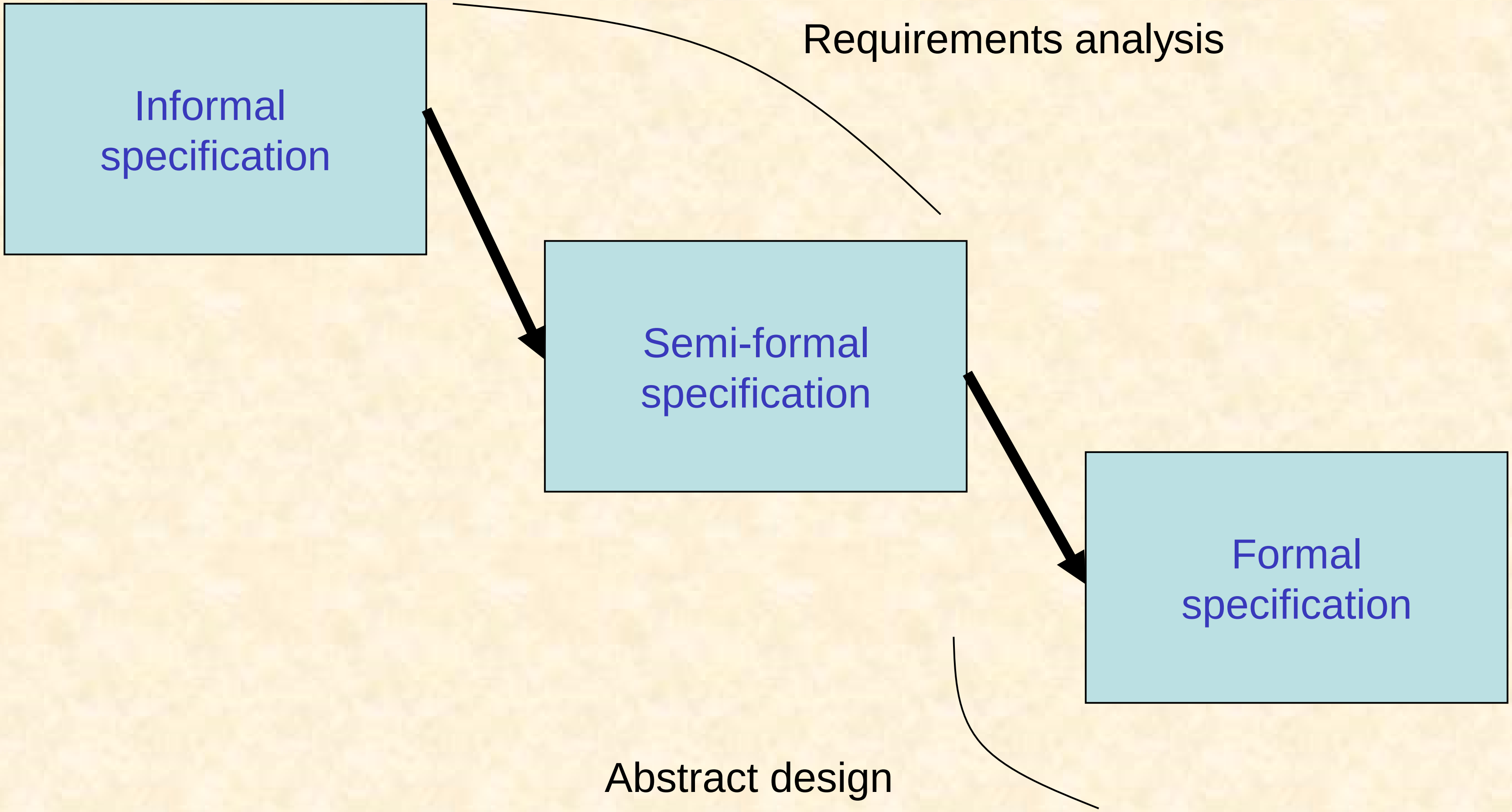
f. what is the scope of a variable, type identifier, constant identifier, invariant, function, and a process?

II.5 The SOFL Three-Step Approach to Writing Formal Specifications

Question: how can a formal specification be effectively constructed with good qualities (completeness, consistency, validity)?

The SOFL three-step approach is a solution!

Three-step formal specification:



Requirements analysis

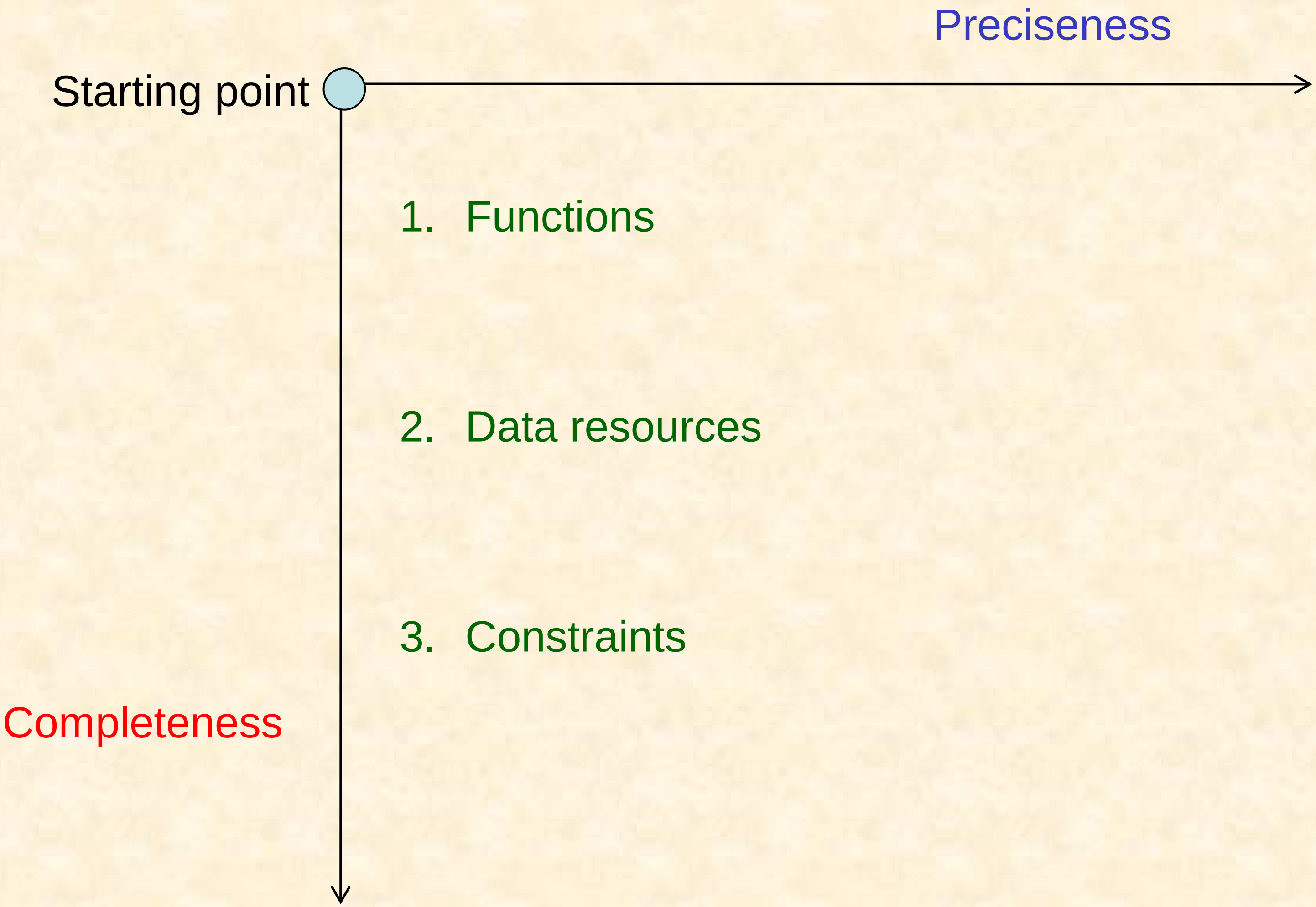
Informal specification

Semi-formal specification

Formal specification

Abstract design

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



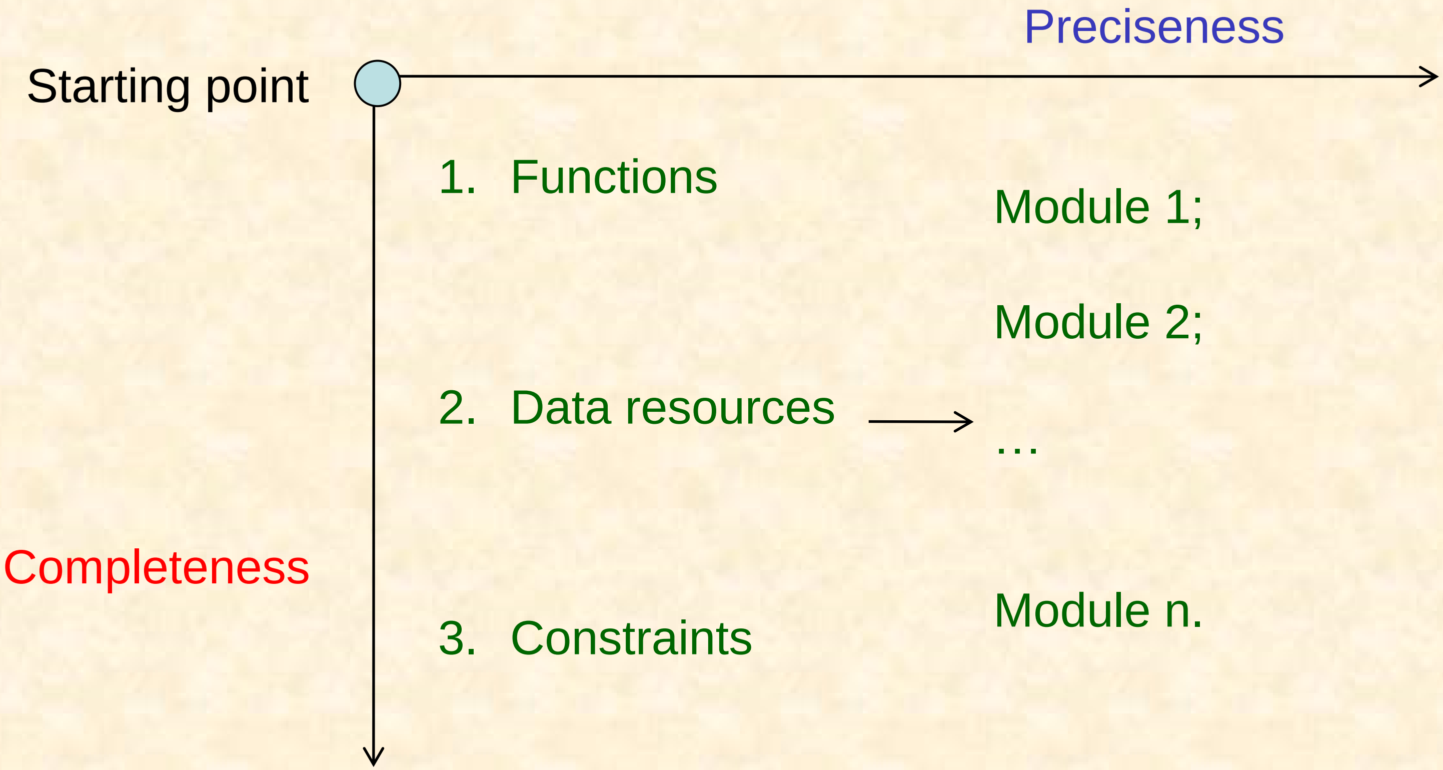
Preciseness

Starting point

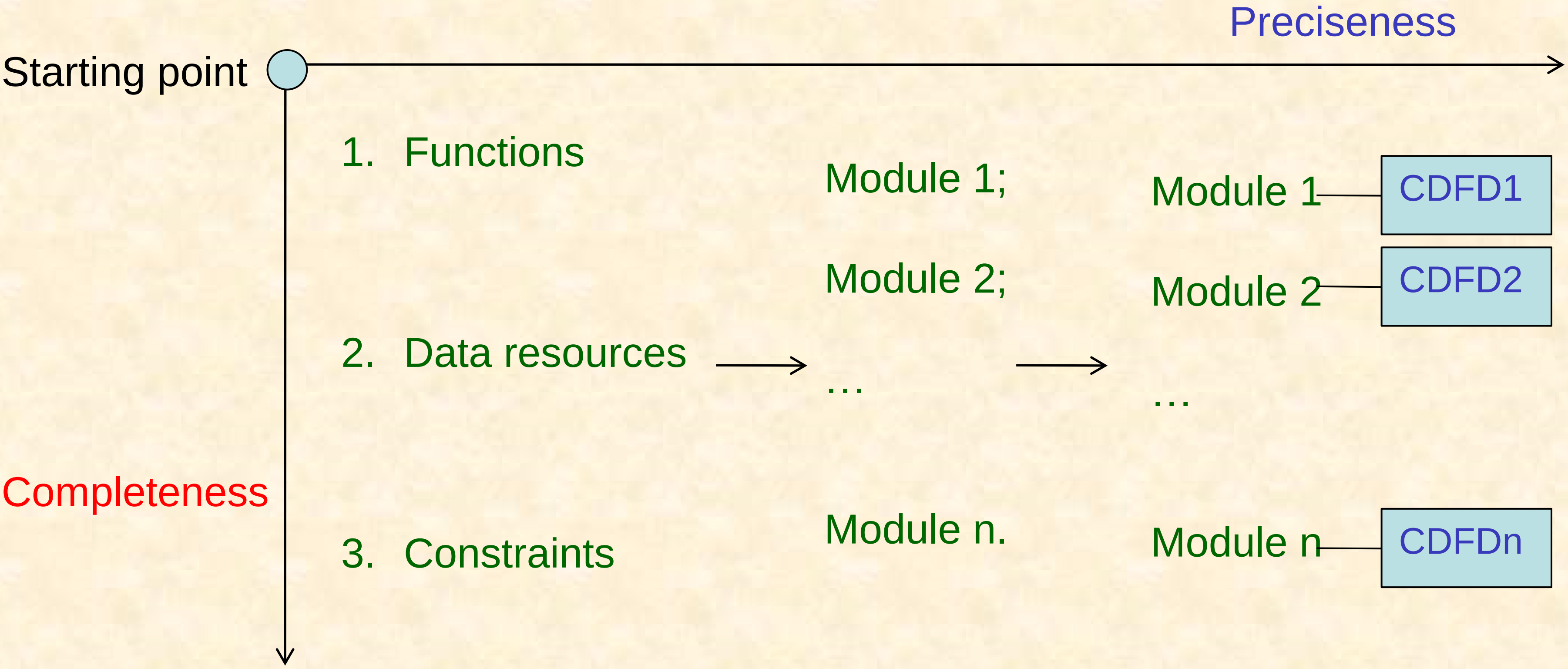
1. Functions
2. Data resources
3. Constraints

Completeness

Tasks for semi-formal specification: (1) Grouping related functions, data resources, and constraints into SOFL modules. (2) Defining necessary data types and variables. (3) Defining the function of each process using pre- and post-conditions at informal level.



Tasks for formal specification: (1) Describe system architecture using hierarchical CDFDs (condition data flow diagram). (2) Formalize the specification of each process in pre- and post-conditions.



A simplified ATM software

1．Informal specification (requirements analysis)

（1）**Functions**:

- Receive commands from the customer
- Confirm the customer’s card id and password
- Withdraw
- Inquire about the customer’s account balance

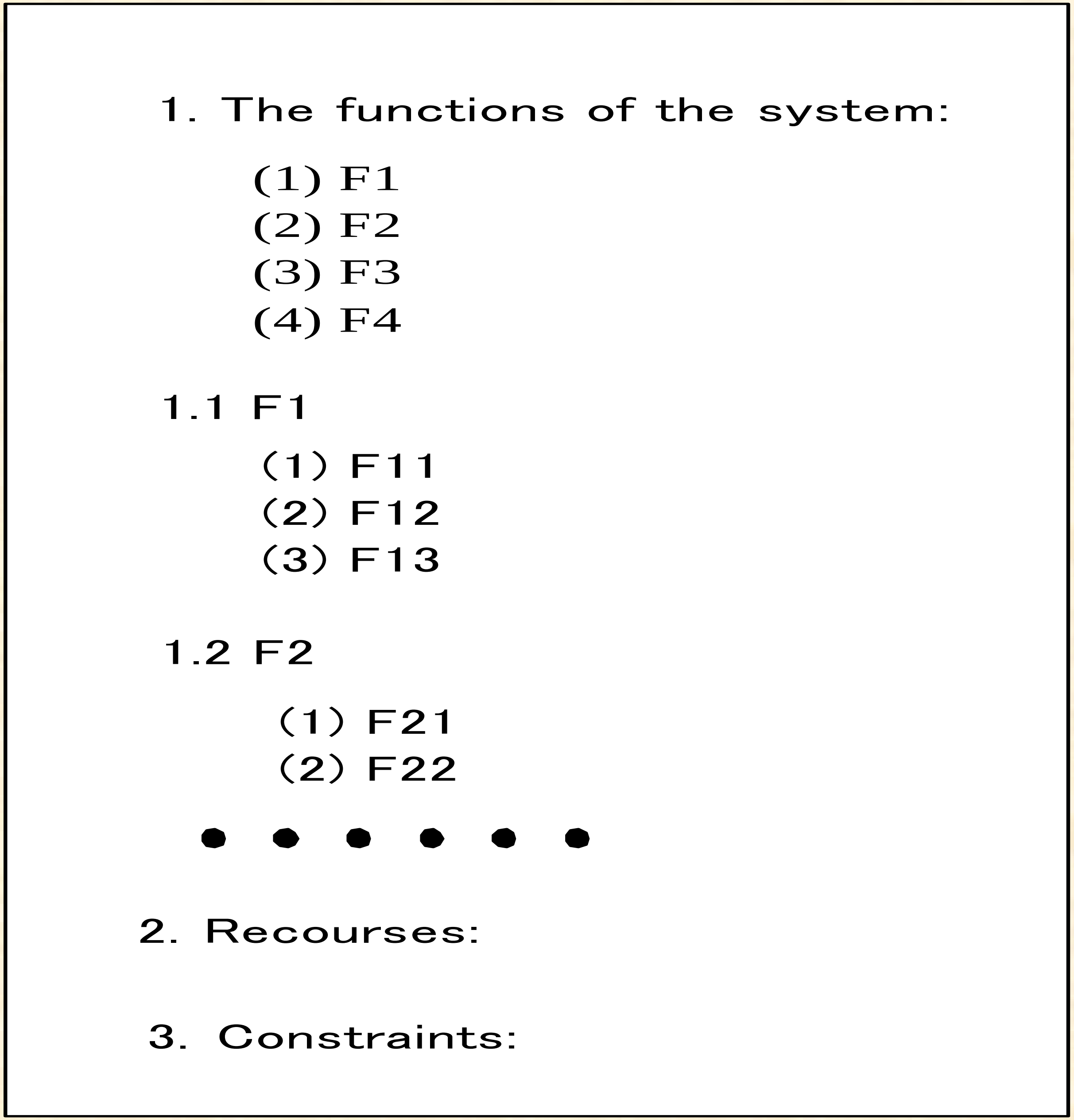
（2）**Data resources**:

- Customers’ account information

（3）**Constraints**：

- A customer can only access to his or her own account。
- No customer is allowed to borrow money from the bank。

Hierarchy of informal specification:



1. The functions of the system:

(1) F1
(2) F2
(3) F3
(4) F4

1.1 F1

(1) F11
(2) F12
(3) F13

1.2 F2

(1) F21
(2) F22

● ● ● ● ● ●

2. Recourses:

3. Constraints:

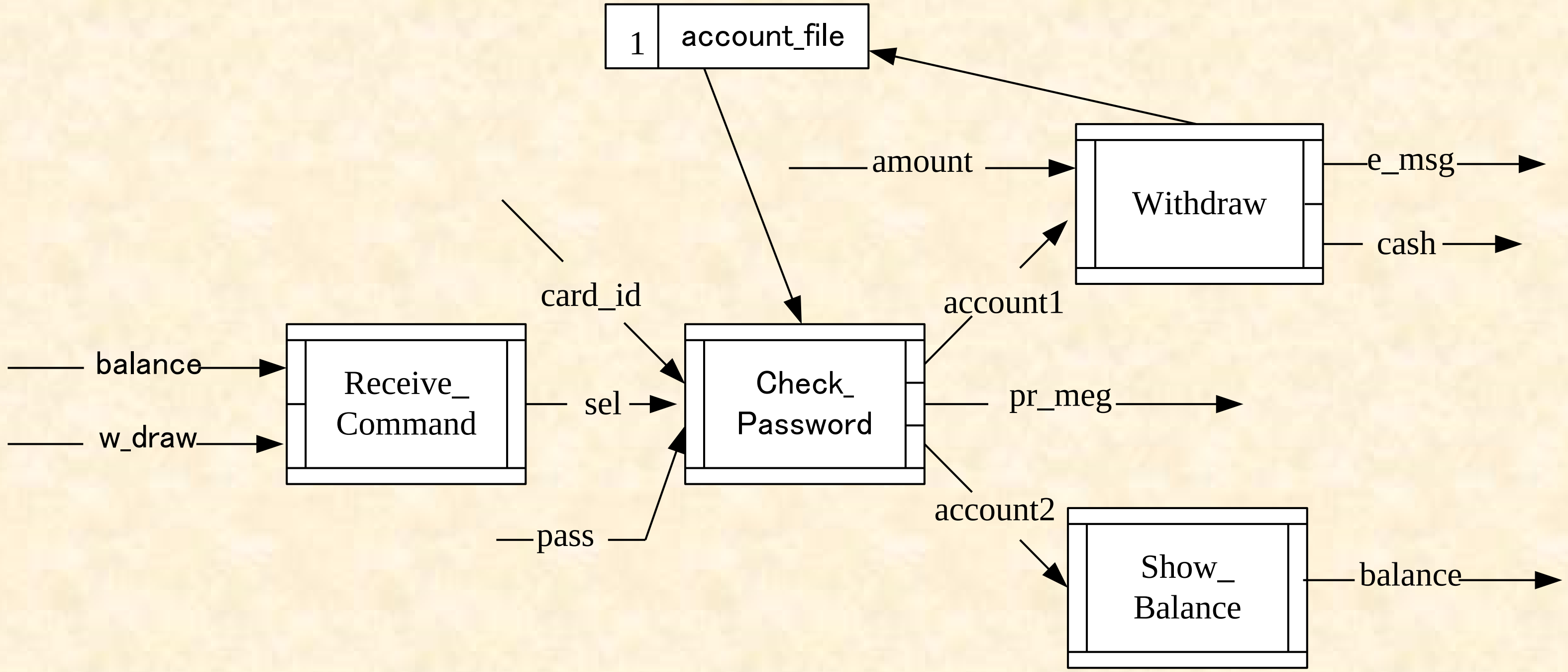
2. Semi-formal specification (requirements analysis):

```
module SYSTEM_ATM
    type
    Account = composed of
                       account_no: nat
                       password: nat
                       balance: real
                     end
 var
    account_file: set of Account;
 inv
     Account balance must be greater than or equal to zero.

 behav CDFD_No1;
 …
```

```
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 not greater 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.
end_process;

...

end_module
```

3. Formal specification (abstract design)

The top level CDFD:



```
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] | x.balance >= 0;

 behav CDFD_No1;
 ...
```

```
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;
end_module
```

Small Project (40%)

A stock reservation and purchase system is required to have the following functions and constraints:

1. Register a customer (name, age, bank account, address, email address, telephone number, and purchased stocks)
2. Register a stock (name, price, unit for sale, limit for selling to one customer)
3. Cancel the information of a registered customer
4. Cancel the information of a registered stock

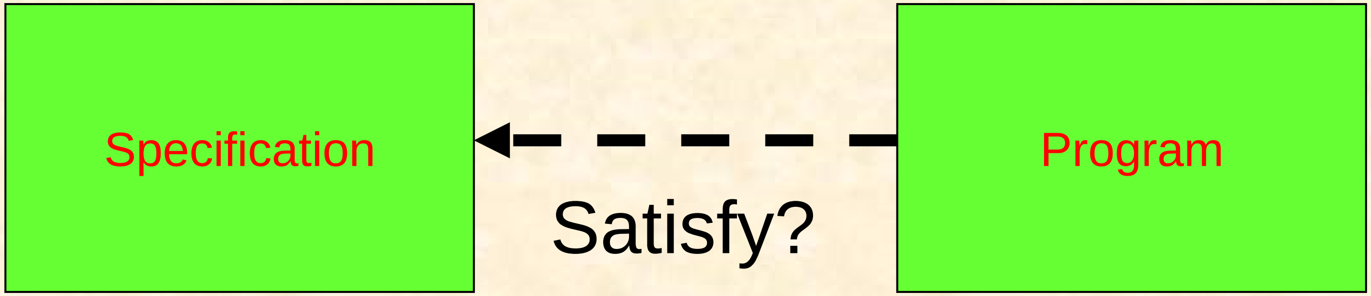
5. Purchase a stock (the purchased stock details should be recorded in the customer's information and the payment for the stock should be made from the customer's bank account)
6. Sell a stock by a customer (the money resulted from the sale of stock should be put into the customer's bank account)
7. For each kind of stock, a customer can only buy at most 1000 units of the stock.
8. A stock name is unique.
9. A customer's name is also unique.
10. The currency used in the system is RMB.

Requirements for students

(1) Take the SOFL three-step approach to construct a formal specification for the stock system. That is, write an informal specification, semi-formal specification, and formal specification.

(2) Independently finish this small project and and submit it to Dr. Wang Ying before 27th December 2022.

III. Hoare Logic for Formal Verification of Program Correctness



Formal verification or proof

- How to ensure that the program is correct with respect to the specification?
- What does the correctness of the program mean?

Partial and total correctness

(1) Partial correctness

Consider the Hoare triple:

{P} S {R}

It shows the partial correctness of program S with respect to P and R. **This means that If the precondition P is true before initiation of a program S, then the postcondition R will be true on its completion, we say that S is correct with respect to P and R.**

Note that the termination of S is assumed.

(2) Total correctness

Program S is totally correct with respect to predicates P and R iff S is partially correct and S terminates; that is,

Total correctness of S =

{P} S {R} +

Termination of S

Hoare logic is designed for the proof of partial correctness of a program. It is established on the basis of first order logic, but includes additional axioms for reasoning about programs.

The key of proving the termination of program S requires finding a loop variant, a mathematical function indicating that each repetition of a loop statement (e.g., while-loop) changes the loop-variable towards making the loop condition evaluate to *false*.

Program execution

One of the most important properties of a program is whether it carries out its intended function.

This property is known as

functional property of program.

Two ways to express the functional property of a program:

(1) List every initial state before the execution of the program and the corresponding final state after the execution terminates.

Examples:

```
{(x, 5), (y, 1)}        initial state 1
int tem = 0;
tem := x;               Swap program
x := y;
y := tem;
{(x, 1), (y, 5)}        final state 1
```

```
{(x, 9), (y, 3)}          initial state 2
int tem = 0;
tem := x;                  Swap program
x := y;
y := tem;
{(x, 3), (y, 9)}          final state 2
```

```
{(x, 20), (y, 10)}        initial state 3
int tem = 0;
tem := x;                  Swap program
x := y;
y := tem;
{(x, 10), (y, 20)}        final state 3

 …                         many others
```

(2) Use pre- and post-assertions (conditions) – Hoare triple:

$$\{P\}\ S\ \{R\}$$

If the assertion P is true before initiation of a program S, then the assertion R will be true on its completion.

Questions:

(1) Who is supposed to write P and R?

(2) What axioms and inference rules do we need for a formal proof of the validity of a specific Hoare triple?

Axioms for Program Proving

To prove the correctness of program S, we first need to establish necessary axioms to define the semantics of each program construct.

Program constructs:

(1)	Assignment	x := f
(2)	Sequence	S1; S2

(3)	Alternation	if B then S1
		if B then S1 else S2

(4)	Iteration	while B do S
(5)	Block	begin…end

Concepts and notations used in expressing axioms and rules:

(1) $Q[e/x]$ **where Q is an expression or formula, x is a variable, and e is an expression. Explanation: the result of replacing all free occurrences of x in Q by e.**

Example:

$$(x + y * z > y * u)[t+1/y] =$$

$$x + (t + 1) * z > (t + 1) * u$$

(2)

$$\frac{A, B}{C}$$

where A, B, and C are predicate formulae. Explanation: a rule of inference which states that if A and B have been proved, then C can be deduced.

This expression is equivalent to:

A and B => C

(3)

$$\frac{A, B \vdash C}{D}$$

where A, B, C, and D are predicate formulae. Explanation: a rule of inference which states that if A has been proved and C is deduced from B, then D can be deduced.

This expression is equivalent to:

A and (B ⊢ C) => D

Axiom 1: Axiom of assignment

$$\frac{}{\{P[f/x]\}\ x := f\ \{P\}}\quad \textbf{Ass-D0}$$

where

x is a variable identifier

f is an expression of a programming language without side effects, but possibly containing x.

P[f/x] is a predicate resulting from P by substituting f for all occurrences of x in P.

Examples:

(1)

{(x + y) * 5 > z}	{P}
x := x + y	S
{x * 5 > z}	{Q}

(2)

{add(x, z) + y * fact(z) > z}

y := add(x, z) + y * fact(z)

{y > z}

Class exercise

Derive the pre-assertion for each of the following two assignment statements based on their post-assertion:

(1)

```
x: = x + y;
{x > y}
```

(2)

```
y := x + y * (x + 5)
{x = 2 and y * x = 10}
```

Two rules of consequence

(1)

$$\frac{\{P\}\ S\ \{R\},\ R \Rightarrow Q}{\{P\}\ S\ \{Q\},} \qquad \textbf{Con-D1}$$

If the execution of program S ensures the truth of the assertion R under the pre-assertion P, then it also ensures the truth of every assertion logically implied by R.

(2)

$$\frac{\{P\}\ S\ \{R\},\ Q => P}{\{Q\}\ S\ \{R\},}$$

Con-D2

If P is known to be a pre-assertion for a program S to produce result R, then so is any other assertion that logically implies P.

Axiom 2: Axiom of composition

$$\frac{\{P\}\ S1\ \{R1\},\ \{R1\}\ S2\ \{R\}}{\{P\}\ S1;\ S2\ \{R\}}\quad \textbf{Com-D3}$$

If the proven result of the first part of a program is identical with the precondition under which the second part of the program produces its intended result, then the whole program will produce the intended result, provided that the precondition of the first part is satisfied.

Take the following two steps to use the rule of composition:

Step 1: Set the target Hoare triple for proof:

{P} S1; S2 {R}

Step 2: Try to find a predicate R1 such that the following two Hoare triples hold:

{P} S1 {R1} and {R1} S2 {R}

Challenge: how to find R1

Example:

The target Hoare triple for proof:

{x = 5 and y = 1} x: = x + 1; y := x + y {x = 6 and y = 7}

Proof:

Step 1: {x = 6 and x + y = 7} (**Ass-D0**)

y := x + y

{x = 6 and y = 7}

This is equivalent to

{x = 6 and y = 1}

y := x + y

{x = 6 and y = 7}

Step 2:

{x + 1 = 6 and y = 1}   (**Ass-D0**)

x: = x + 1

{x = 6 and y = 1}

This is equivalent to:

{ x = 5 and y = 1}

x: = x + 1

{x = 6 and y = 1}

Step 3:

```
{ x = 5 and y = 1}     (Com-D3)
 x: = x + 1;
 y := x + y
{x = 6 and y = 7}
```

Class exercise

Prove the validity of the following Hoare triple:

```
{x = 5 and y = 2}
x: = x * y;
y := x + y * (x + 5)
{x = 10 and y = 40}
```

Answer for the exercise

```
{x = 5 and y = 2}
{x * y = 10 and 10 + 10 * y + y * 5 = 40}
{x * y = 10 and x * y + y * (x * y + 5) = 40}
x: = x * y;
{x = 10 and x + y * (x + 5) = 40}
y := x + y * (x + 5)
{x = 10 and y = 40}
```

Axiom 3: Axioms of alternation (selection, choice)

(1) if B then S1

$$\frac{\{P \wedge B\}\ S1\ \{R\},\ P \wedge \neg B \Rightarrow R}{\{P\}\ \text{if } B \text{ then } S1\ \{R\}} \quad \textbf{Alt-D4}$$

If the execution of S1 ensures the truth of R under the condition that both P and B are true, and the conjunction of P and the negation of B implies R, then the alternation statement will ensure the truth of R under the pre-assertion P.

(2) if B then S1 else S2

$$\frac{\{P \wedge B\}\ S1\ \{R\},\ \{P \wedge \neg B\}\ S2\ \{R\}}{\{P\}\ \text{if } B \text{ then } S1 \text{ else } S2\ \{R\}}\quad \textbf{Alt-D5}$$

If the execution of S1 ensures the truth of R under the condition that both P and B are true, and the execution of S2 ensures the truth of R under the condition that the conjunction of P and the negation of B, then the alternation statement will ensure the truth of R under the pre-assertion P.

Important points on the rules of alternation:

(1) The same post-assertion R is used. The following case is incorrect:

$$\frac{\{P \wedge B\}\ S1\ \{R1\},\ \{P \wedge \neg B\}\ S2\ \{R2\}}{\{P\}\ \text{if } B \text{ then } S1 \text{ else } S2\ \{R\}}$$

where neither R1 nor R2 is equivalent to R.

(2) The following alternative rule can be used.

$$\frac{\{P \wedge B\}\ S1\ \{R1\},\ \{P \wedge \neg B\}\ S2\ \{R2\}}{\{P\}\ \text{if } B \text{ then } S1 \text{ else } S2\ \{R1 \vee R2\}} \quad \textbf{Alt-D6}$$

where neither R1 nor R2 is equivalent to R.

Proof:

H: $\{P \wedge B\}\ S1\ \{R1\}$

Infer: $\{P \wedge B\}\ S1\ \{R1 \vee R2\}$ (**Con-D1**, $R1 \Rightarrow R1 \vee R2$)

H: $\{P \wedge \neg B\}\ S2\ \{R2\}$

Infer: $\{P \wedge \neg B\}\ S2\ \{R1 \vee R2\}$ (**Con-D1**, $R2 \Rightarrow R1 \vee R2$)

Infer: $\{P\}$ if B then S1 else S2 $\{R1 \vee R2\}$ (**Alt-D5**)

Example:

The target Hoare triple for proof:

```
{x <> y}
if x > y
then x := y – 1
else x := y;
{x = y ∨ x < y}
```

Proof:

Step 1: Prove

$$\{x <> y \wedge x > y\}\ x := y - 1\ \{x = y \vee x < y\}$$

Step 2: Prove

$$\{x <> y \wedge x <= y\}\ x := y\ \{x = y \vee x < y\}$$

Step 3: Prove the target Hoare triple.

Step 1: Prove

$$\{x <> y \wedge x > y\}\ x := y - 1\ \{x = y \vee x < y\}$$

Proof:

$\{y - 1 = y \vee y - 1 < y\}$ **(Ass-D0)**

$x := y - 1$

$\{x = y \vee x < y\}$

This is equivalent to

$\{y < y + 1\}$

$x := y - 1$

$\{x = y \vee x < y\}$

Now the main task is to prove:

$x <> y \wedge x > y => y < y + 1$ (L1)

This can be easily proved, because

$y < y + 1 <=> true$

Therefore, we have

$\{x <> y \wedge x > y\}$ (L1, **Con-D2**)

$x := y – 1$

$\{x = y \vee x < y\}$

Step 2: Prove

$\{x <> y \land x <= y\}\ x := y\ \{x = y \lor x < y\}$

Proof:

$\{y = y \lor y < y\}$ **(Ass-D0)**

$x := y$

$\{x = y \lor x < y\}$

This is equivalent to

$\{true\}$

$x := y$

$\{x = y \lor x < y\}$

Now the main task is to prove:

$x <> y \wedge x <= y => true$ (L2)

This is true according to the rule of =>.

Therefore, we have

$\{x <> y \wedge x <= y\}$ (L2, **Con-D2**)

$x := y$

$\{x = y \vee x < y\}$

Step 3: Prove the target Hoare triple:

```
{x <> y}
if x > y
then x := y – 1
else x := y;
{x = y ∨ x < y}
```

Proof:

```
{x <> y}                    (step 1, step 2, Alt-D5)
if x > y
then x := y – 1
else x := y;
{x = y ∨ x < y}
```

Class exercise

Prove the validity of the following Hoare triple:

```
$\{x > 0\}$
 if $x > y + 10$
 then $x := y + 20$
 else $y := x + 10$
 $\{x = y + 20 \lor x < y\}$
```

Answer for the exercise

Step 1:

$\{x > 0 \land x > y + 10\}$

$x := y + 20$

$\{x = y + 20 \lor x < y\}$

Step 2:

$\{x > 0 \land x <= y + 10\}$

$y := x + 10$

$\{x = y + 20 \lor x < y\}$

Step 3:

```
{x > 0}
 if x > y + 10
 then x := y + 20
 else y := x + 10
 {x = y + 20 ∨ x < y}
```

Proof:

Step 1:

$\{y + 20 = y + 20 \lor y + 20 < y\}$ Ass-D0

$x := y + 20$

$\{x = y + 20 \lor x < y\}$

$x > 0 \land x > y + 10 \Rightarrow$ (L1)

$y + 20 = y + 20 \lor y + 20 < y$

$\{x > 0 \wedge x > y + 10\}$ (L1, Con-D2)

$x := y + 20$

$\{x = y + 20 \vee x < y\}$

Step 2:

$\{x = x + 30 \vee x < x + 10\}$ (Ass-D0)

$y := x + 10$

$\{x = y + 20 \vee x < y\}$

$x > 0 \wedge x <= y + 10 =>$ (L2)

$x = x + 30 \vee x < x + 10$

$\{x > 0 \wedge x <= y + 10\}$ (L2, Con-D2)

```
y := x + 10
```

$\{x = y + 20 \vee x < y\}$

Step 3:

$\{x > 0\}$ (Step 1, Step 2, Alt-D5)

```
 if x > y + 10
 then x := y + 20
 else y := x + 10
```

$\{x = y + 20 \vee x < y\}$

Axiom 4: Axiom of iteration (while B do S)

$$\frac{\{P \land B\}\ S\ \{P\}}{\{P\}\ \text{while } B \text{ do } S\ \{P \land \neg B\}} \qquad \text{Ite-D7}$$

If P holds and B is true before the execution of statement S and its execution sustains the truth of P, then the execution of the iteration statement under the precondition P will sustain the truth of P and makes B false.

P is called invariant of the loop.

Important points on the rule of iteration:

(1) Deriving the invariant P is a process of abstracting the while-loop statement into a predicate expression that expresses its meaning (or function).

(2) For the same while-loop, there may be more than one invariant. For example, true can be an invariant for any while-loop.

(3) Deriving appropriate invariant P is the most challenging task in program proving, and there is no systematic method for this.

Example:

A program of finding the quotient q and remainder r obtained on dividing x by y, where both x and y are non-zero natural numbers.

```
 r := x;
q := 0;
while y <= r do
 begin
 r := r - y;
 q := 1 + q
 end
```

S

An important property of this program is when it terminates, we can recover the numerator x by adding to the remainder r the product of the divisor y and the quotient q (i.e., $x = r + y \times q$). Furthermore, the remainder is less than the divisor. These properties can be expressed formally:

$$\{true\}\ S\ \{x = r + y \times q \wedge \neg y <= r\}$$

where S represents the program displayed above.

The proof can be conducted in two steps:

Sept 1: $\{true\}\ r := x;\ q := 0;\ \{x = r + y \times q\}$

Step 2:

```
{x = r + y × q}
while y <= r do
  begin
  r := r - y;
  q := 1 + q
  end
{x = r + y × q ∧ ¬y <=r}
```

A formal proof of the correctness of program S:

1 true => x = x + y × 0 Lemma 1

2 {x = x + y × 0} r := x {x = r + y × 0 } Ass-D0

3 {x = r + y × 0} q := 0 {x = r + y × q} Ass-D0

4 {true} r := x {x = r + y × 0 } Con-D2(1,2)

5 {true} r := x; q := 0 {x = r + y × q} Com-D3(4,3)

6 x = r + y × q ∧ y <= r =>
x = (r – y) + y × (1 + q) Lemma 2

7 {x = (r – y) + y × (1 + q)} r := r – y
{x = r + y × (1 + q)} Ass-D0

8 {x = r + y × (1 + q)} q := 1 + q
{x = r + y × q)} Ass-D0

9 {x = (r – y) + y × (1 + q)} r := r – y; q := 1 + q

{x = r + y × q)} Com-D3(7, 8)

10 {x = r + y × q) ∧ y <= r} r := r – y; q := 1 + q

{x = r + y × q)} Con-D2(6,9)

11 {x = r + y × q)}

```
    while y <= r do
     begin
      r := r – y; q := 1 + q
     end
```

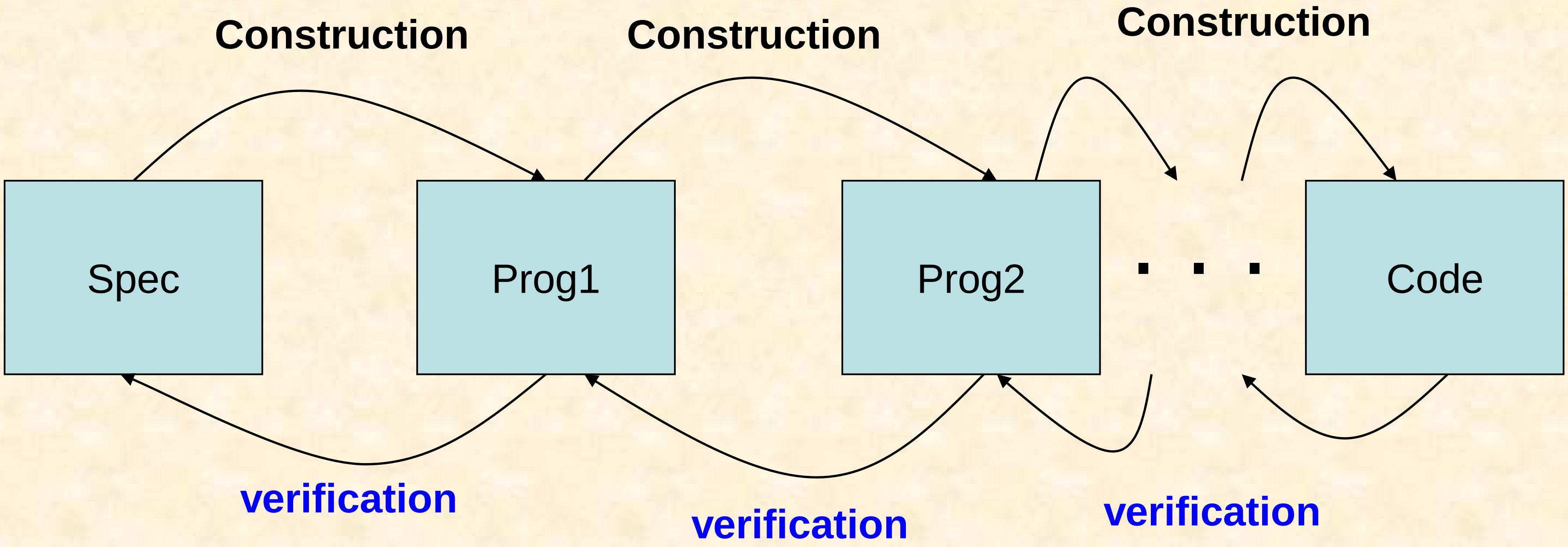
{x = r + y × q ∧ ¬y <= r} **Ite-D7(10)**

```
12 {true} r := x; q := 0;
          while y <= r do
          begin
            r := r – y; q := 1 + q
          end
   {x = r + y × q ∧¬y <= r}          Com-D3(5, 11)
```

Observation:

This program should have used x > 0 and y > 0 in the pre-assertion, but since it only ensures the termination of the while loop, its absence in the pre-assertion does not affect the proof of the partial correctness.

Software development based on formal verification


Construction

Construction

Construction

Spec

Prog1

Prog2

. . .

Code

verification

verification

verification

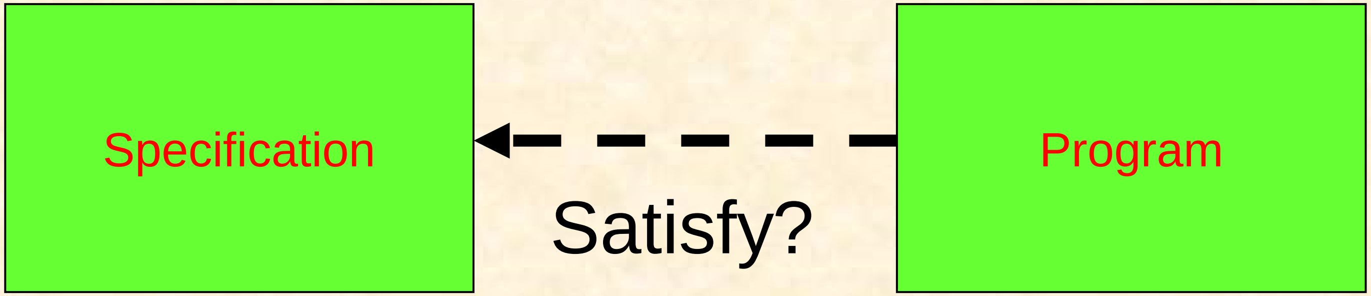
IV. Specification-Based Inspection and Testing

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



Specification

Satisfy?

Program

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.

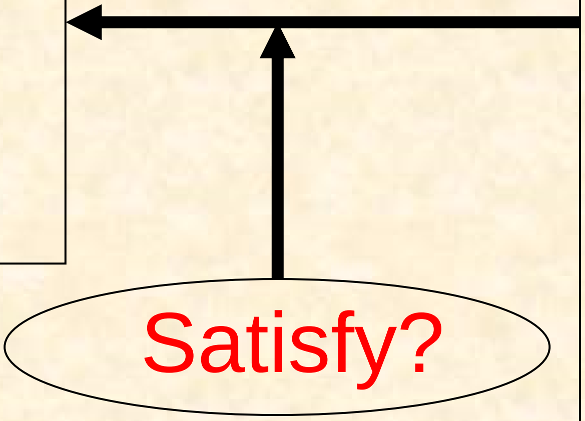
Scenario-based inspection: a strategy for ``divide and conquer”

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

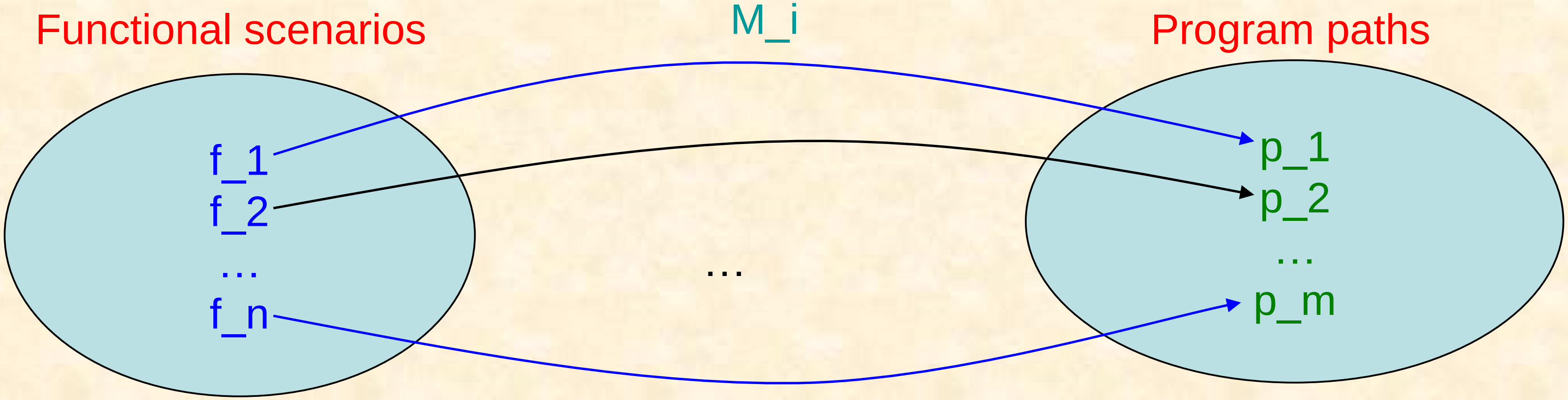
```
int A(int x) {
If (x > 0) {
  if (x > 10)  y = x + 1;
  else  y = x – 1;
   return y; }
 else System.out.println(“the
          pre is violated”)  }
```



Functional scenario:

Apre ∧Gi ∧Di

(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.

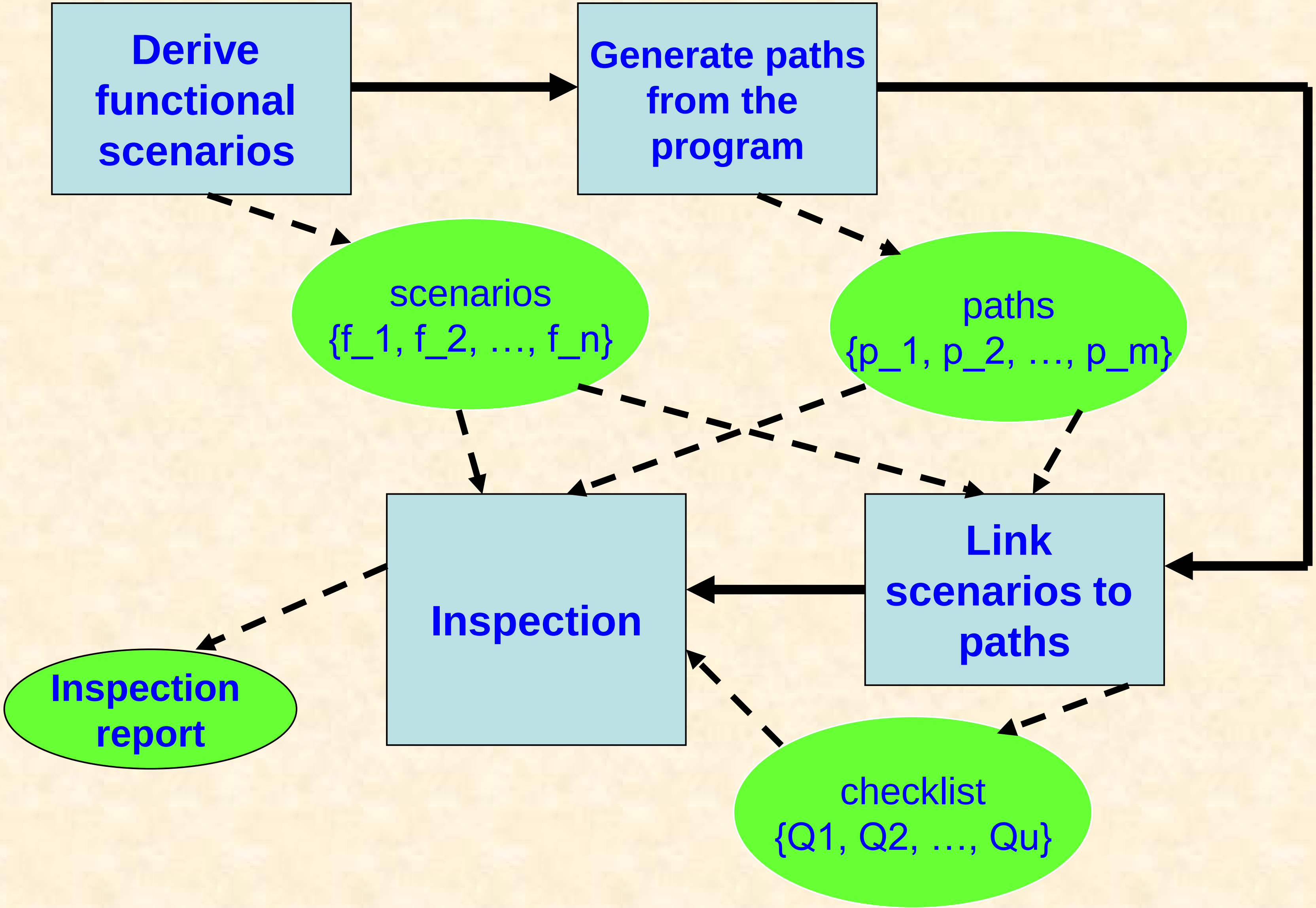
Formally, the principle is described as follows:

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

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

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

A Process for Scenario-Based Inspection


Derive functional scenarios
Generate paths from the program
scenarios {f_1, f_2, …, f_n}
paths {p_1, p_2, …, p_m}
Inspection
Link scenarios to paths
Inspection report
checklist {Q1, Q2, …, Qu}

(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∈{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
```

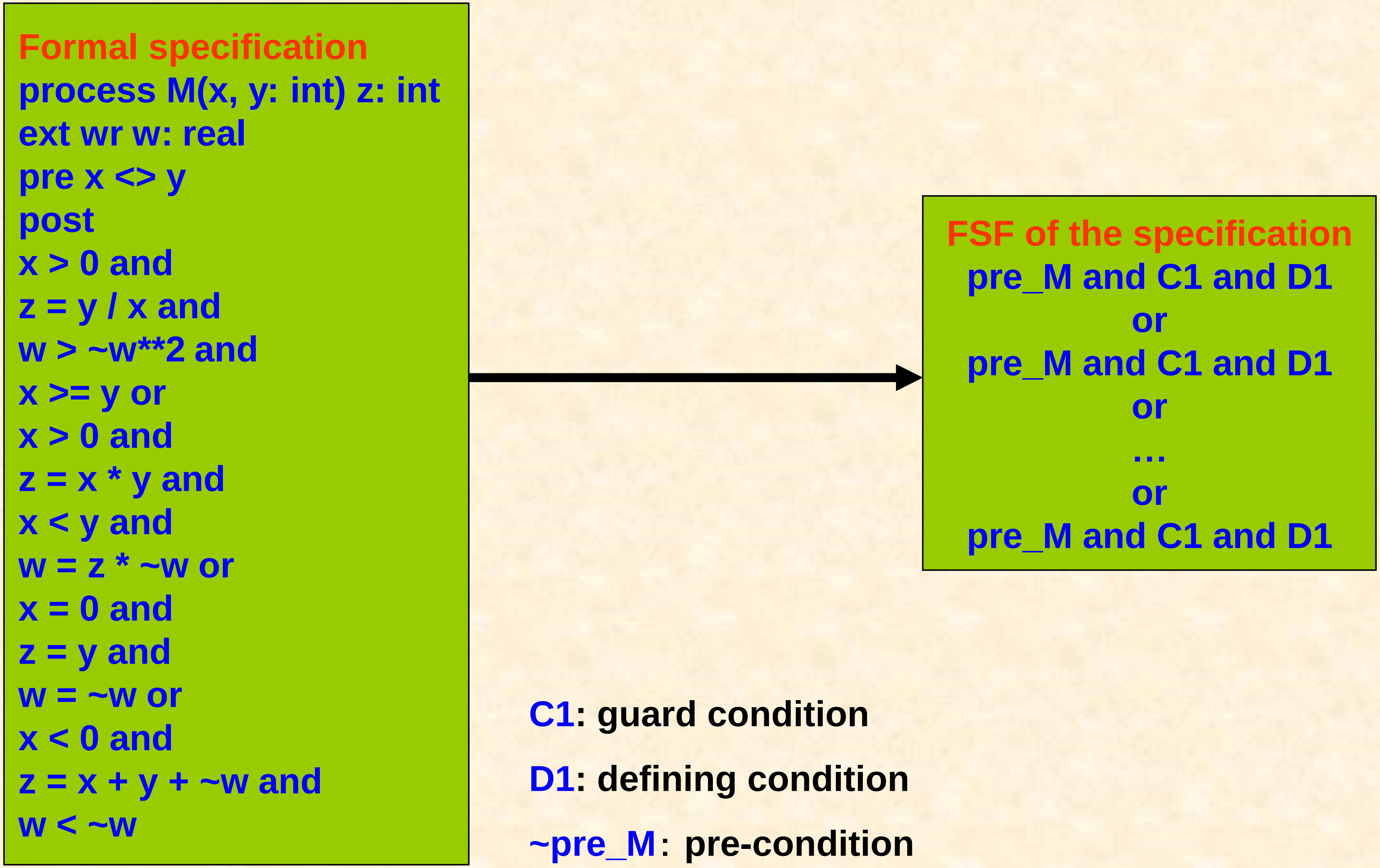
No.2 Transform the disjunctive normal form into the functional scenario form:

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 > ~w2 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

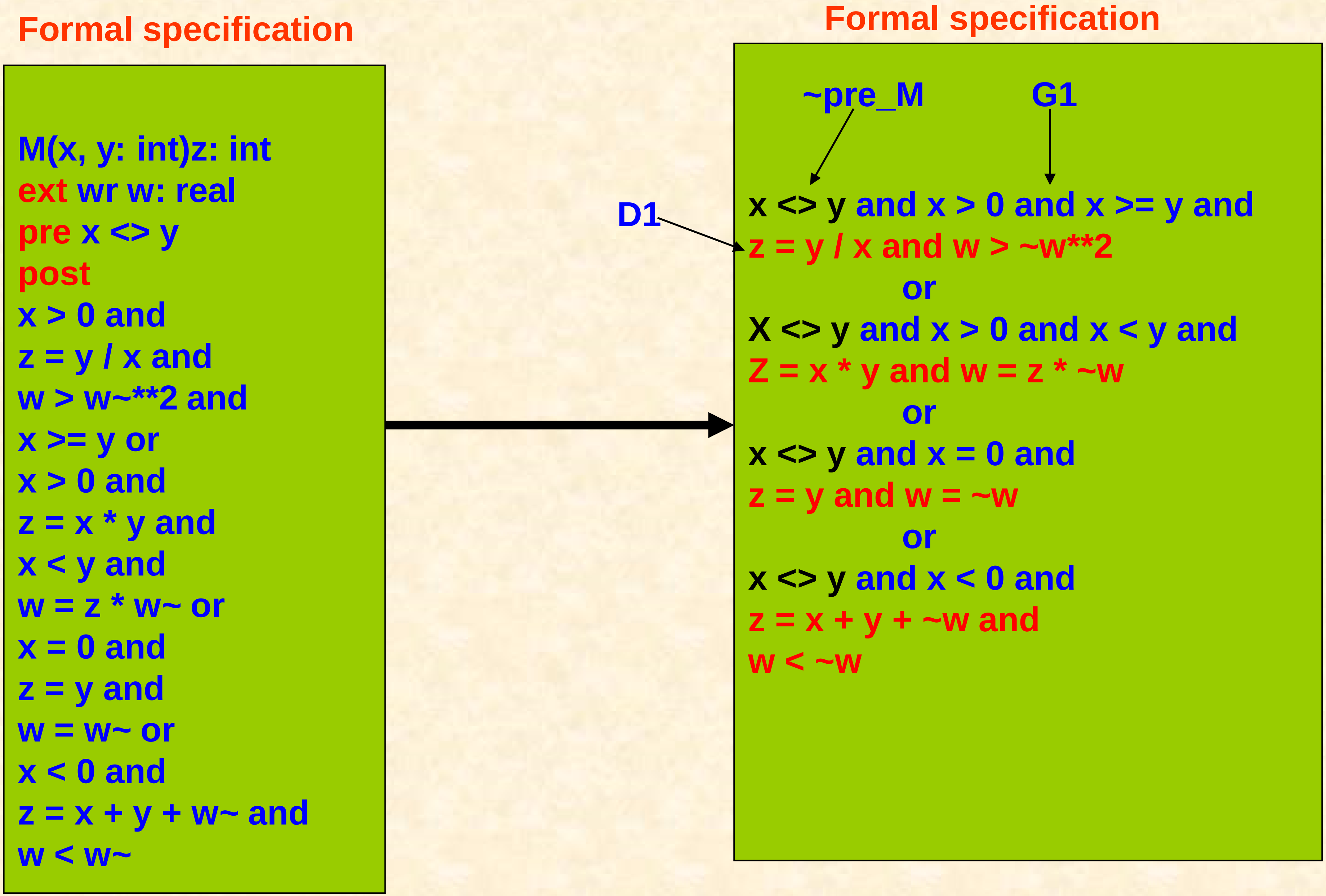
pre_M and C1 and D1
or
pre_M and C1 and D1
or
…
or
pre_M and C1 and D1

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

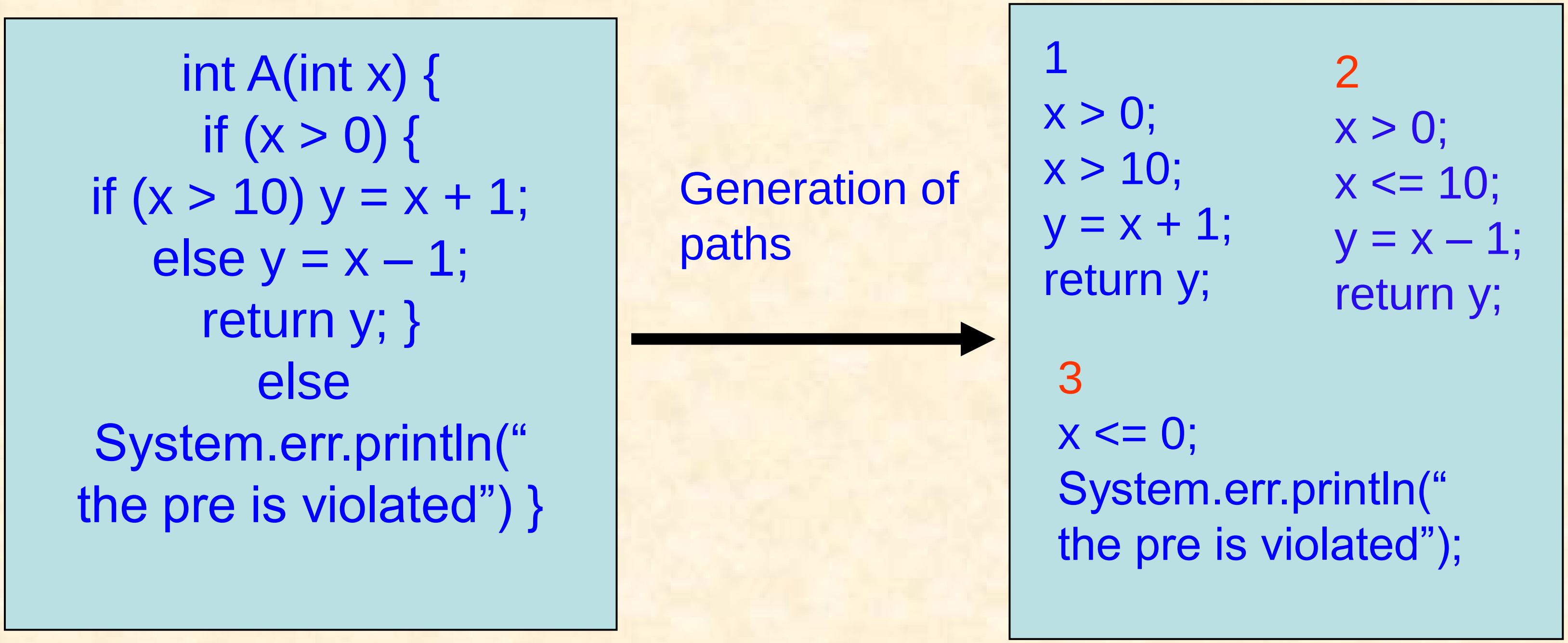
G1

D1

```
x <> y and x > 0 and x >= y and
z = y / x and w > ~w**2
                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

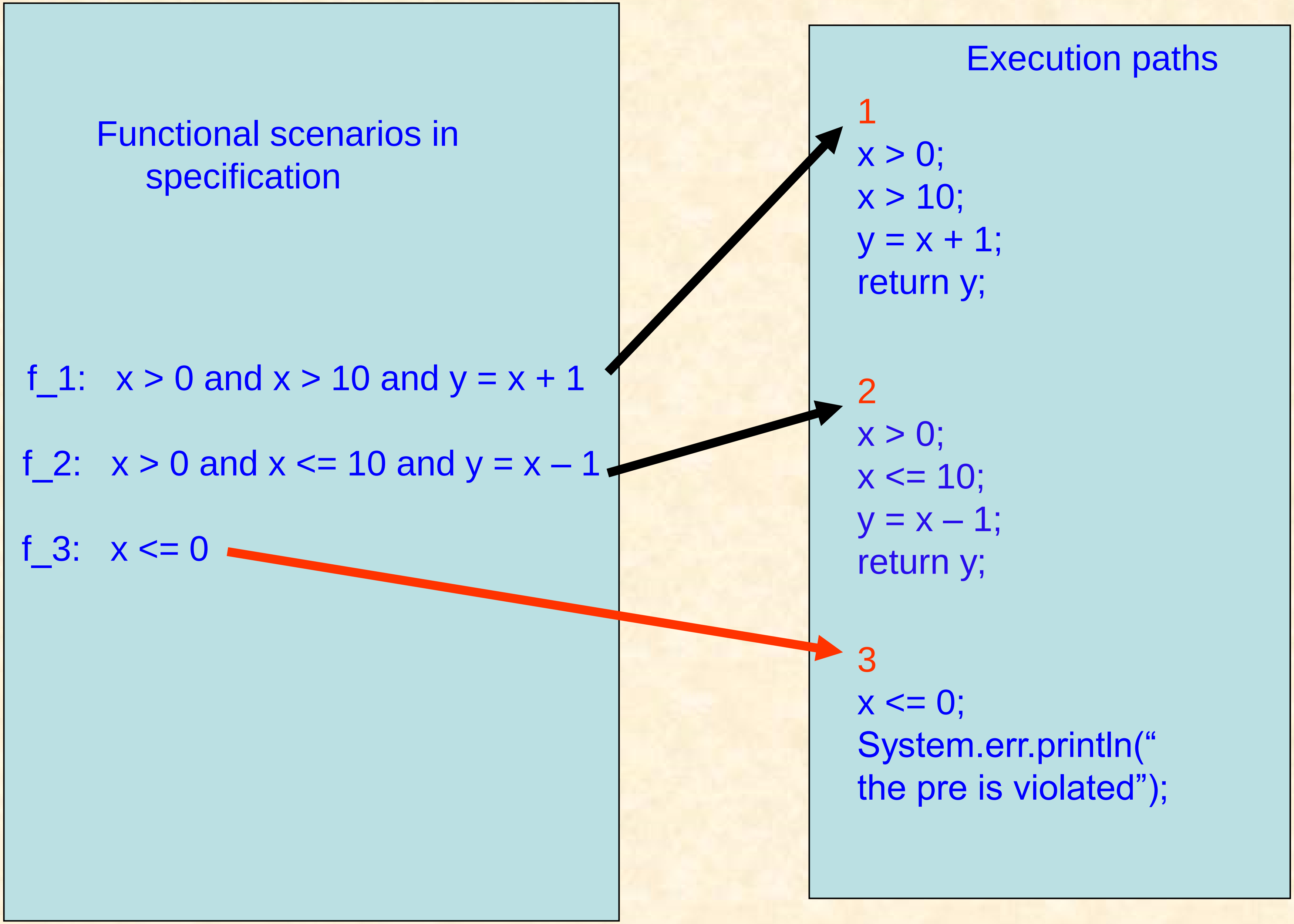
Two strategies:

- 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.**

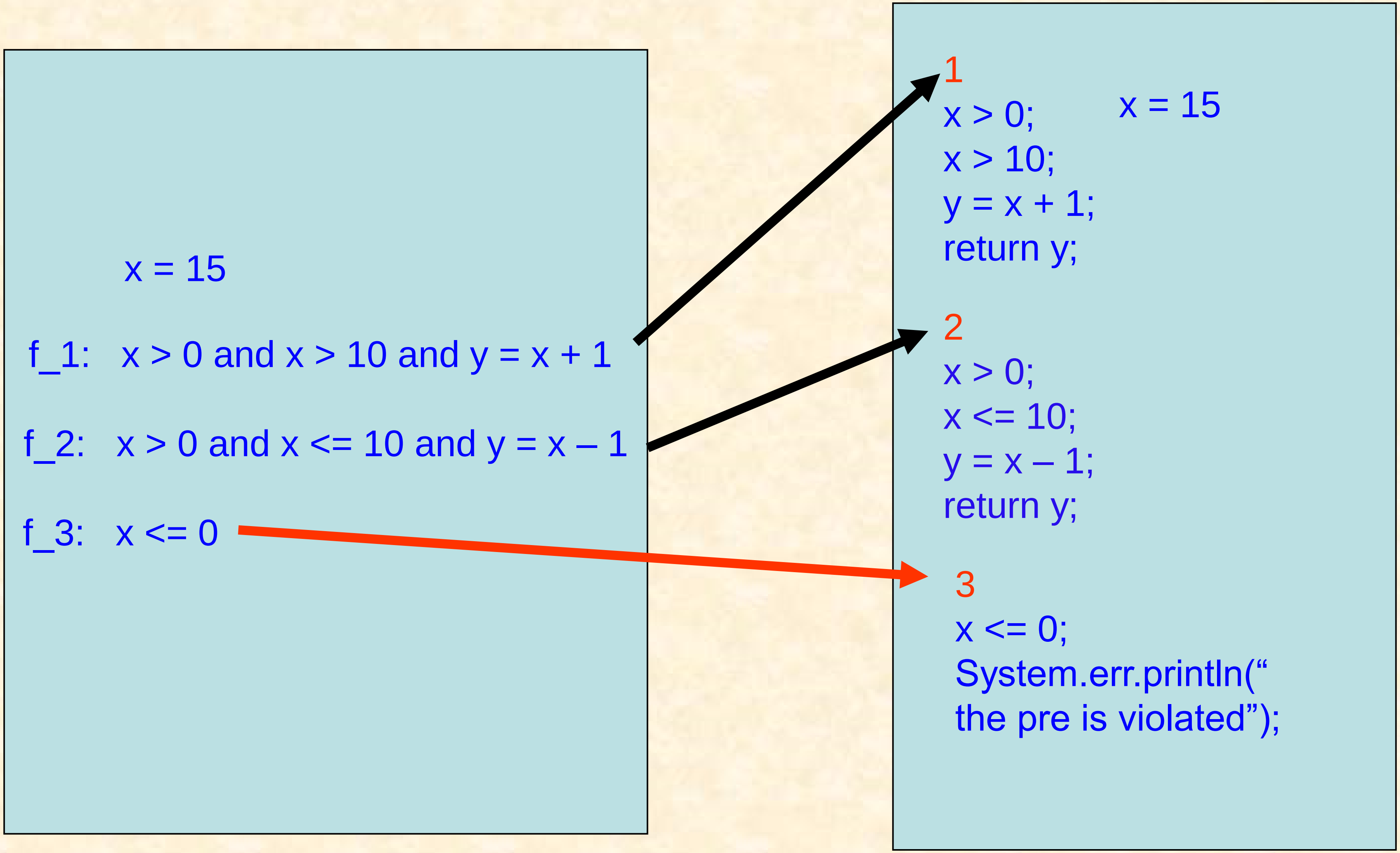
  Example questions on the checklist:

  (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 > 0;          x = 15
x > 10;
y = x + 1;
return y;
```

2

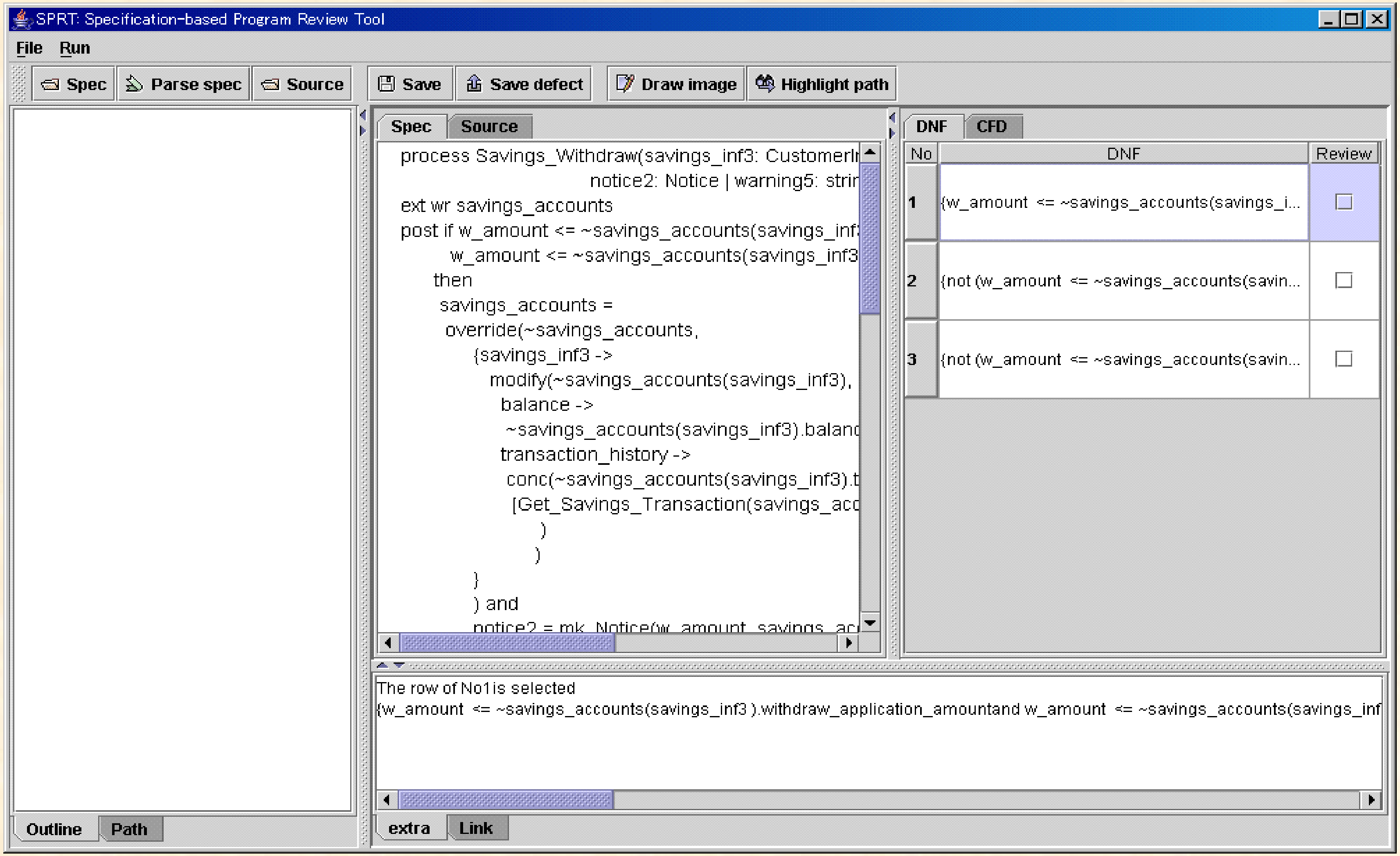
```
x > 0;
x <= 10;
y = x – 1;
return y;
```

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.

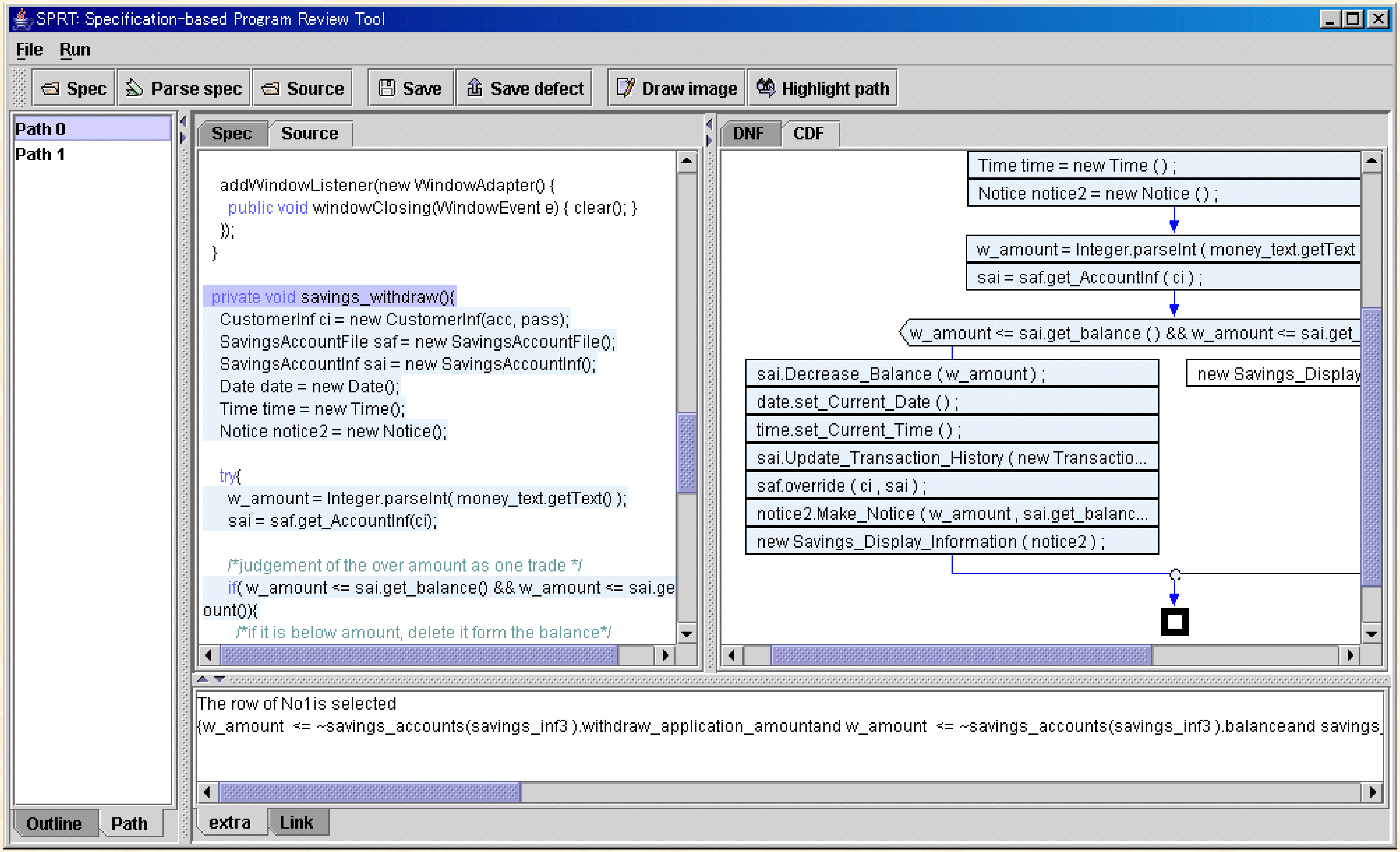


```
process Savings_Withdraw(savings_inf3: CustomerIr
                                   notice2: Notice | warning5: strir
ext wr savings_accounts
post if w_amount <= ~savings_accounts(savings_inf:
         w_amount <= ~savings_accounts(savings_inf3
      then
       savings_accounts =
        override(~savings_accounts,
             {savings_inf3 ->
                modify(~savings_accounts(savings_inf3),
                  balance ->
                   ~savings_accounts(savings_inf3).balanc
                  transaction_history ->
                   conc(~savings_accounts(savings_inf3).t
                    [Get_Savings_Transaction(savings_acc
                         )
                        )
             }
             ) and
             notice2 = mk_Notice(w_amount, savings_ac
```

No	DNF	Review
1	{w_amount <= ~savings_accounts(savings_i...	□
2	{not (w_amount <= ~savings_accounts(savin...	□
3	{not (w_amount <= ~savings_accounts(savin...	□

The row of No1 is selected
{w_amount <= ~savings_accounts(savings_inf3).withdraw_application_amountand w_amount <= ~savings_accounts(savings_inf

Automatic derivation of program paths from a Java method.

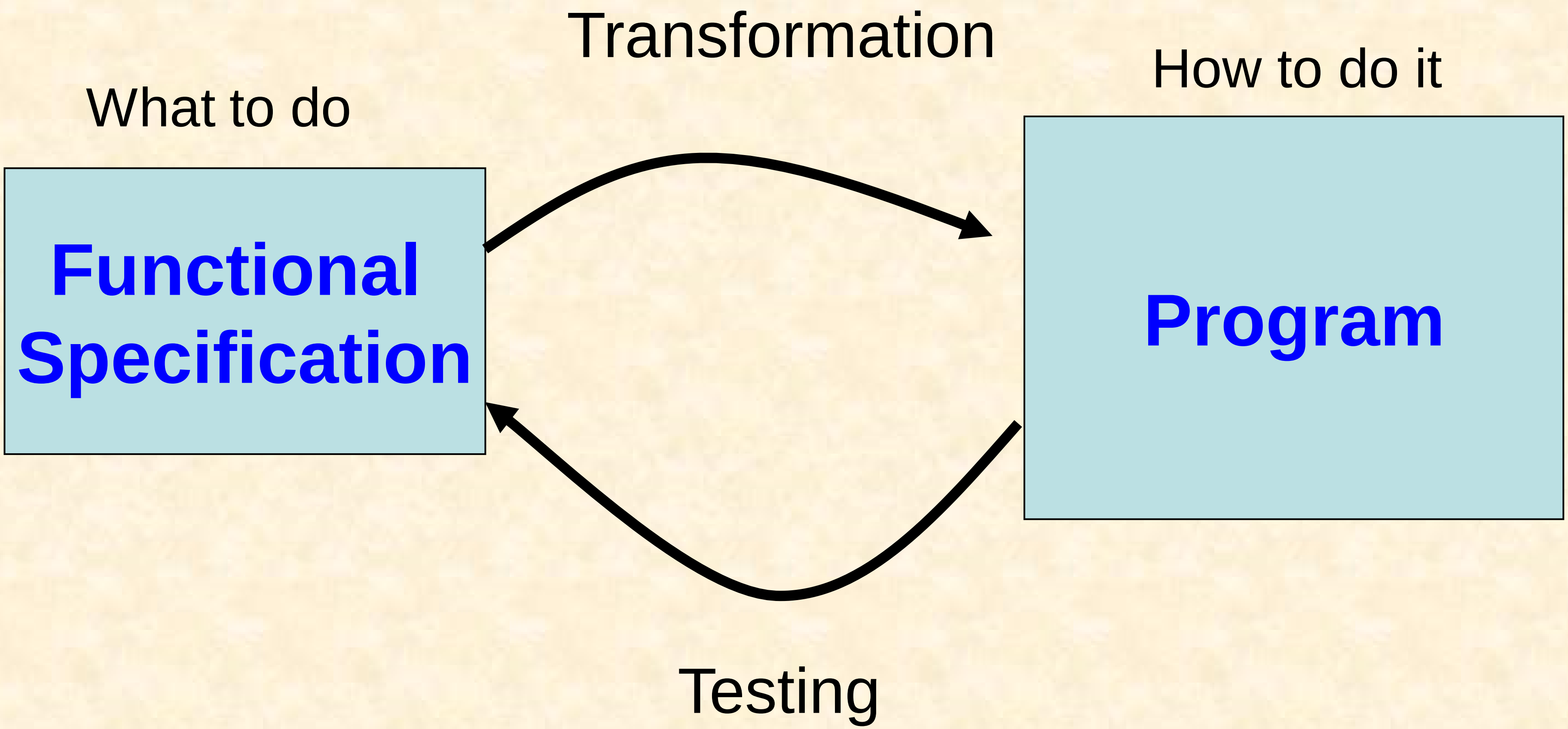


```
 addWindowListener(new WindowAdapter() {
   public void windowClosing(WindowEvent e) { clear(); }
 });
}

private void savings_withdraw(){
  CustomerInf ci = new CustomerInf(acc, pass);
  SavingsAccountFile saf = new SavingsAccountFile();
  SavingsAccountInf sai = new SavingsAccountInf();
  Date date = new Date();
  Time time = new Time();
  Notice notice2 = new Notice();

  try{
   w_amount = Integer.parseInt( money_text.getText() );
   sai = saf.get_AccountInf(ci);

   /*judgement of the over amount as one trade */
   if( w_amount <= sai.get_balance() && w_amount <= sai.ge
ount()){
     /*if it is below amount, delete it form the balance*/
```

```
Time time = new Time ( ) ;
Notice notice2 = new Notice ( ) ;
w_amount = Integer.parseInt ( money_text.getText
sai = saf.get_AccountInf ( ci ) ;
w_amount <= sai.get_balance ( ) && w_amount <= sai.get_
sai.Decrease_Balance ( w_amount ) ;
date.set_Current_Date ( ) ;
time.set_Current_Time ( ) ;
sai.Update_Transaction_History ( new Transactio...
saf.override ( ci , sai ) ;
notice2.Make_Notice ( w_amount , sai.get_balanc...
new Savings_Display_Information ( notice2 ) ;
new Savings_Display
```

```
The row of No1 is selected
(w_amount  <= ~savings_accounts(savings_inf3 ).withdraw_application_amountand w_amount  <= ~savings_accounts(savings_inf3 ).balanceand savings_
```

IV.2 Specification-Based Program Testing



Transformation

What to do

How to do it

Functional Specification

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

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

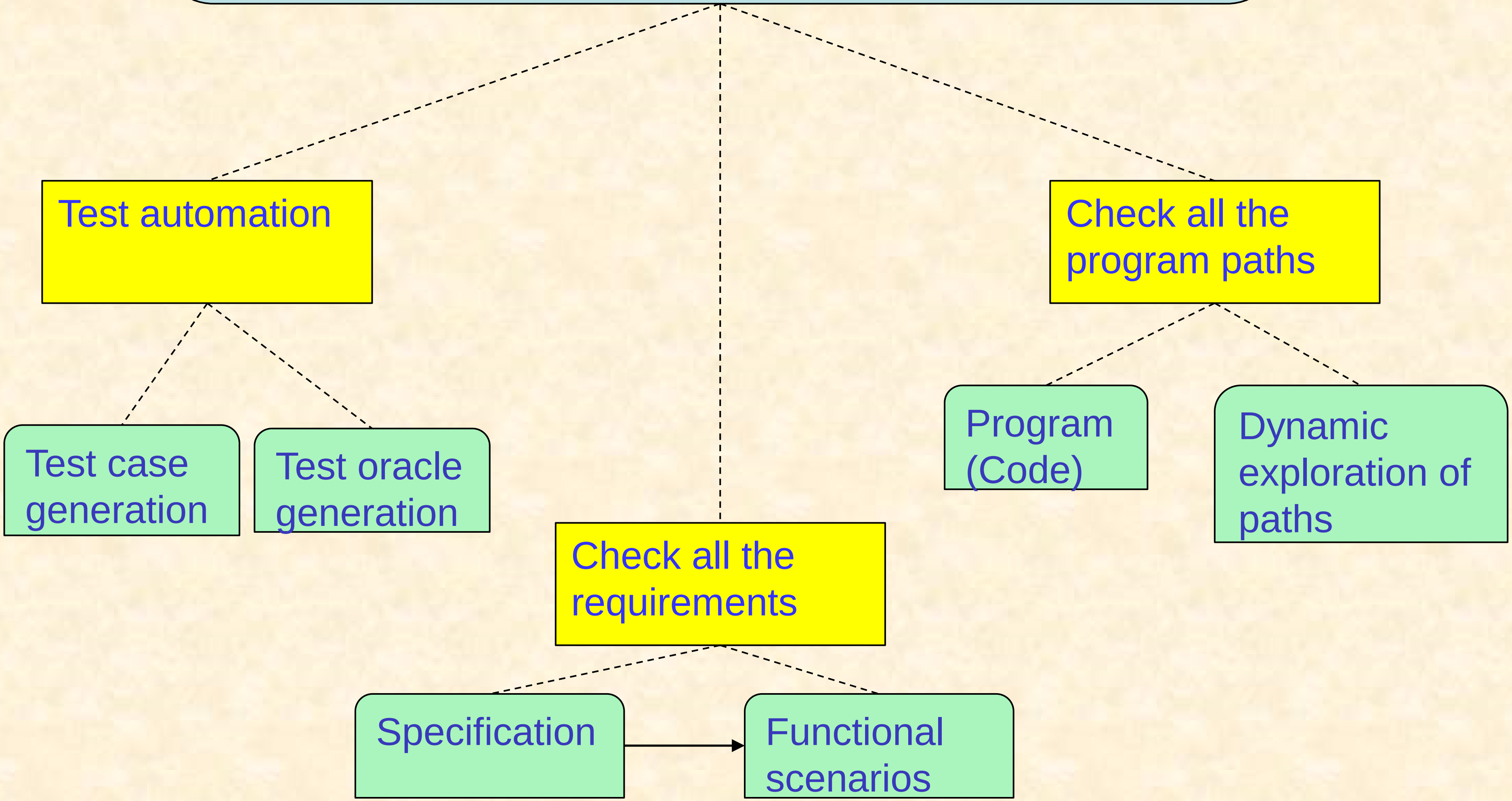
The features of specification-based testing:

(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

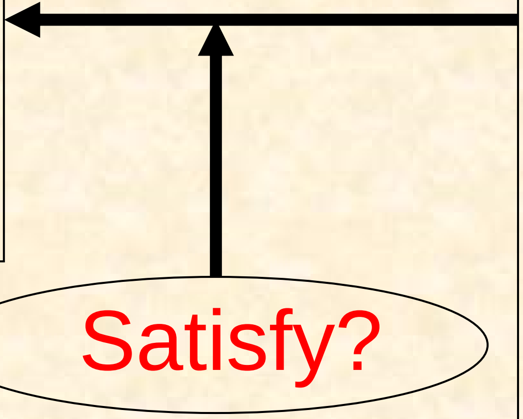
Functional scenario-based testing

Specification (in SOFL)

```
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")  }
```

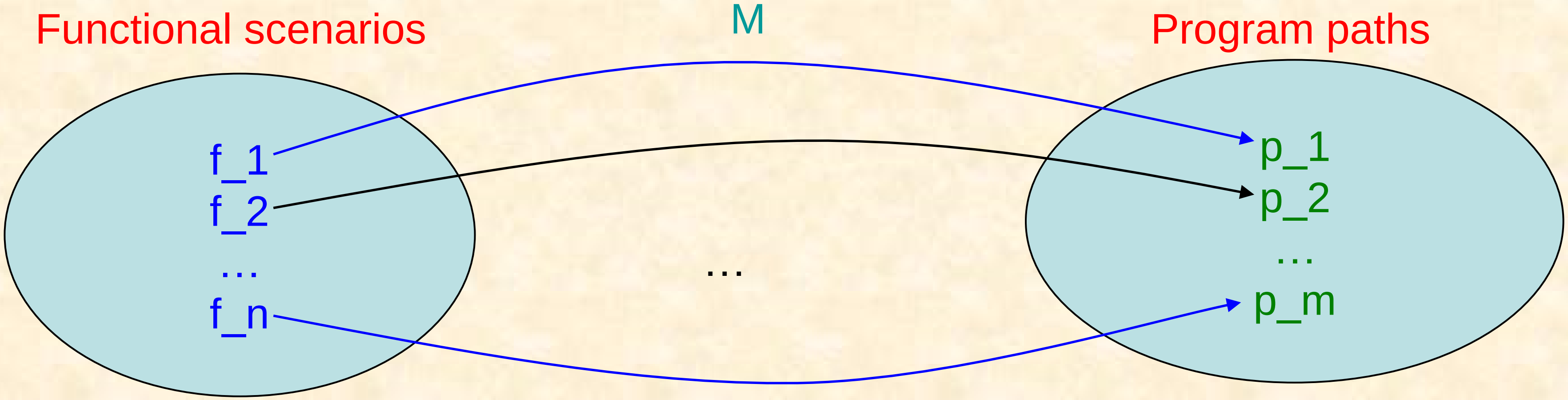


Satisfy?

Functional scenario:

Apre ∧Gi ∧Di

(i=1,…,n)



Functional scenarios

Program paths

Specification:

process A(x: int) y: int

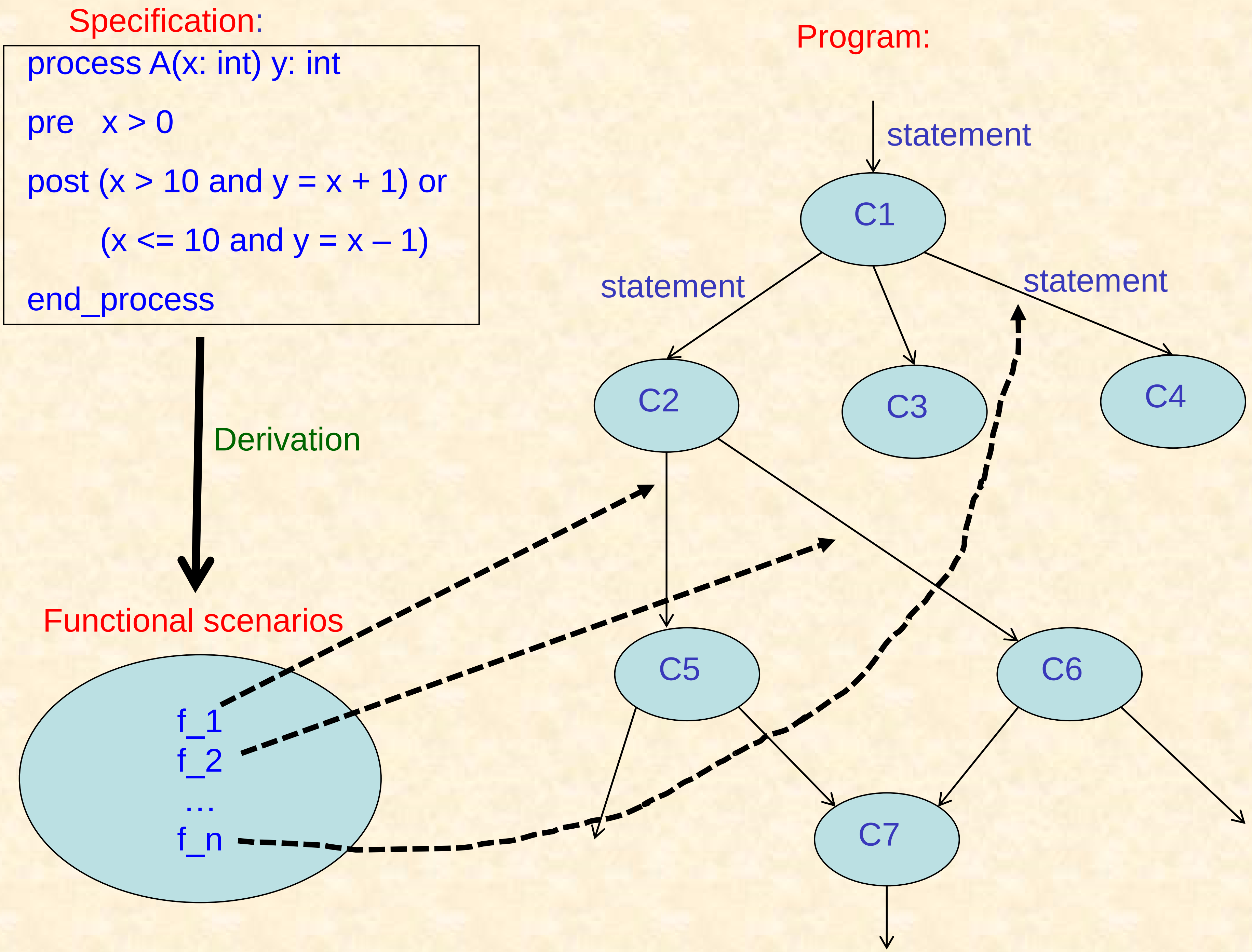
pre x > 0

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

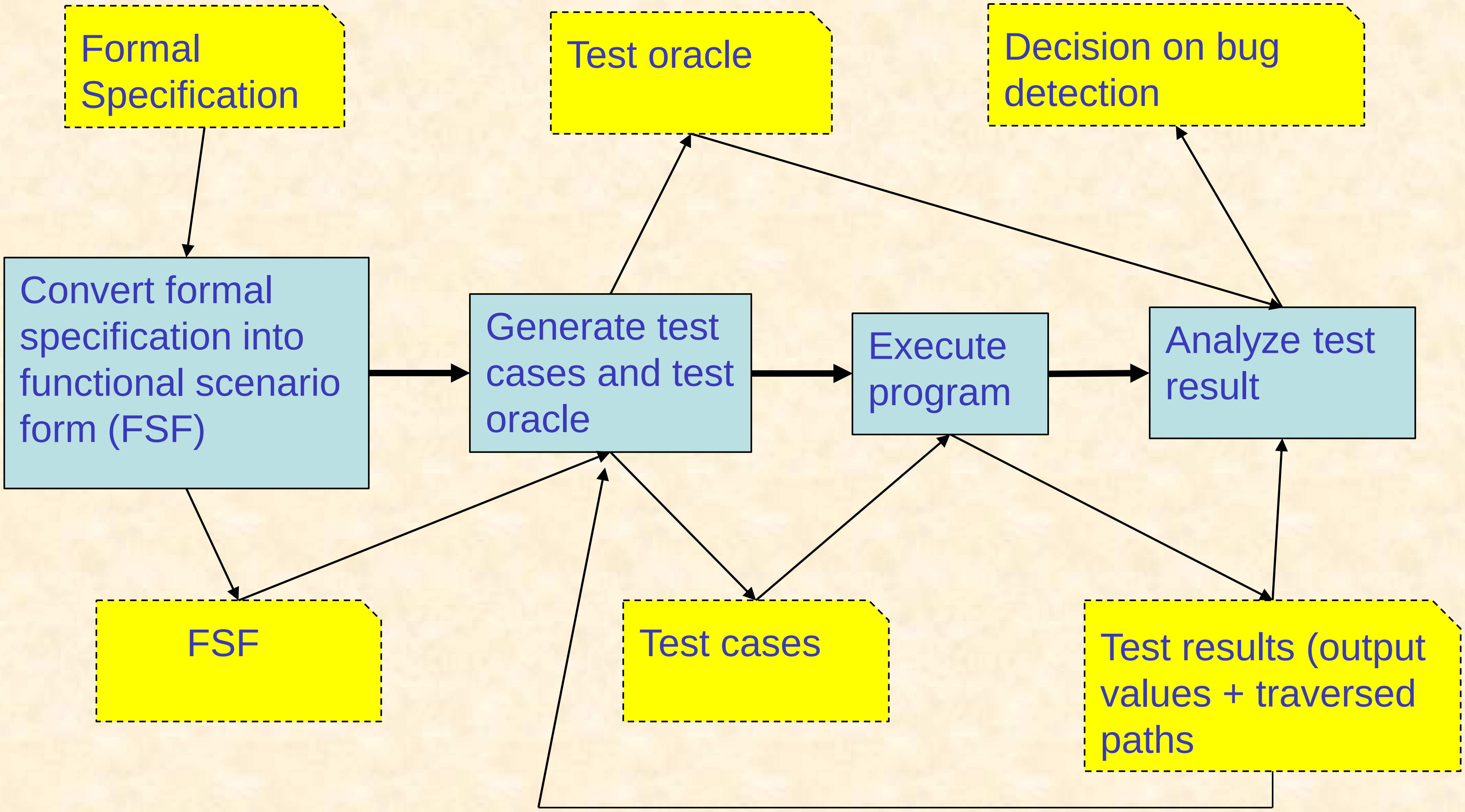
(x <= 10 and y = x – 1)

end_process

Program:

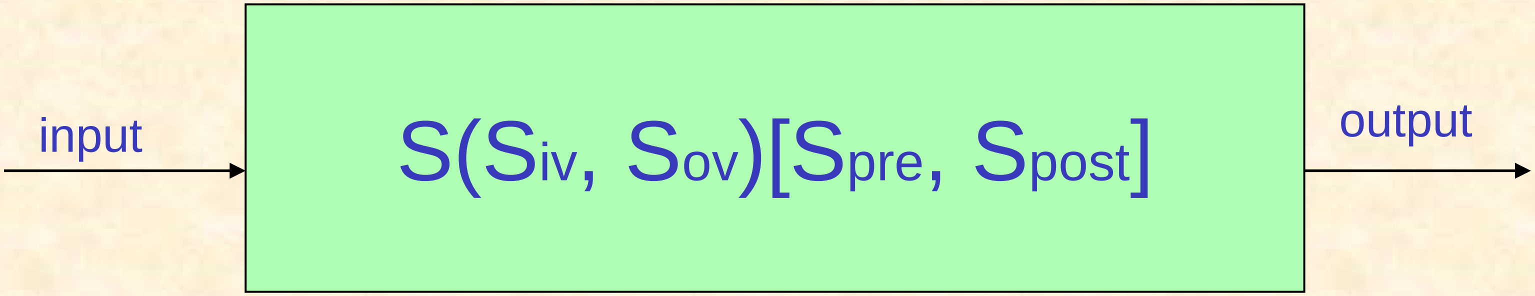


The framework for functional scenario-based testing (V-Method)


Formal Specification
Test oracle
Decision on bug detection
Convert formal specification into functional scenario form (FSF)
Generate test cases and test oracle
Execute program
Analyze test result
FSF
Test cases
Test results (output values + traversed paths

Functional scenario form (FSF) and functional scenario

Let



$$S(S_{iv}, S_{ov})[S_{pre}, S_{post}]$$

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:

$$S_{pre} \ => G_1 \vee G_2 \vee \cdots \vee G_n$$

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,

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

(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:

$\neg S_{pre}(tc)$

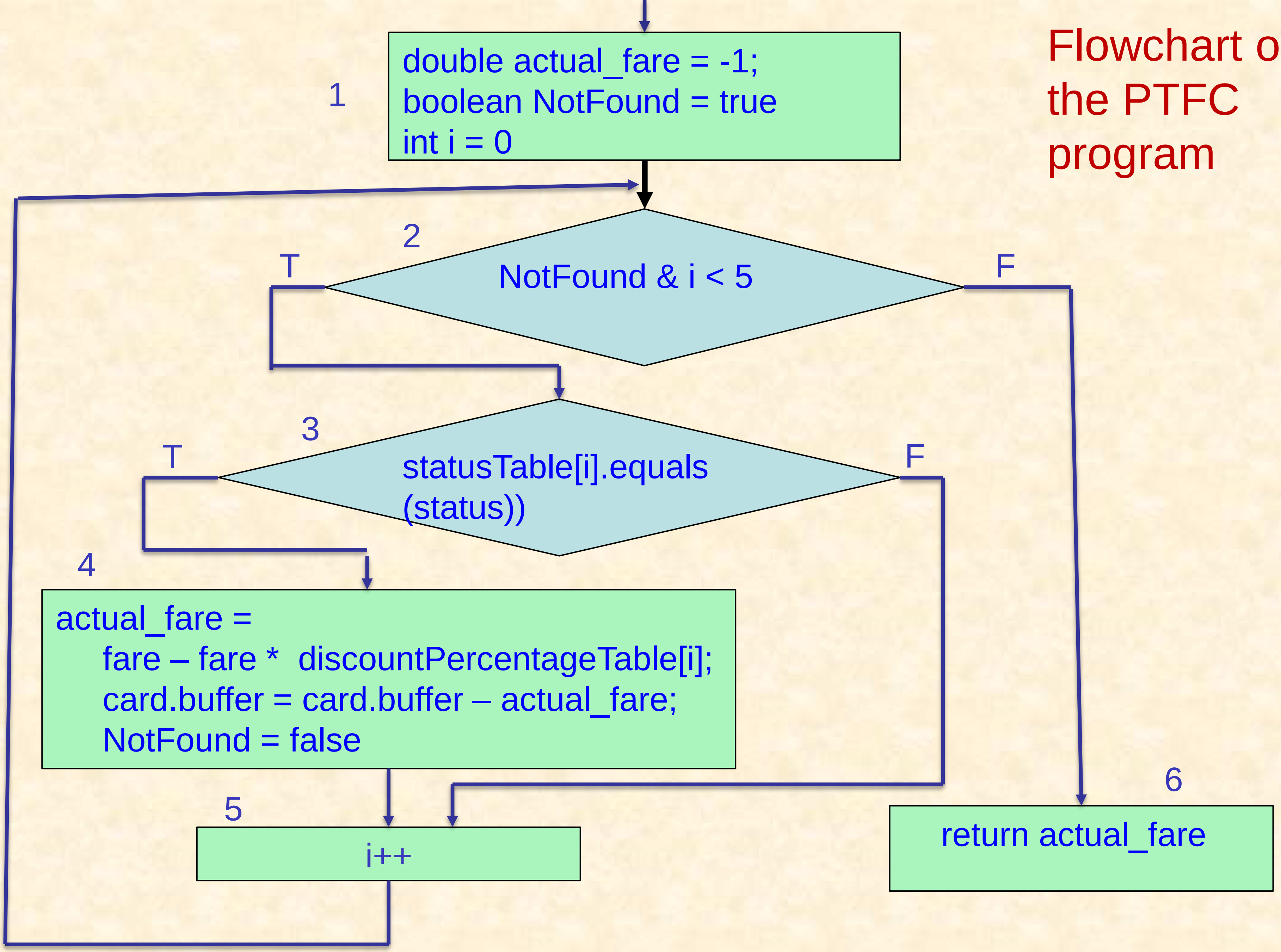
(4) For every path p in the representative path set R_{PP}, there must exist a test case tc in T such that the following condition is satisfied:

$traversed(p, tc)$

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



1

```
double actual_fare = -1;
boolean NotFound = true
int i = 0
```

2

NotFound & i < 5

3

statusTable[i].equals (status))

4

```
actual_fare =
    fare – fare *  discountPercentageTable[i];
    card.buffer = card.buffer – actual_fare;
    NotFound = false
```

5

i++

6

return actual_fare

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).

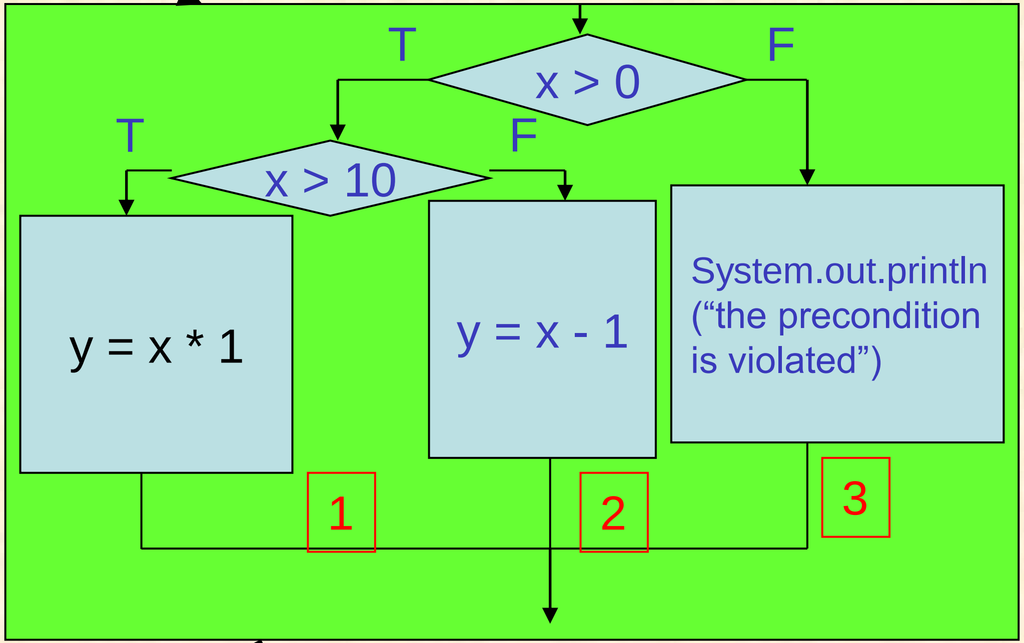
Test case generation

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 $x_1, x_2, \ldots, x_q$.

(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: $(E_1 \ominus E_2)(x_1, x_2, \dots, x_q)$

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

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:

$Q_1 \land Q_2 \land \cdots \land Q_n$

(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:

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

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 {$R_1$, $R_2$, ... , $R_u$} for {$Q_1$, $Q_2$, ⋯, $Q_n$}.
No.2. $t_0$ := {}; i := 1; flag := 0; /*initializing
          $t_0$ representing the generated test data*/
No.3. while ( i ≤ u & flag ≤ 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 ($t_i$ == {}){
  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.

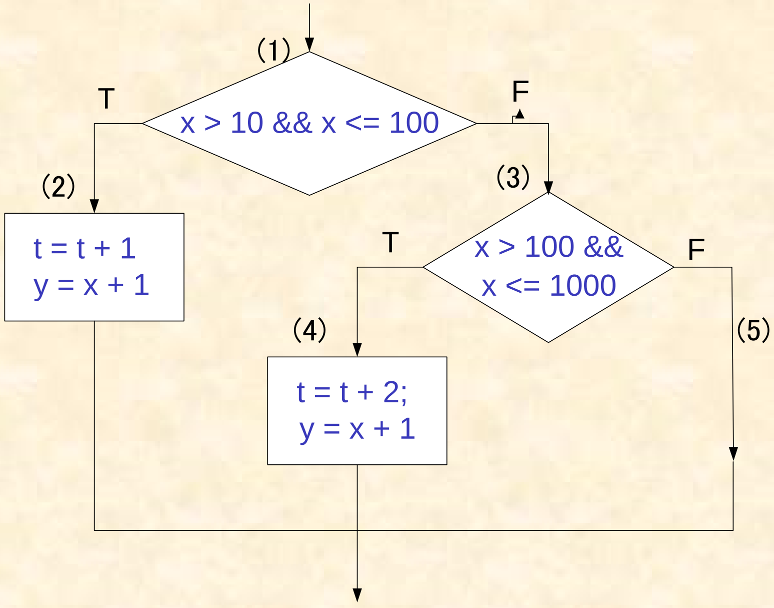
Example: the functional scenario:

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

x is input

y is output

int t; //global variable declared before



Three paths: [(1), (2)],
[(1), (3), (4)]
[(1), (3), (5)]

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

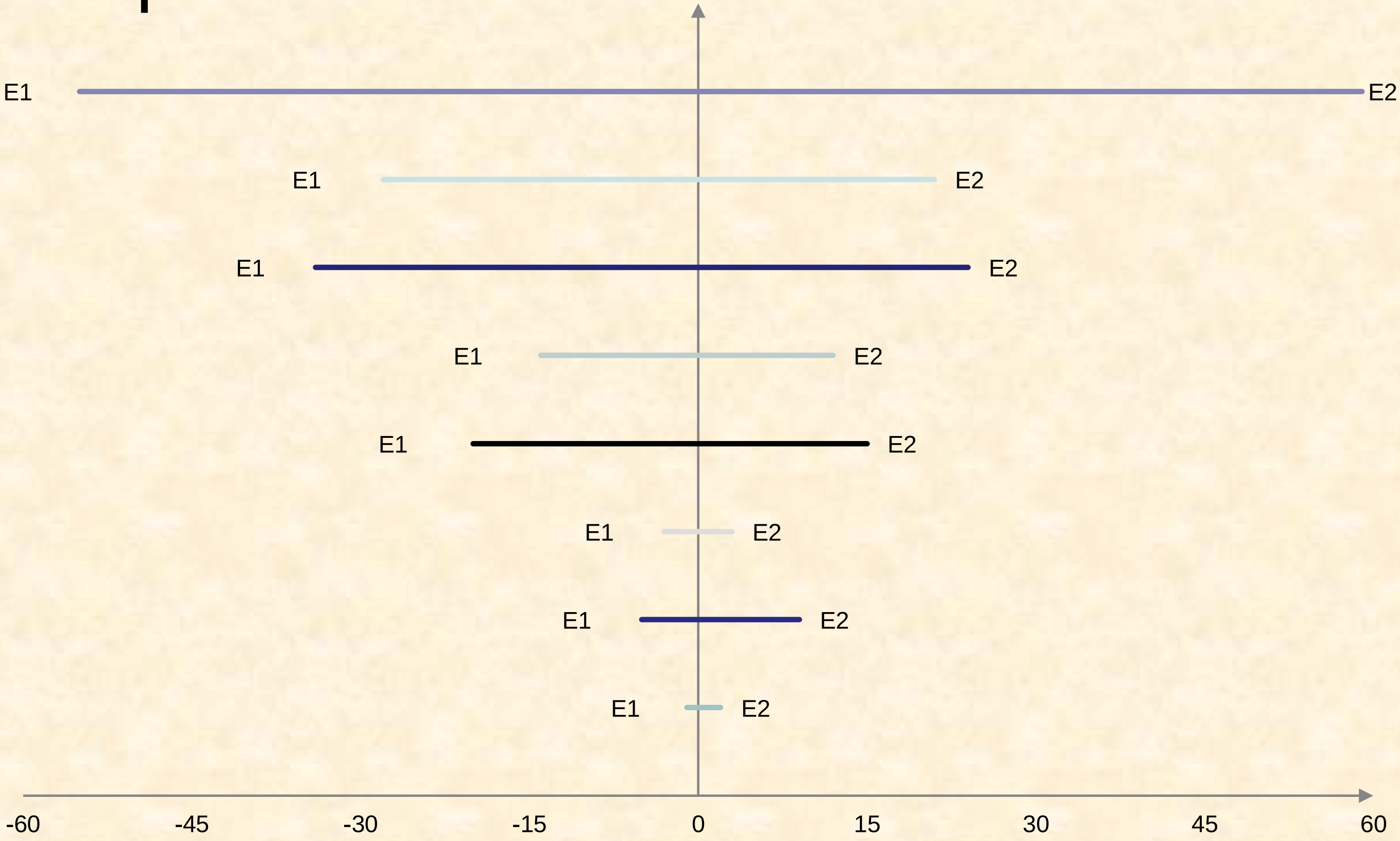
Let $E_1(x_1, x_2, ..., x_n)$ R $E_2(x_1, x_2, ..., x_n)$ denote that expressions E_1 and E_2 have relation R, where $x_1, x_2, ..., x_n$ are all input variables involved in these expressions.

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

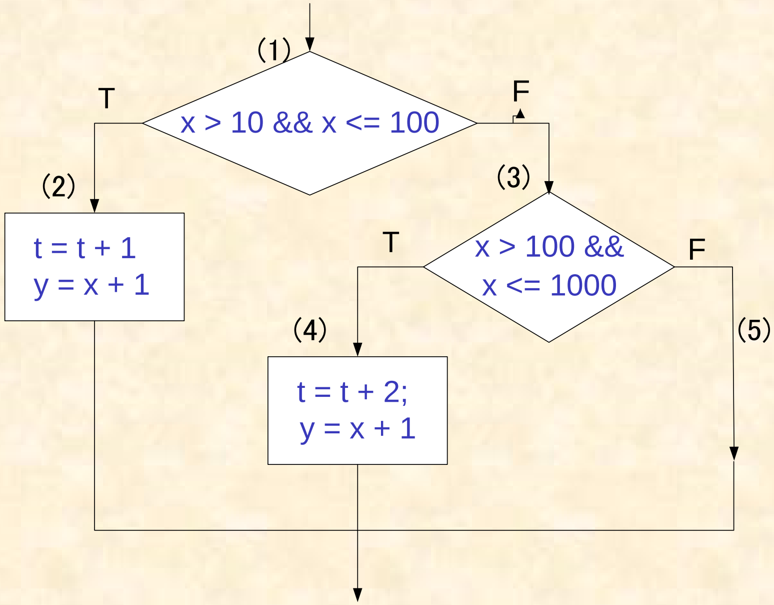
int t; //global variable declared before

Example: the functional scenario:

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

x is input

y is output



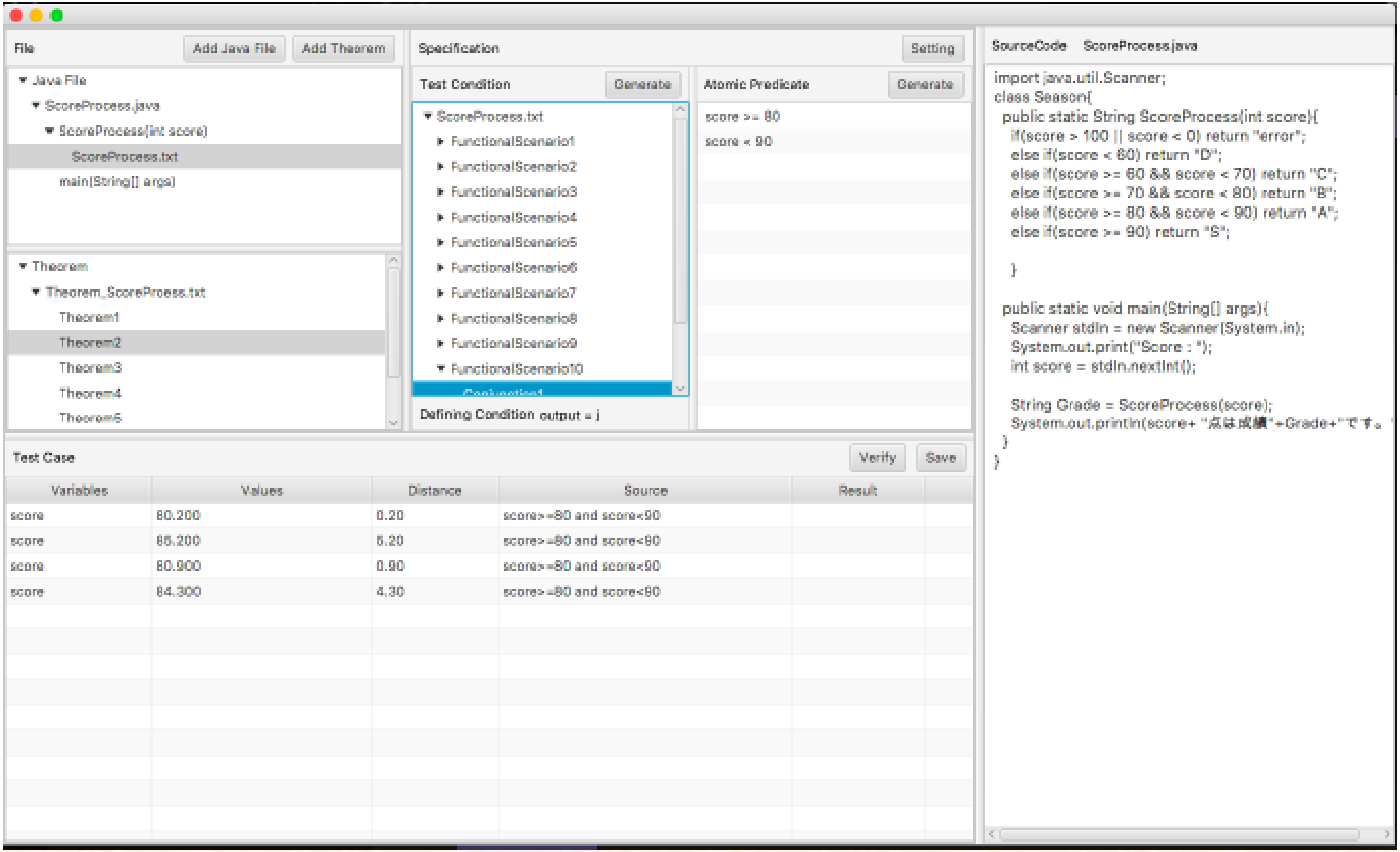
Three paths: [(1), (2)],
[(1), (3), (4)]
[(1), (3), (5)]

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



Defining Condition output = j

```
import java.util.Scanner;
class Season{
  public static String ScoreProcess(int score){
    if(score > 100 || score < 0) return "error";
    else if(score < 60) return "D";
    else if(score >= 60 && score < 70) return "C";
    else if(score >= 70 && score < 80) return "B";
    else if(score >= 80 && score < 90) return "A";
    else if(score >= 90) return "S";

    }

  public static void main(String[] args){
    Scanner stdIn = new Scanner(System.in);
    System.out.print("Score : ");
    int score = stdIn.nextInt();

    String Grade = ScoreProcess(score);
    System.out.println(score+ "点は成績"+Grade+"です。
  }
}
```

Variables	Values	Distance	Source	Result
score	80.200	0.20	score>=80 and score<90	
score	85.200	5.20	score>=80 and score<90	
score	80.900	0.90	score>=80 and score<90	
score	84.300	4.30	score>=80 and score<90	

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!