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version of the functional description by considering how to deal with the situation that the requested amount to be withdrawn is greater than the balance of the account: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?

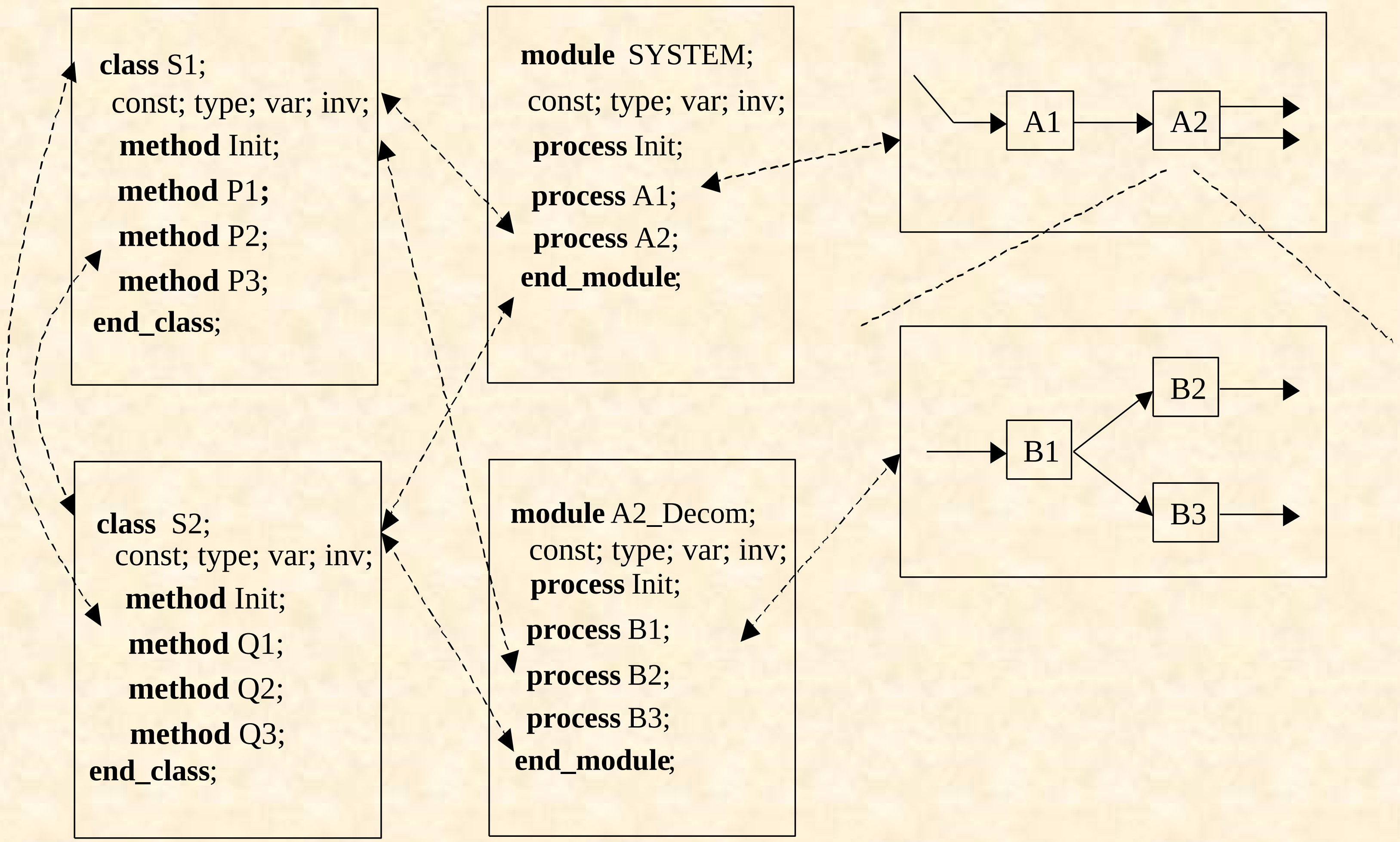
In SOFL we use module for functional abstraction.

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;
```

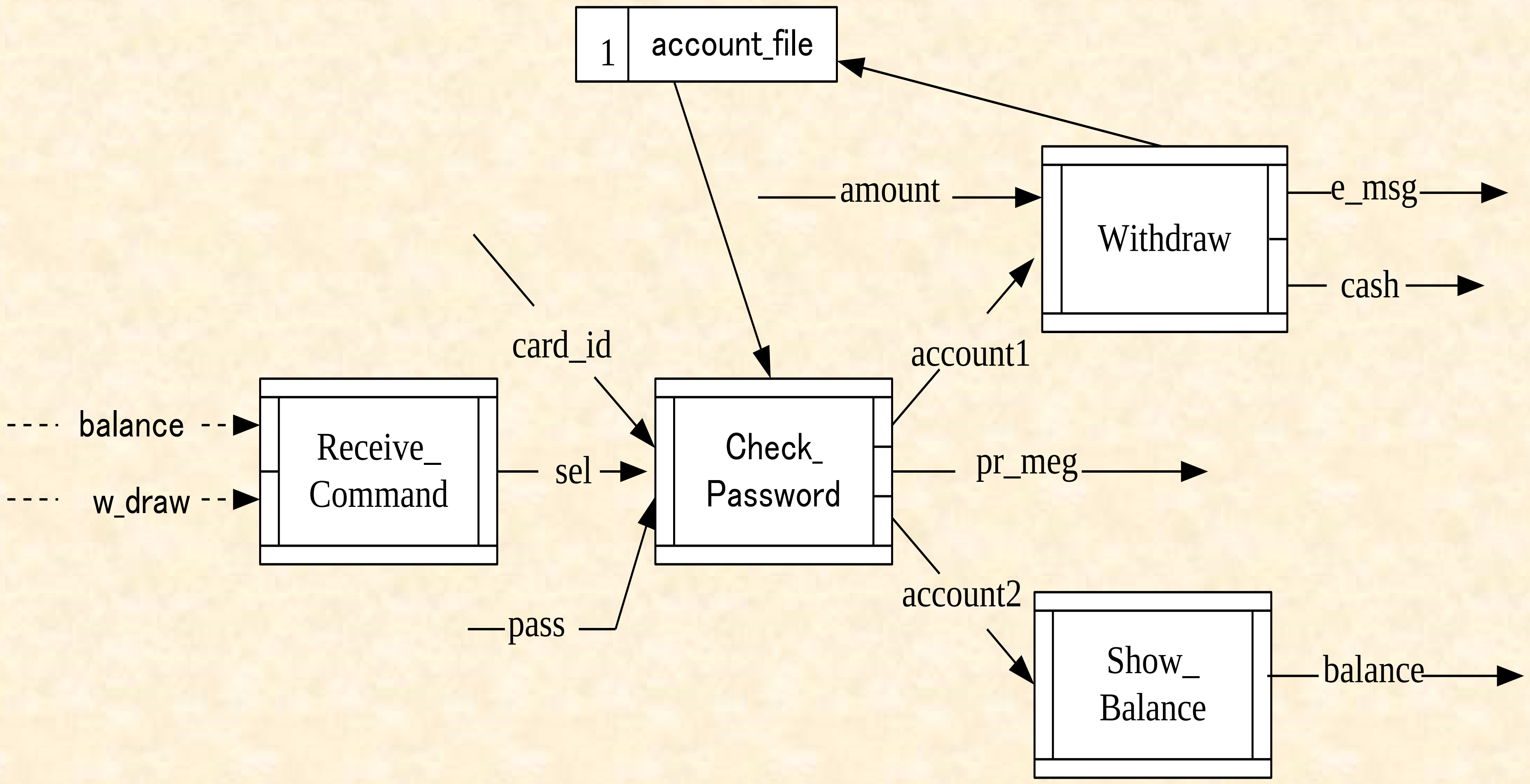
```
module SYSTEM;
  const; type; var; inv;
  process Init;
  process A1;
  process A2;
end_module;
```

```
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;
```

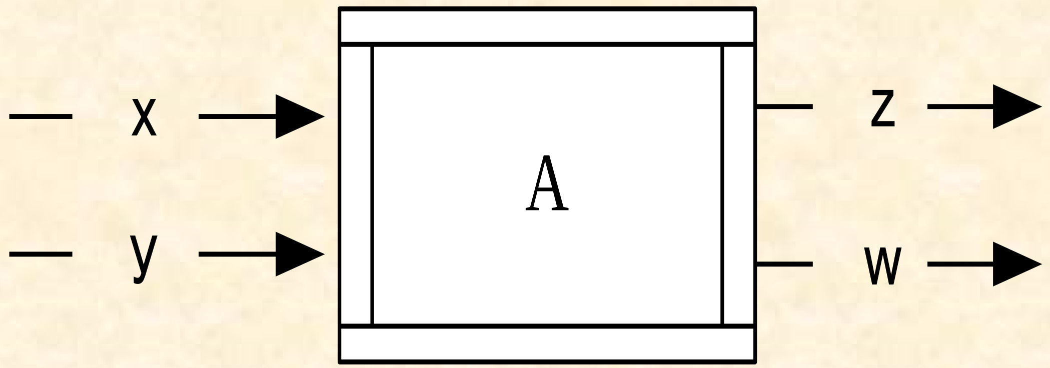
Condition Data Flow Diagrams (CDFD)



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)

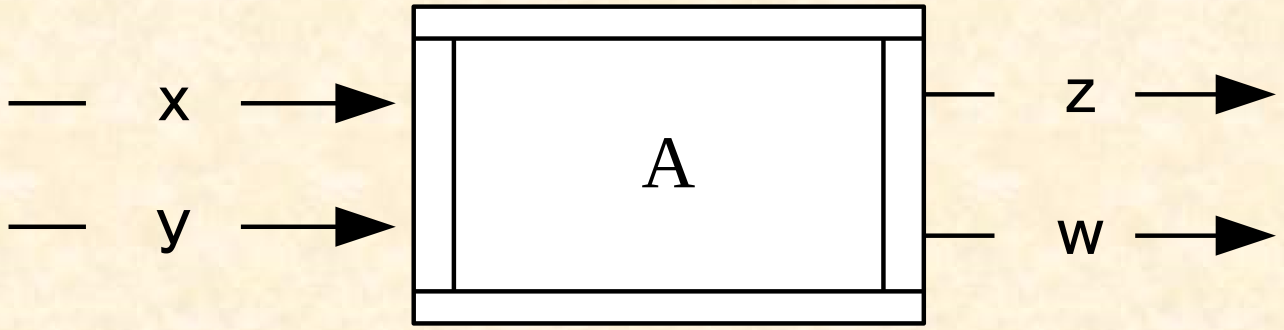
The meaning of process A:

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:

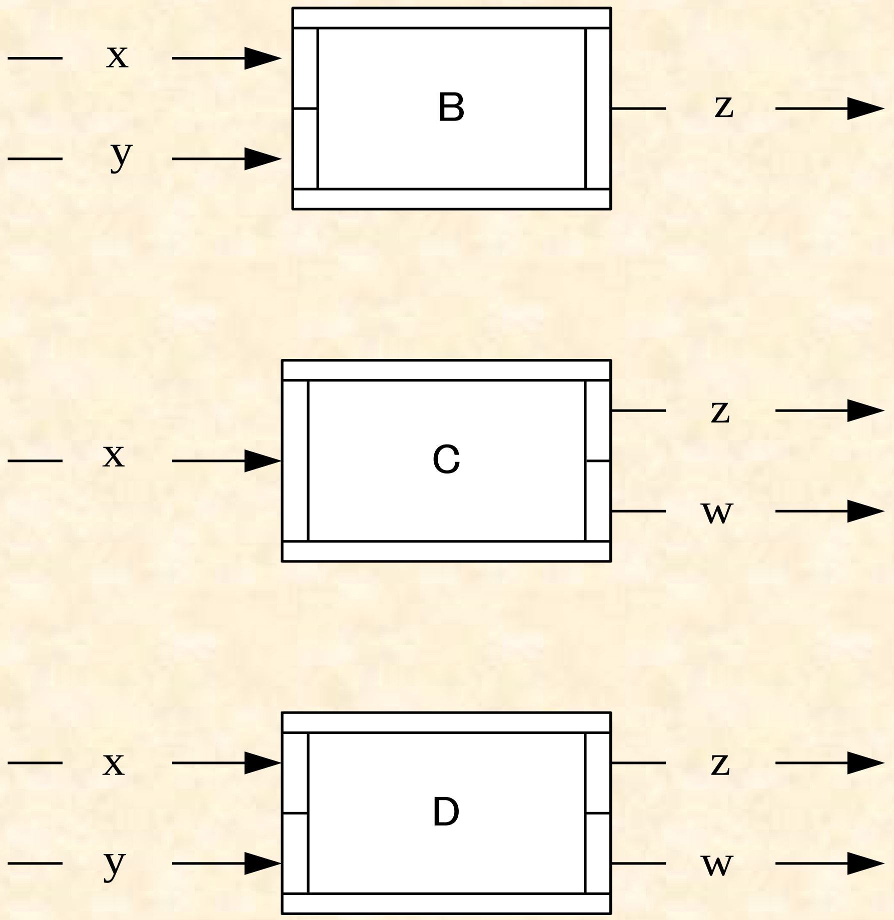
```
process A(x, y: int) z, w: int
end_process
```

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
```

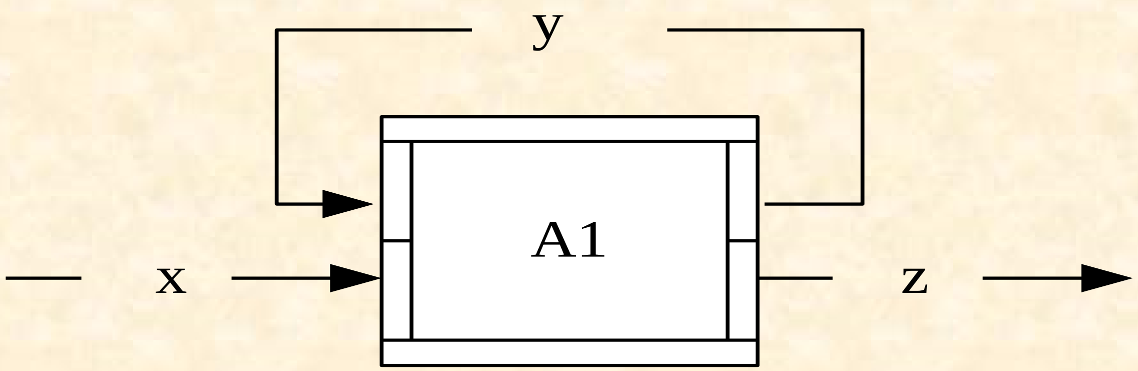
This specification does not tell exactly which of z and w will be generated as the result of an execution of process C. A more deterministic specification is:

```
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
```

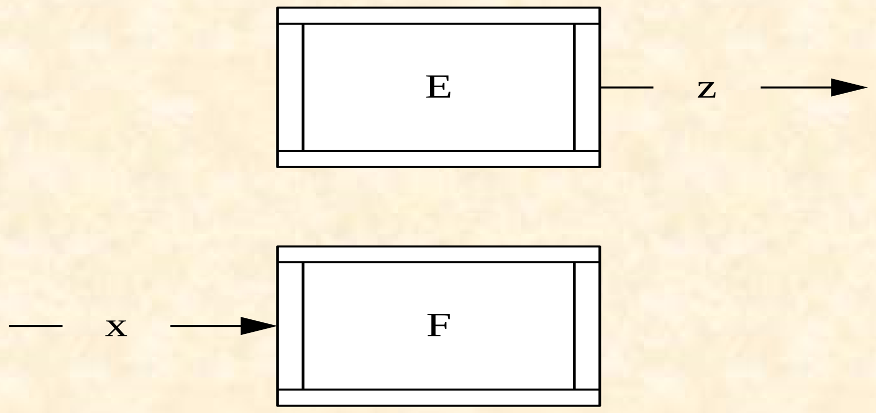
The specification of a process with a data flow loop



```
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.

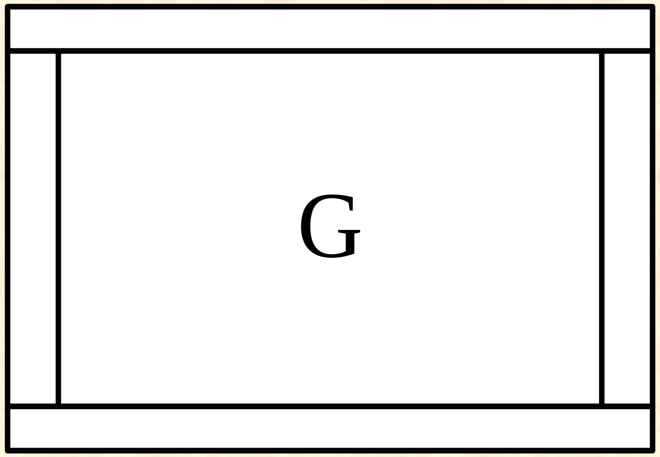
A process may have no input or output data flow



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

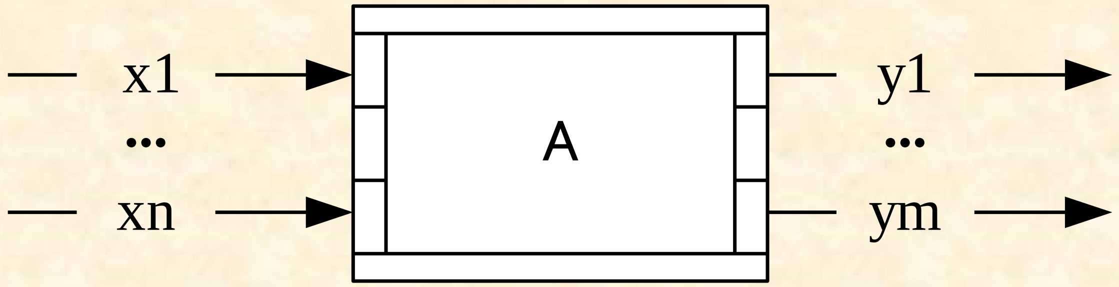
```
process F(x: nat0)
pre  x > 5
post true
end_process
```

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
```

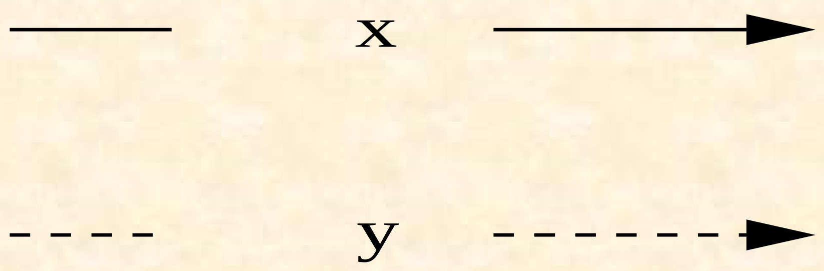
Each xi_dec (i =1..n) is a set of input variable declarations separated by comma, such as:

    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

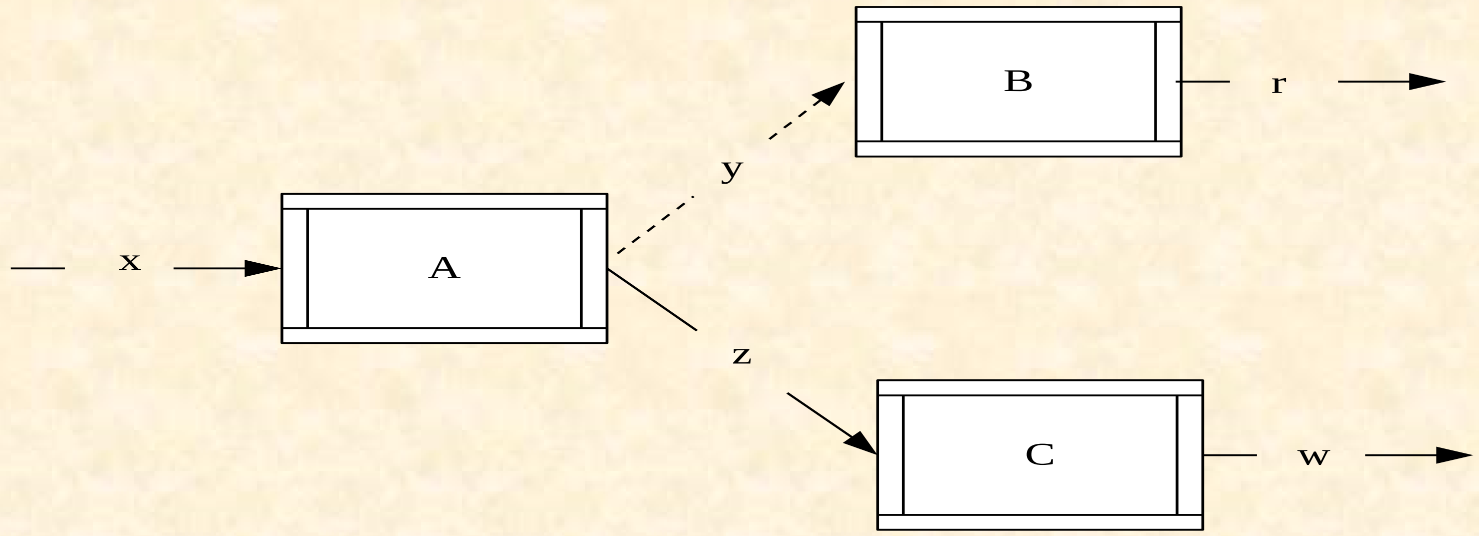
A data flow represents a data transmission from one process to another.



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

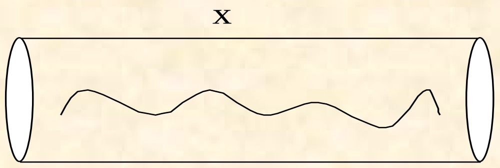


Active data flow: (1) provide useful value, (2) enable processes.
Control data flow: (1) enable processes.

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.

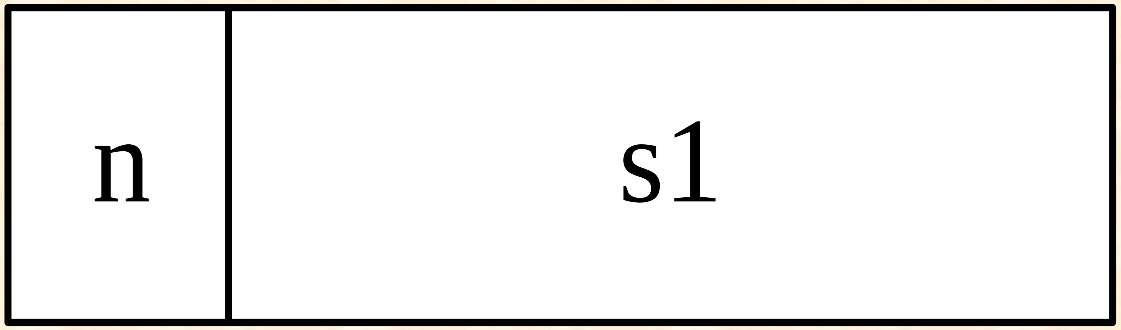
Expression of an available data flow

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)

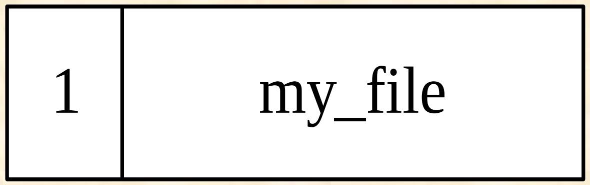
Data stores

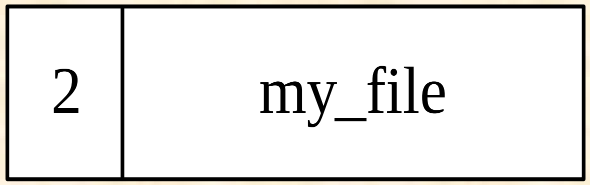
Definition: A data store, or store, is a variable holding data in rest.



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.





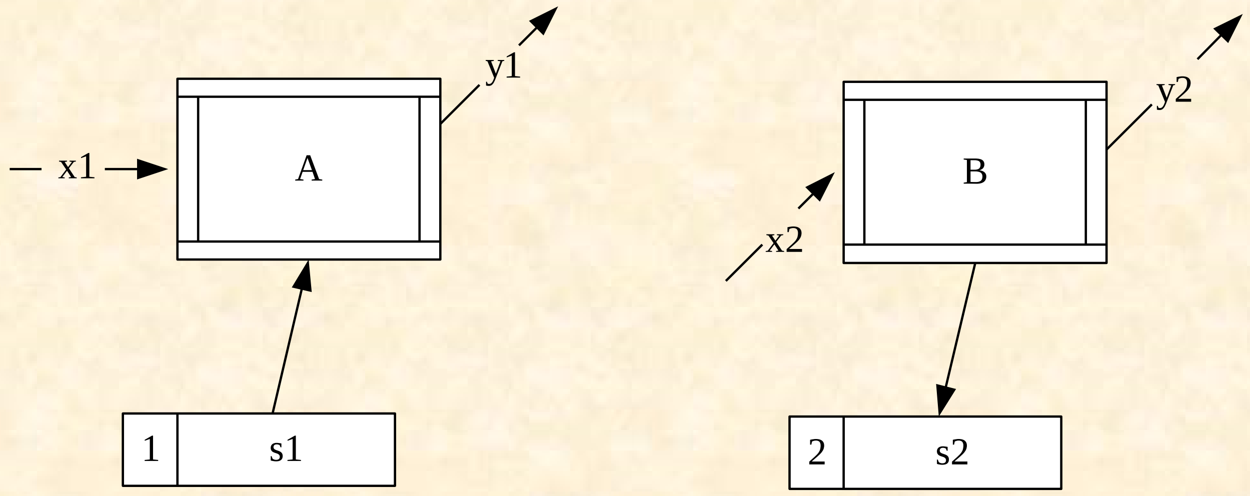
To distinguish them, we may use the following names to represent these two stores in the formal specification:

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,

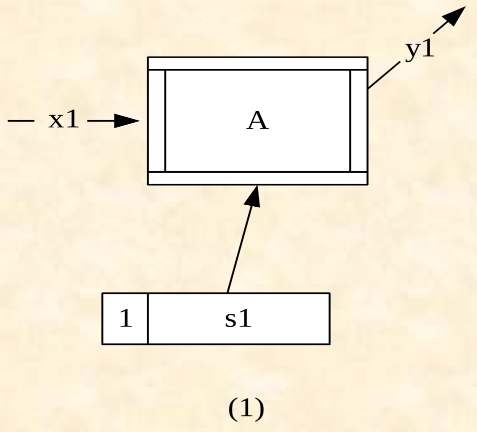


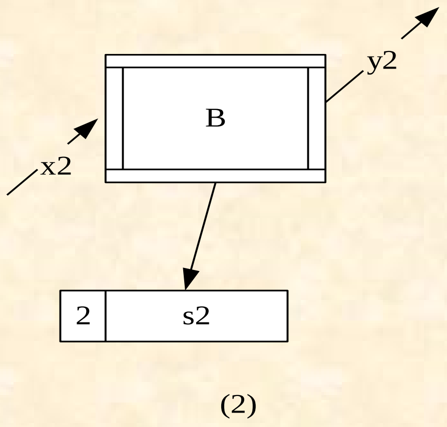
(1) (2)

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





```
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.

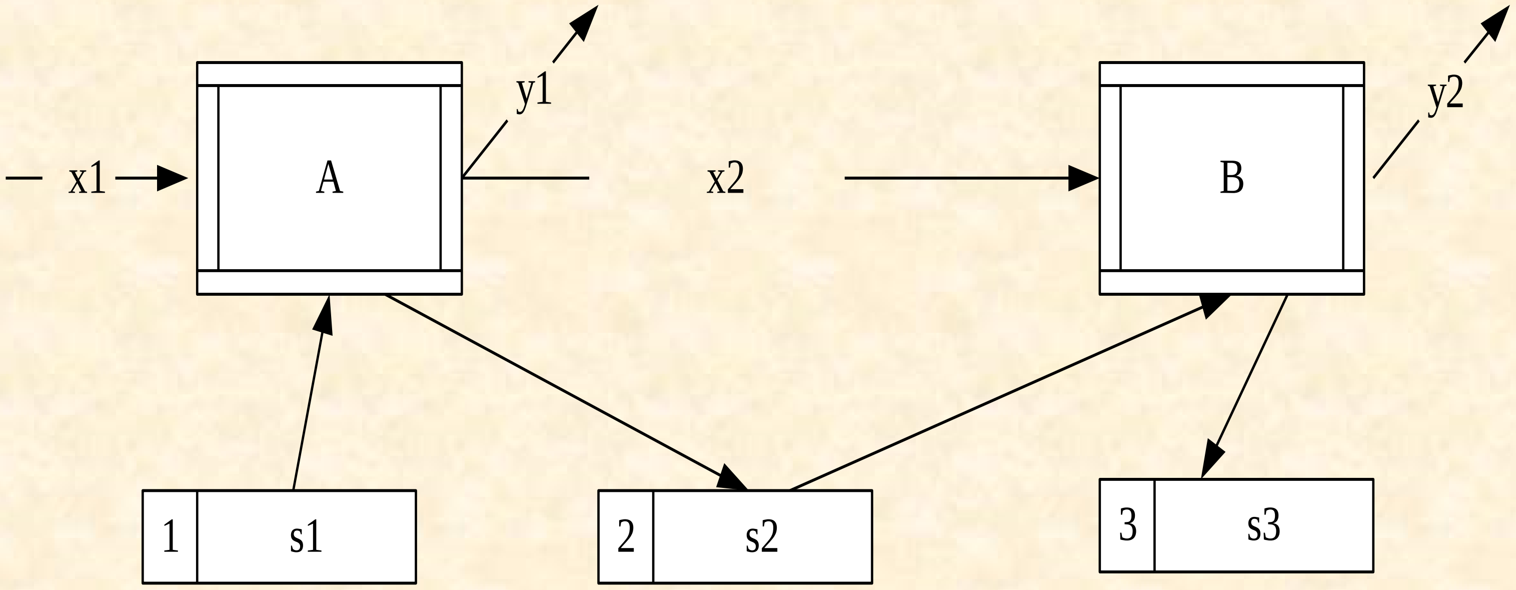
Convention: if a state variable is rd type of variable, then in the postcondition we use the non-decorated variable to denote both the initial value and final value of the variable, because they are the same in this case.

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.

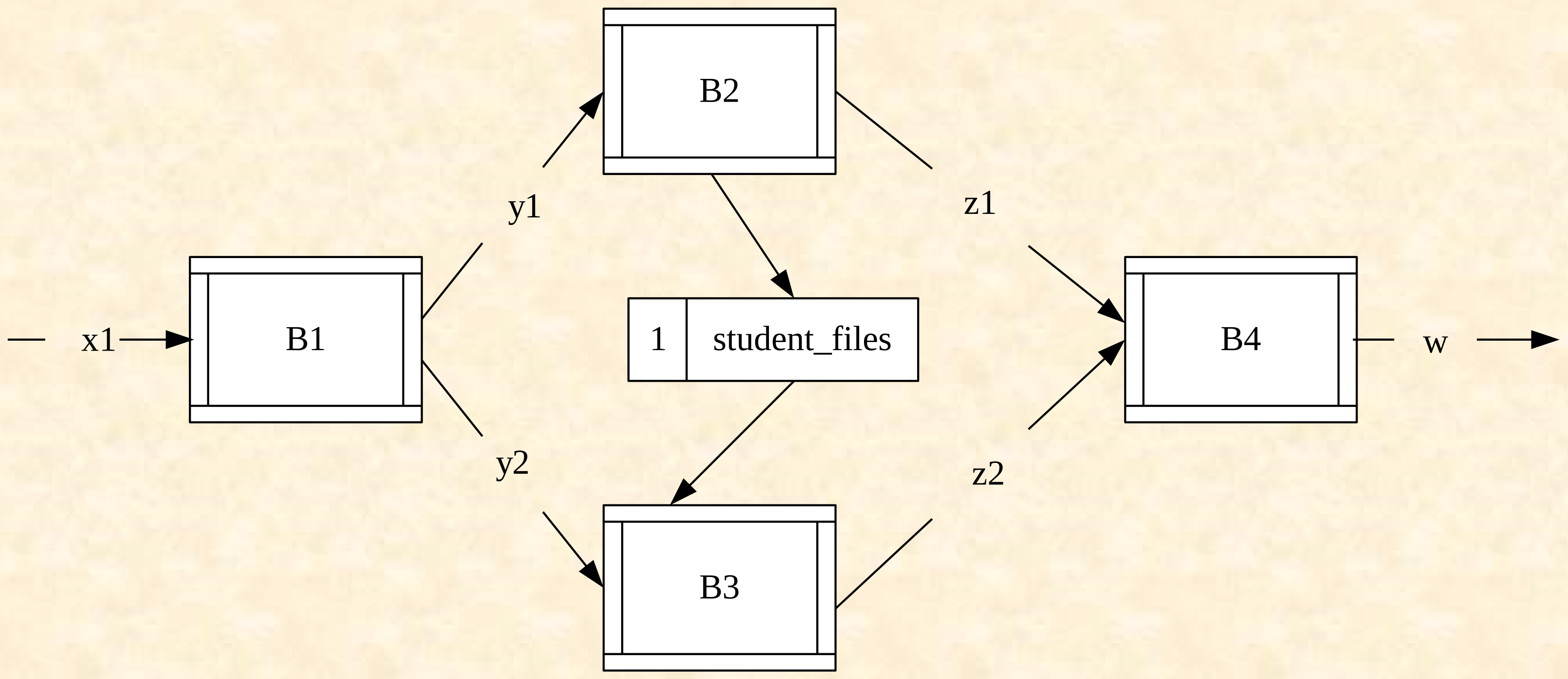
  Example: card_id, pass, w_draw

Restriction on parallel processes

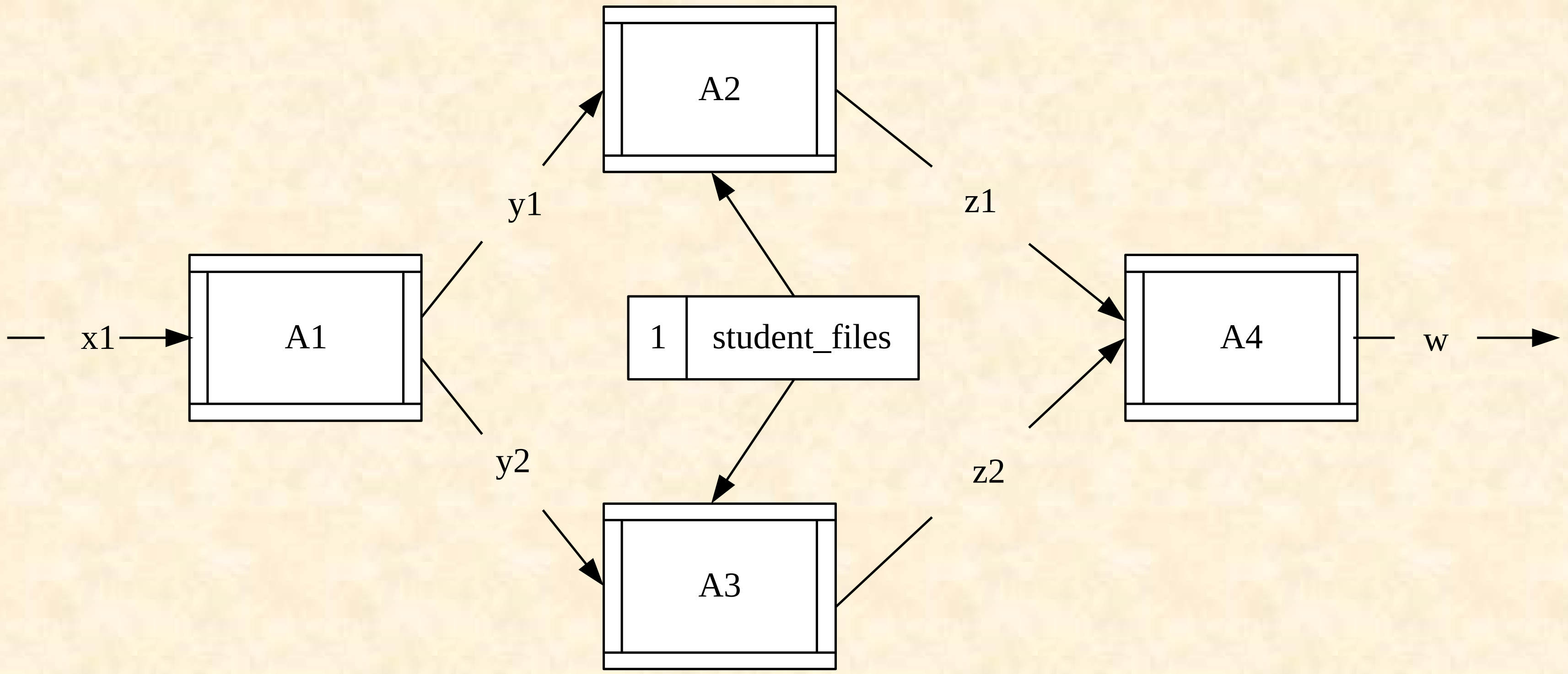
Two parallel processes cannot read from and write to the same data store. Thus, we can avoid possible confusion in operation on the data store.

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

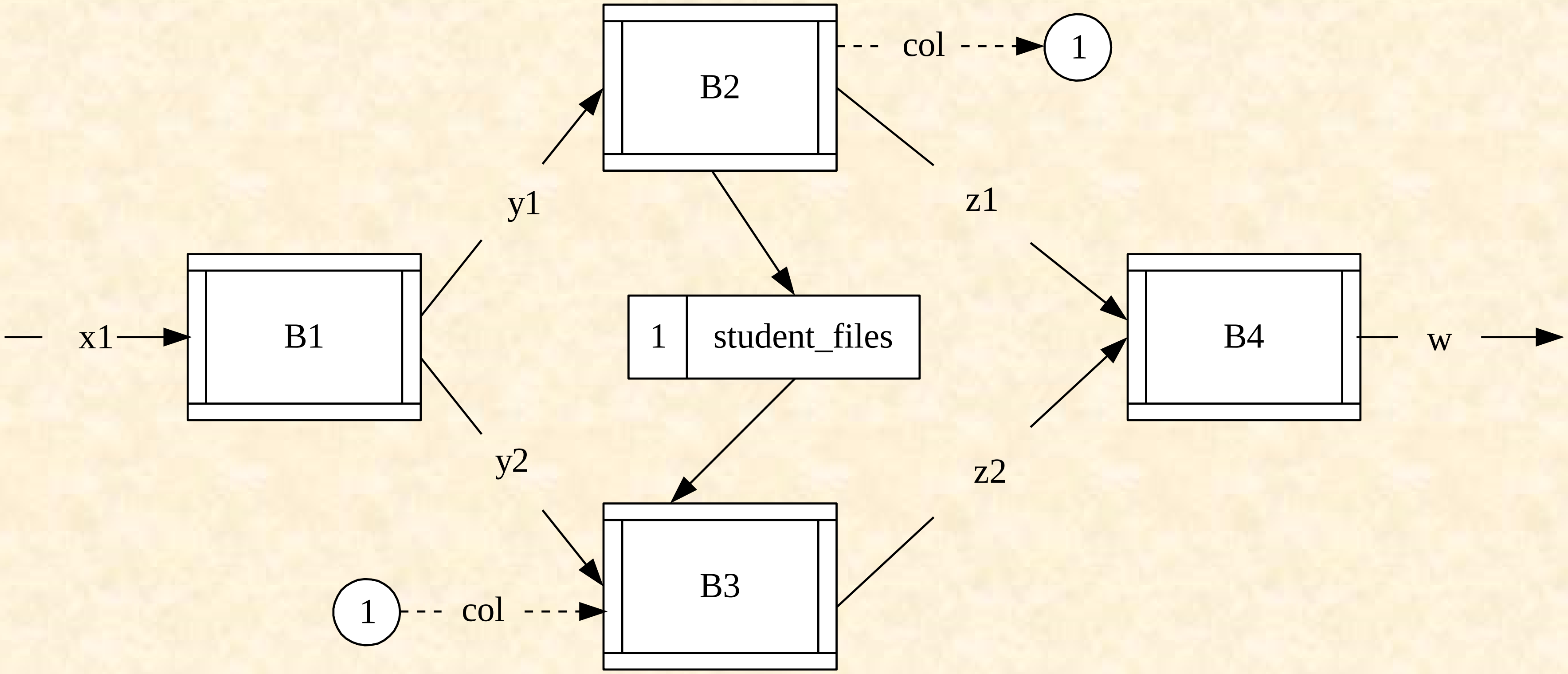
Example: the CDFD below is not allowed.



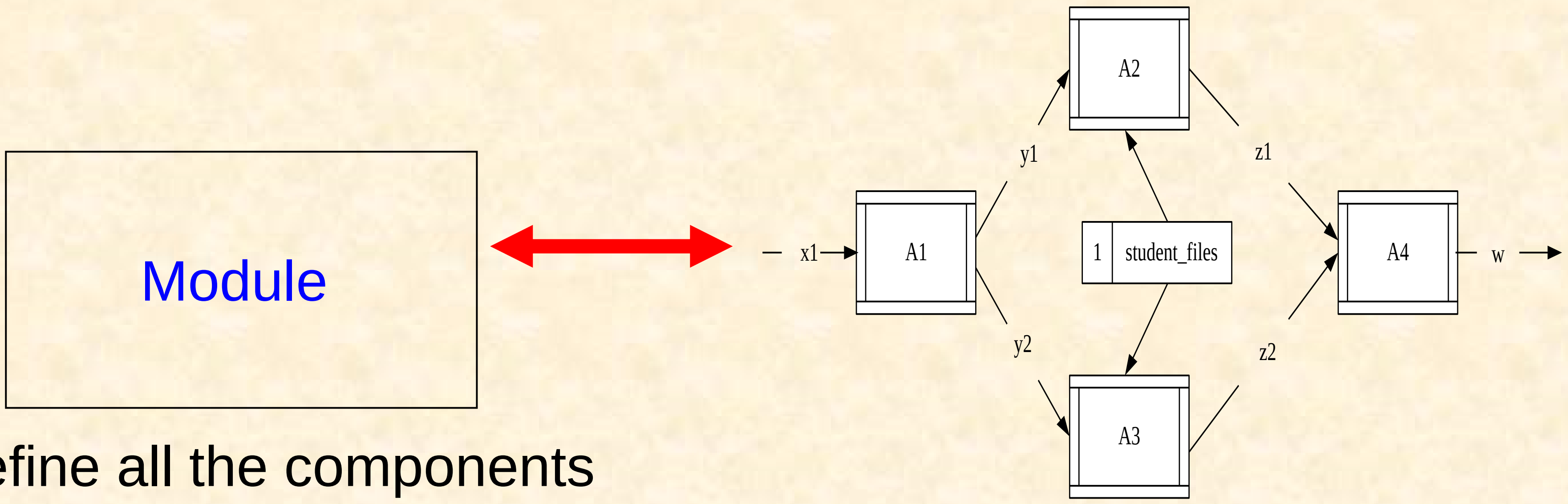
Example: the CDFD below is allowed.



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:

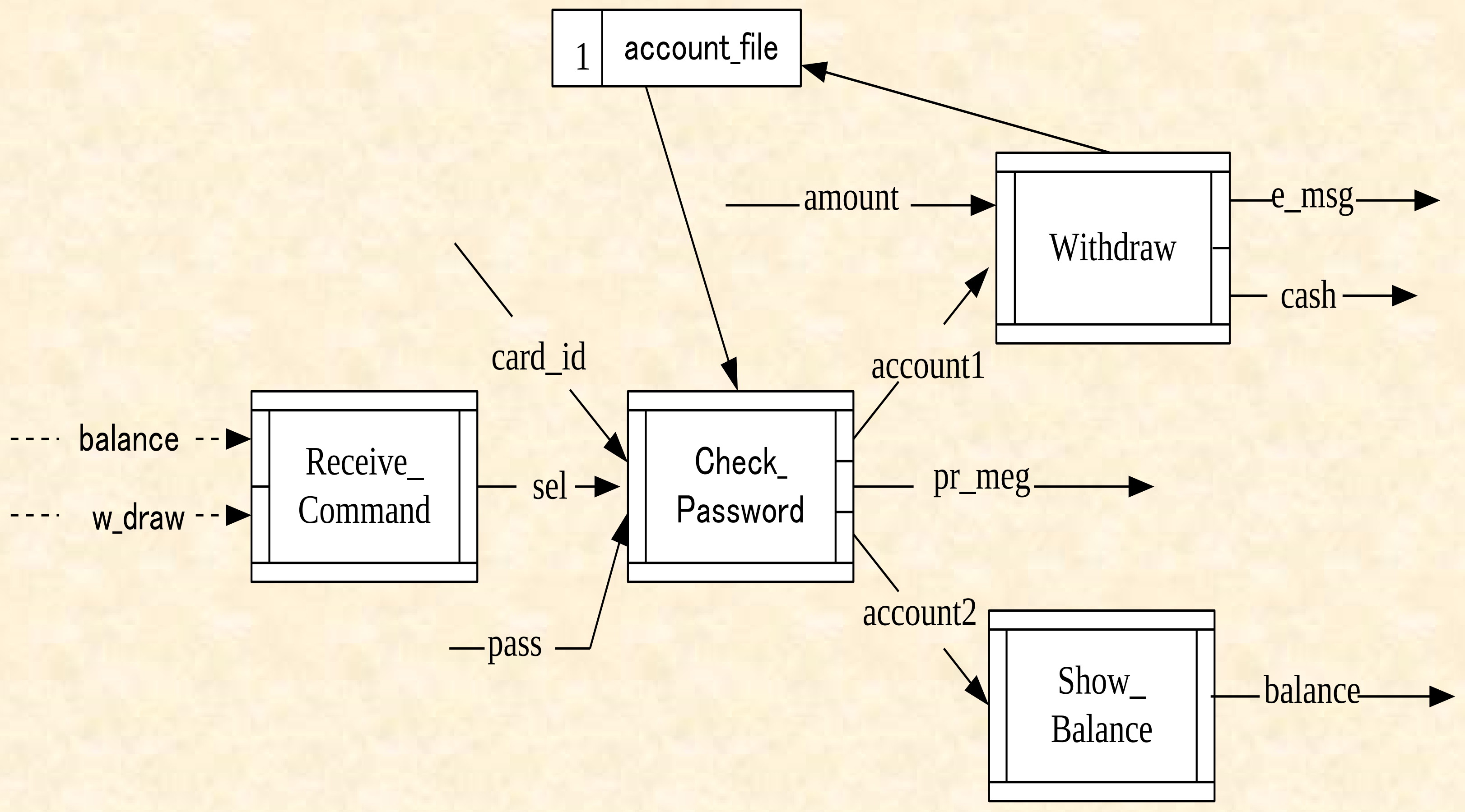
```
var
   student_files: set of Address; /*student_files is
                                    defined as a
                                    collection of
                                    home addresses,
                                    and each address is
                                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

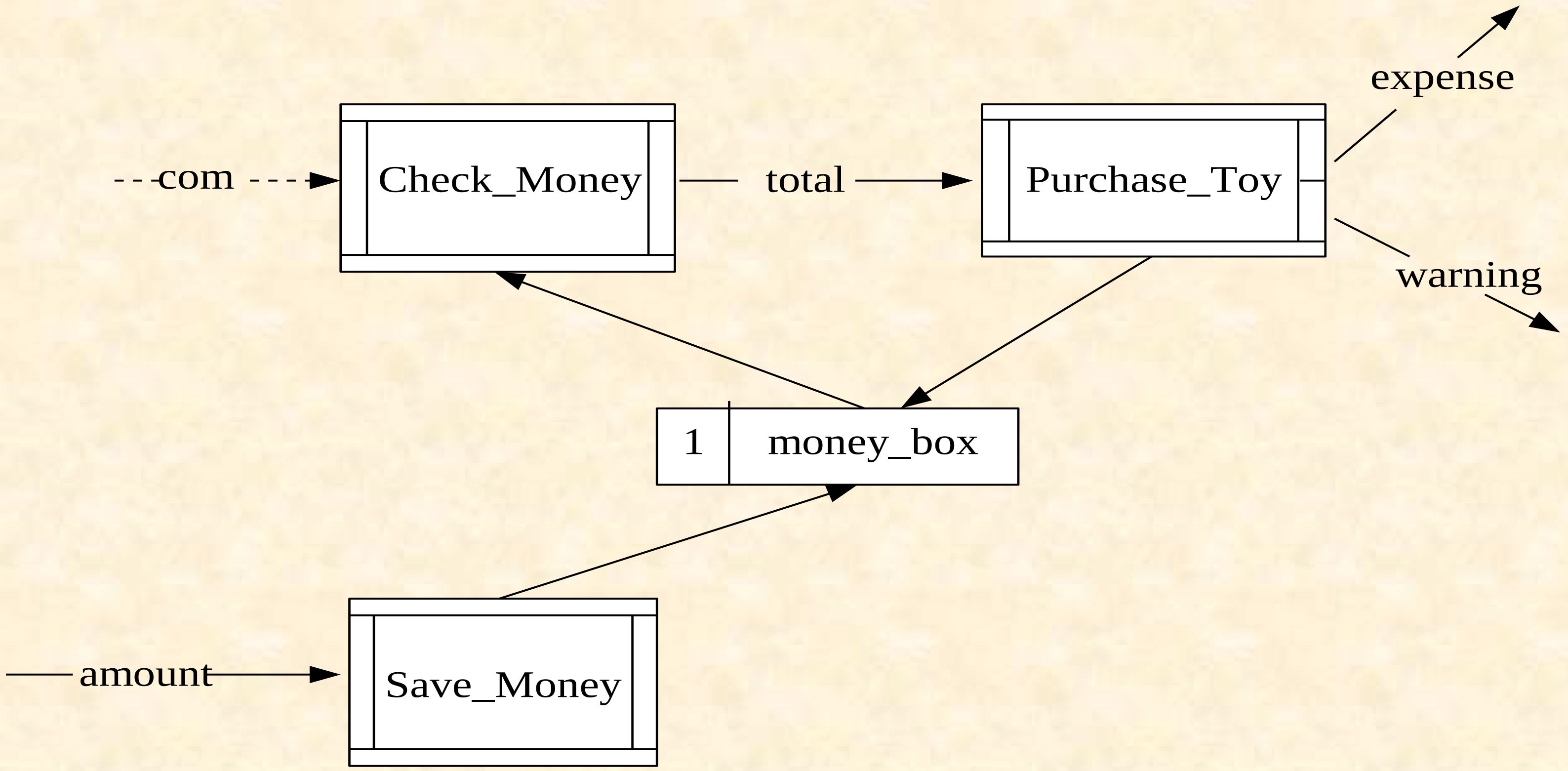
Class exercise 2

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:

$n! = n * (n - 1) * (n - 2) * \ldots * 3 * 2 * 1$

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.