4. The Module

Definition 4.1 A module is a functional abstraction: it has a behavior represented by a graphical notation, known as condition data flow diagram (CDFD), and a structure to encapsulate data 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 based on the predicate logic.
4.1 Module for abstraction

An effective way to gain the understanding of system function is **abstraction and decomposition**

Definition 4.2 **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 ATM functional abstraction

(1) Provide the buttons showing the balance and withdraw for selection.
(2) Insert a cash-card and supply a password.
(3) If showing the balance is selected, the current balance is given.
(4) If withdraw is selected, the amount of the money to withdraw is properly provided.
(5) The requested amount of money must be supplied in cash.
Abstraction may have different levels: For example, if we refine function (4) in the precious abstraction, we get a refinement (concrete version):

(4’) If withdraw is selected and the password is correct, the amount of the money to withdraw is provided; otherwise, if the password is wrong, a message for reentering the correct password must be given.
We can refine (4’) further to get the following concrete version of functional description by considering how to deal with the situation that the requested amount to be withdrawn is greater than the balance of the account: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is correct, and the requested amount is less than the balance of the account, the money of the requested amount will be withdrawn. Otherwise, if either the password is wrong or the requested amount is greater than the balance, an appropriate message must be displayed for correction.
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 speaking, a module has the following structure in general:
A 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;

process Init;
process A1;
process A2;
end_process;

module A2_Decom;
process Init;
process B1;
process B2;
process B3;
end_module;

process Init;
process A1;
process A2;
end_process;
module ModuleName / UpperLevelModule;
  const ConstantDefinition;
  type TypeDefinition;
  var VariableDefinition;
  inv TypeAndStateInvariants;
  behav CDFD_Figure No.;
process Init(); /* initialize the state variables local to
  this module, but do not initialize the state variables
  provided from the the outside of this module, explain
  later. This process can be omitted if there is no state
  variable local to this module is defined in the var
  section.*/
  process_1;
  process_2;
  ...
  process_n;
  function_1;
  ...
  function_m;
end-module;
4.2 Condition Data Flow Diagrams (CDFD)
4.3 Processes

A process models a transformation from input to output. It is similar to a VDM operation or a procedure in Pascal, but it may also be treated as an object in an Object-Oriented programming language (because it allows to receive various kinds of data items and provide various kinds of services for different requests).
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:

\[
\text{process } A(x: \text{ int}, y: \text{ int}) \ z: \text{ int}, w: \text{ int} \\
\text{pre } x > 0 \text{ and } y > 0 \\
\text{post } z = x + y \text{ and } w = x - y \\
\text{end_process}
\]

or

\[
\text{process } A(x, y: \text{ int}) \ z, w: \text{ int} \\
\text{pre } x > 0 \text{ and } y > 0 \\
\text{post } z = x + y \text{ and } w = x - y \\
\text{end_process}
\]
A process specification with no specific precondition or postcondition:

\[
\text{process } A(x, y: \text{int}) \ z, w: \text{int} \\
\text{pre } \text{true} \\
\text{post } z = x + y \ \text{and} \ w = x - y \\
\text{end_process}
\]

\[
\text{process } A(x, y: \text{int}) \ z, w: \text{int} \\
\text{pre } x > 0 \ \text{and} \ y > 0 \\
\text{post } \text{true} \\
\text{end_process}
\]
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 pre and postconditions:

```
process A(x, y: int) z, w: int
end_process
```
Processes with multiple ports
Specifications of process B

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

The following specification is inappropriate:
process B(x: int | y: int) z: int
pre x > 0 and y > 0
post z = x + 1 and z = y - 1
end_process
Another possibility of process B

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

Where true(y) is a predicate (not a truth value) defined as follows:
true(y) = true if y is available
true(y) = nil if y is unavailable.
Specifications of process C

process C(x: int) z: int | w: int
pre  x > 0
post z = x + 1 or w = x * 2
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: int | w: int
pre  x > 0
post x < 10 and z = x + 1 or x >= 10 and w = x * 2
end_process
Incorrect expression in the specification of process C

process C(x: int) z: int | w: int
pre x > 0
post z = x + 1 and w = x * 2
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 x <> nil and Q_1(z, x) or
     y <> nil 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 true(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 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.
A process may have one empty input port and/or empty output port

process A1(x: int | ) | y: int
pre  true
post x > 0 and y = x + 1 or 
    x <= 0 and y = x -1 or 
    x = nil
end_process
The general form of a process

process $A(x_1_{\text{dec}} | x_2_{\text{dec}} | ... | x_n_{\text{dec}})$

$y_1_{\text{dec}} | y_2_{\text{dec}} | ... | y_m_{\text{dec}}$

pre $P(x_1, x_2, ..., x_n)$

post $Q(x_1, x_2, ..., x_n, y_1, y_2, ..., y_m)$

end_process
Each \( \text{xi}_\text{dec} (i = 1..n) \) is a sequence of input variable declarations separated by comma, such as:

\[
\text{xi}_1: \text{Ti}_1, \text{xi}_2: \text{Ti}_2, \ldots, \text{xi}_n: \text{Ti}_n
\]

where \( \text{xi}_1, \text{xi}_2, \ldots, \text{xi}_n \) are the data flow variables connecting to input port \( \text{xi} \), and \( \text{Ti}_1, \text{Ti}_2, \ldots, \text{Ti}_n \) are their types, respectively.
A rule for input and output data flow variables

All the input data flow variables must be different, and so are output data flow variables, in order to avoid confusion in the precondition and/or postcondition of the process.

The following process is therefore illegal:

For example, if let precondition be: $x > x + 1$. Then we do not know which $x$ in the expression corresponds to which input data flow $x$ in the graphical representation, and it looks like the precondition cannot become true.
4.4 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, like x, and another is called control data flow, like y.
Example for explaining the meaning and 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 and data through the variable is available.
Definition 4.3 Let $x$ be a data flow variable of type $T$. Then, $x$ is defined if a value of $T$ is bound to $x$. Otherwise, $x$ is undefined.

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: \texttt{sign}, which means signal.

\[
\text{sign} = \{!\}
\]

An active data flow must not be declared with the type \texttt{sign}.
Expression of the definedness of a variable

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

- $\text{bound}(x)$
- $x \not= \text{nil}$
Definition 4.4 A data store, or store, is a variable holding data in rest.

- $s_1$ 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 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 make 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 during its execution, which is represented by a directed line pointing to the store from the process or pointing to the process from the store.
For example,

Process A read data from store s1, which is regarded as a state variable of process A.

Process B write data to store s2, which is in fact a state 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 rw s2: int
pre  x2 > 0
post y2 = ~s2 + x2 and
    s2 = ~s2 - x2
end_process
Decorated state variables in the postcondition:

\(\sim s2\) denotes the value of variable \(s2\) before execution of process B. Such a value is known as initial value of variable \(s2\).

\(s2\) denotes the value of variable \(s2\) after execution of process B. Such a value is called 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.
Multiple connections among processes and stores
The general structure of a process specification

process \( A(x_1: Ti_1 \mid x_2: Ti_2 \mid \ldots \mid x_n: Ti_n) \)
\( \quad y_1: To_1 \mid y_2: To_2 \mid \ldots \mid y_m: To_m \)
\( \quad \text{ext acc}_1 z_1: Te_1 \)
\( \quad \quad \quad \quad \text{acc}_2 z_2: Te_2 \)
\( \quad \quad \quad \quad \ldots \)
\( \quad \quad \quad \quad \quad \text{acc}_q z_q: Te_q \)
\( \quad \text{pre} \quad P(x_1, x_2, \ldots, x_n, z_1, z_2, \ldots, z_q) \)
\( \quad \text{post} \quad Q(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m, \)
\( \quad \quad \quad \quad \quad ~z_1, ~z_2, \ldots, ~z_q, z_1, z_2, \ldots, z_q) \)
\( \text{end_process} \)
The proof obligation for a process

The condition to ensure that an implementation satisfies the specification of a process is called proof obligation for the process.

Example:
The proof obligation for process B:

\[
\text{forall}[x_1, x_2: \text{int}, x_3: \text{int}, \sim s: \text{int}] | \\
((x_1 > 0 \text{ and } x_2 > 0 \text{ or } \text{nil} > 0 \Rightarrow \\
(\text{exists}[y_1: \text{int}, s: \text{int}] | y_1 > x_1 + x_2 + \sim s \\
\quad \text{and } s = \sim s - (x_1 + x_2) \text{ or} \\
\quad \text{nil} + \text{nil} \geq \text{nil} + \sim s \text{ and } s = \sim s + \text{nil} \\
)) \text{ or} \\
(\text{nil} > 0 \text{ and } \text{nil} > 0 \text{ or } x_3 > 0 \Rightarrow \\
(\text{exists}[y_2, y_3: \text{int}, s: \text{int}] | \text{nil} > \text{nil} + \text{nil} + \sim s \\
\quad \text{and } s = \sim s - (\text{nil} + \text{nil}) \text{ or} \\
\quad y_2 + y_3 \geq x_3 + \sim s \text{ and } s = \sim s + x_3 \\
))
\]
The implication of the proof obligation

- It requires that the process be satisfiable (that is, there exists a program that implements the specification).
- It requires that the program implementing the process specification terminates.
- If precondition is not satisfied, anything can happen and in this case the process specification is not responsible for whatever happens.
A process describes a relation between initial states and final states of the process.

\[ \text{Sem}(A) = \{(s_0, s_1) \mid \text{pre}_A(s_0) \text{ and } \text{post}_A(s_0, s_1)\} \]

Process A actually associates only those initial states satisfying the precondition to the final states meeting the postcondition. All the initial states that do not satisfy the precondition have no precisely defined final states to be associated with.
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 words 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, and w_draw
There are three conditional structures that can be used in a CDFD:

- single condition structure
- binary condition structure
- multiple condition structure
The meaning of the single condition structure:

1. if \( x \) is available and satisfies condition \( C(x) \), \( x_1 \) is generated to have the same value as \( x \), and \( x \) is consumed.

2. if \( x \) is available and does not satisfy condition \( C(x) \), \( x \) is just consumed, without generating \( x_1 \).
The meaning of the binary condition structure:

1. if data flow $x$ is available and satisfies condition $C(x)$, then data flow $x_1$ with the same value as $x$ will be made available.
2. otherwise, if $C(x)$ is false, data flow $x_2$ with the same value as $x$ will be made available.
3. in either case of 1 and 2 above, the input data flow $x$ will be consumed.
4. if $C(x)$ is "undefined", $x$ is just consumed, without producing any of $x_1$ and $x_2$.

Note that the small black circle marks the branch when $C(x)$ evaluates as false.
The semantics of the multiple condition structure

1. if x is available and satisfies condition $C_i(x)$ ($i=1...n$), the corresponding data flow $x_i$ with the same value as x is made available.

2. otherwise, if $C_i(x)$ evaluates as false, then $C_{i+1}(x)$ will be tested and such tests go on until one of the $C_i(x)$ ($i=1...n$) evaluates as true.

2. however, if none of $C_1(x)$, $C_2(x)$, ..., $C_n(x)$ is satisfied by x, then $x_{n+1}$ with the same value as x is made available as default.

3. in any case above, the generation of the output data flow results in the consumption of x.
Merging and Separating structures

(1) Merging structure
It defines that $x$ is generated as a composite object that has three fields: $x_1$, $x_2$, and $x_3$.

(2) Separating structure
It means that the composite object $x$ is decomposed into three individual data flows: $x_1$, $x_2$, and $x_3$. 
An example of using the merging and separating structures:
Diverging structures

- **Nondeterministic structure:** transforming $x$ to either $x_1$, $x_2$, or $x_3$, but only one of them.

- **Broadcasting structure:** transforming $x$ to $x_1$, $x_2$, and $x_3$. 
An example of using a nondeterministic structure:
An example of using a broadcasting structure:
Renaming structure

The renaming structure passes the value of $x_1$ to $y_1$, the value of $x_2$ to $y_2$, ..., the value of $x_n$ to $y_n$. 

An example of using the renaming structure in a CDFD: suppose that process A is defined as follows:

process A1(y1: int, x1: int)
pre x1 + y1 > 0
post true
end_process

We want to use the functionality of process A1 in the CDFD, but we need to change the data flow variables in order to keep consistent with their specifications.
Connecting structures

The connecting nodes allows a single CDFD to be drawn on different pages or to avoid possible confusion.
An example of a CDFD without using connecting nodes:
Important issues on CDFDs

Questions: how can a CDFD be enabled, executed, and terminated?

In order to answer these important questions, we need to introduce several concepts:

- starting process
- starting node
- terminating process
- terminating node.
Starting processes

Definition 4.5 A starting process of a CDFD is a process with an empty input port or an input port whose data flows are not the output data flows of any other processes in the same CDFD.
Example: both process A1 and process A2 are a starting process of the CDFD.
Definition 4.6 A starting node is either a starting process or any of the nodes involved in the conditional structures, merging and separating structures, diverging structures, and renaming structures whose input data flows are connected to no processes and nodes in the same CDFD.
Example: the conditional node: $x > 5$ is a starting node.
Definition 4.7 A terminating process is a process with an empty output port or an output port whose data flows are not the input data flows of any other processes in the same CDFD.

According to this definition, several kinds of processes can be a terminating process. The following list gives all the possibilities:

- a process with no output data flow (empty output port).
- a process with one output port whose data flows connect to no processes.
- a process with one empty output port together with other non-empty output ports.
- a process with output data flows that are connected to no processes in the same CDFD.
Example: both process A4 and A5 are a terminating process.
Terminating nodes

Definition 4.8 A **terminating node** is either a terminating process or a node in the conditional structures, merging and separating structures, diverging structures, and renaming structures that has no output data flow connecting to other processes and nodes.
Example: process $A_4$, $A_5$, and the node in the broadcasting structure are a terminating node.
Enabling and Executing a CDFD

Definition 4.9 A CDFD is enabled if one of its starting nodes is enabled.

Definition 4.10 A CDFD starts execution if one of its input node starts execution.
Example:
Definition 4.11 An execution of a CDFD is said to be terminated if the following two conditions are satisfied.

(1) All of the terminating nodes are terminated.
(2) No process in the CDFD is enabled.
Example:
Restriction on parallel processes

Two parallel processes cannot read from and write to the same data store. Thus, we can avoid possible confusion in operating 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_file 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.
Disconnected CDFDs

A disconnected CDFD is a disconnected graph: there exist at least two processes or structures, such as conditional, diverging structures, which are not connected to each other by data flows in the CDFD.
Example: both the CDFDs below are a disconnected CDFD.
External processes

When modeling a system with a CDFD, we should pay attention not only to the correctness and preciseness of the CDFD, but also to the readability of the CDFD. One way to improve the readability of a CDFD is to show explicitly the entities that provide input data flows to the starting nodes of the CDFD or that receive output data flows produced by terminating nodes.
Definition 4.12 An external process is an entity that stands outside the system under construction. It can be a person, organization, and so on.

An external process is represented by a dotted line box. For example, external process A.
An example of using external processes:
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;
InitializationProcess;
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:

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

Example:

```plaintext
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;
```

The store variables together with all the data flow variables (as well as their values) in a CDFD constitute the state of the CDFD.
Example:

```plaintext
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 the store variables, and possibly some data flow variables).
The form of invariants:

\[ \text{inv} \]
\[ \text{Invariant}_1; \]
\[ \text{Invariant}_2; \]
\[ \ldots \]
\[ \text{Invariant}_v; \]

Example:

\[ \text{inv} \]
\[ \text{forall}[x: \text{Address}] \mid \text{len}(x) \leq 50; \quad \text{card}(	ext{student_files}) \leq 1000; \]

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

\[ \text{place: Address}; \]

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:

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

```plaintext
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:

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

- **Receive_Command**
  - card_id
  - pass
  - sel

- **Withdraw**
  - amount
  - cash
  - e_msg

- **Show_Balance**
  - balance
  - account
  - pr_msg
  - account1
  - account2
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 natural number with four digits. */

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 balance <> nil and sel = true or w_draw <> nil 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 output 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 rw 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_msg = "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;
Compound expressions

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 is used to declare some identifiers denoting expressions in predicate expressions. Two let expressions are designed for this purpose.

2.1 The first let expression takes the format:

```latex
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 other than a variable (whose value may change). This let expression is equivalent to the expression:

```latex
P[E_1/v_1, E_2/v_2, ..., E_n/v_n]
```
Example 1:

```
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
```
Example 2:

```
let checking = exists![x: account_file] |
    x.account_no = card_id and
    x.password = pass

in
    if checking
    then if sel = false
        then account1 = get({x | x inset
            account_file and checking})
        else account2 = get({x | x inset
            account_file and checking})
    else pr_meg = "Reenter your pass or insert
        the correct card"
```
2.2 The second let expression:

\[
\text{let } x: T \mid R(x) \text{ in } P(x)
\]

This let expression introduces a pattern \( x \) that is bound to a value of set \( T \) (which may also be a type) satisfying condition \( R(x) \). Pattern \( x \) is usually involved in \( P \).

Note that condition \( R(x) \) may be omitted in a let expression so that pattern \( x \) will be introduced as any value in type \( T \) with no constraint. Thus, the following format can also be used:

\[
\text{let } x: T \text{ in } P(x)
\]

Example 1:

\[
\text{let } x: \text{nat} \mid x > 5 \text{ in } y > x + 1 \quad \text{let } x: \text{nat} \text{ in } y > x + 1
\]
Constraint on this kind of let expressions:

To keep this kind of let expressions simple, we do not allow the introduction of multiple binding, such as \( x_1: T_1, x_2: T_2 \). If such a multiple binding is really needed in an expression, we can use the let expression several times, as

\[
\text{let } x_1: T_1 \mid R_1(x_1) \\
\text{in} \\
\text{let } x_2: T_2 \mid R_2(x_2) \\
\text{in } P(x_1, x_2)
\]
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
Reference to pre and postconditions

Assume process A has been defined before, then we can use:

process B()
pre P and pre_A
post Q or post_A
end_process
If you want to define another process, say C, with the same input and output data flows, external variables, and pre and postconditions as those of process A, you can define C as follows:

```
process C equal A
end_process
```
Robust process specification:

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

For example, process Get is not robust.

```plaintext
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 because Get may not deal with those inputs that do not satisfy the precondition: \( z \geq a \).

The robust specification of process Get is:

```plaintext
process Get(z : nat, a : nat) c : nat
ext wr mbox : nat
pre  true
post if z \geq a
   then c = a and mbox = z - a
   else c = 0 and mbox = \sim 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 output whereas a process allows many outputs.
- A function does not access external variables (like stores in CDFDs) whereas a process may do so.
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:

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

This means that function A will be defined in later development phase (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 function.
- an exit is necessary to ensure that any application of the function terminates.
Example: the factorial function is:

\[ n! = n \times (n - 1) \times (n - 2) \times \ldots \times 3 \times 2 \times 1 \]

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

```plaintext
function fact(n: nat) : nat == if n = 1 then n else n * fact(n - 1) end_function
```
The implicit specification of fact is:

```plaintext
function fact(n: nat) : nat
post if n = 1
    then fact = n
else fact = n * fact(n - 1)
end_function
```
Exercise 4

1. Answer the questions:
   a. what is a process?
   b. what is a data flow?
   c. what is the difference between active data flows and control data flows?
   d. what is a data store?
   e. what is the difference between data stores and data flows?
   f. what are the conditional structures for?
   g. what are the merging and separating structures for?
   h. what are the diverging structures for?
   i. what are the connecting structures for?
   j. what is a condition data flow diagram (CDFD)?
   k. what is a module for?
   l. what is the general structure of a module?
   m. what is an invariant?
   n. what is the general structure of a process?
   o. how to make a reference to the precondition or postcondition of a process?
   p. what is a function?
   q. what is the difference between a process and a function?
   r. what are the general formats of explicit and implicit specifications of a function?
   s. what is a recursive function, and what are the important points in writing recursive functions?
2. Define a calculator as a module. Assume that \textit{reg} denotes the register that is accessed by various operations. The operations include \textit{Add}, \textit{Subtract}, \textit{Multiply}, and \textit{Divide}. Each operation is modelled by a process.

3. Write a module defining 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.
**Figure 4**

- **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 necessary amount for purchasing a toy
- **warning**: a warning message for the shortage of the money in the money-box
4. Change the following compound expressions into equivalent classical predicate expressions.
   a. let $a = x + y$, $b = z + w$ in $a \times 2 \times b + b \times y \times w$
   b. if $x > 0$ then $a = x + 1$ else $a = x + 10$
   c. $a = \text{case } x \text{ of } 1, 2, 3 \rightarrow x + 1; 4, 5 \rightarrow x + 2; 6 \rightarrow x \times x; \text{default } \rightarrow x \text{ end}$

5. 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.