14. The software development process

The outline of this part:

- Software process using SOFL
- Requirements analysis
- Abstract design
- Evolution
- Detailed design
- Program
- Validation and verification
14.1 Software process using SOFL

The development of a complex software system needs a well-planned process to ensure its quality.
14.2 Requirements analysis

Requirements analysis is an activity to
- discover
- understand
- document the user's requirements.

It is usually the starting point of building a system, after the feasibility study of the system is completed with a positive conclusion.
SOFL advocates a three-step approach to building a formal design specification:

Informal $\rightarrow$ Semi-formal $\rightarrow$ Formal specifications

In this process informal and semi-formal specifications are used to document user requirements, while formal specification is used for design.
The reasons for the adoption of informal and semi-formal specifications for user requirements:

- Facilitate the communication between the user and the developer
- Strike a good balance between readability and preciseness of the requirements specification.
- Help the domain experts and project managers to review the user requirements specification for consistency and completeness at the highest level.
The informal specification

We must aim to achieve a well-organized informal specification that contains the following three aspects:

- the functions to be implemented
- the resources to be used
- the necessary constraints on both functions and resources

All these requirements can be organized based on the principle of building SOFL hierarchical modules (i.e., abstraction and decomposition).
Example: a hotel reservation system:

1. The required functions:
   - Reserve room
   - Cancel reservation
   - Change reservation
   - Check in
   - Check out
2. The resources of the hotel to be managed:

- Single rooms: 100
- Twin rooms: 50
- Double rooms: 50

3. The constraints reflecting the policy of the hotel:

- One customer can only reserve one room each time.
- Only can customers with reservations check in the hotel.
Decomposition (example):

1.1 Reserve room includes the functions:

- Check the vacancy
- Register the customer on the reservation list
- Issue the reservation number
The requirements hierarchy at the informal level:

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. Resources:

3. Constraints:
The semi-formal specification

The tasks and features of a semi-formal specification should include the following five aspects:

- Associate data resources, constraints, and functions in modules. That is, using modules to encapsulate the data resources, related data constraints, and the related operations that conform to the functional constraints.

- The specification is defined as a set of related modules. Each module defines either a function given in the informal specification or a derived function resulting from a further decomposition of an existing function.

- All the data used in the specification are defined with appropriate data types precisely in modules, but their constraints, which are usually given as invariants, are defined in an informal manner. In these definitions types are allowed to be defined as given types, if necessary.
The CDFD for each module is constructed, if it is not unnecessary.

All the processes and functions involved in a CDFD are defined in the associated module. Such definitions include the declaration of input data flows, output data flows, and external variables representing related stores. They also include the pre and postconditions, but the contents of the pre and postconditions are usually written in an informal manner.
Example: a semi-formal specification for the hotel reservation system:

```plaintext
module Reservation_system;

type
  FullName = given;

Customer = composed of
  name: FullName
  address: string
  tel: nat0
  pass_no: string
  reservation_no: nat0
end;

Room = composed of
  room_no: nat0
  room_type: {<Single>, <Twin>, <Double>}
  status: {<Reserved>, <Check In>, <Check Out>}
end;

ReservationList = map Customer to Room;

Rooms = set of Room;

ReservationRequest = given;

CancellationRequest = given;

ChangeRequest = given;
```
var
rlist: ReservationList;
rooms: Rooms;
inv
(1) every room in rooms is available for reservation;
(2) the status of every customer on the reservation list rlist must be either <Reserved> or <Check In>;
behav CDFD_1;
process Reserve (res-req: ReservationRequest)
    no-vacancy: string | res_no: nat0
ext rlist, rooms
pre the customer is not on the reservation list rlist.
post if there is vacancy
    then reserve a room for the customer and issue
    a reservation number.
    else produce a no_vacancy message.
end_process;
process Cancel(cancel_req: CancellationRequest) {
    confirmation: string
}

ext rlist

pre the customer is on the reservation list rlist

post (1) delete the customer's reservation from rlist.
    (2) release the reserved room resource by returning it to rooms so that it can be used for another reservation.

end_process;
process Change(chage-req: ChangeRequest)
  no-vacancy-mes: string | confirmation-mes: string

ext rlist, rooms
pre the customer is on the reservation list rlist
post (1) if the requested room is available, cancel the original reservation and make a new reservation.
  (2) if the requested room is not available, generate a no vacancy message.
end_process;
process Check_In(customer: Customer)  
  no_reservation_mes: string  
  room_no: nat0  
ext rlist  
post (1) if the customer has a reservation recorded on the reservation list rlist, check in the customer.  
(2) if the customer has no reservation, a message of refusing check in is issued.  
end_process;
process Check_Out(room_no: nat0)
    warning_mes: string |
    bill_receipt: string
ext rlist, rooms
post (1) if room_no is associated with a check in customer on the reservation list rlist, delete the customer's information and release the room.
    (1.1) calculate the cost and print out the bill.
    (1.2) receive payment and print out receipt.
(2) if a room to be checked out is associated with no customer on the reservation list rlist, generate a warning message.
end_process;
end_module;
14.3 Abstract design

Abstract design does two things:

1) **transforming the semi-formal requirements specification** into a formal design specification. This includes **defining all the necessary data types using non-given types** available in SOFL and **formalization of pre and postconditions** of all the processes and functions involved in the semi-formal specification, as well as all the **invariants**.

2) **Defining the architecture** of the entire system and functional definitions of its components.
Specifically speaking, apart from the criteria imposed to the semi-formal specification, a formal design specification is required to satisfy the following additional criteria:

- All the modules are integrated into a hierarchy of modules that are associated with a hierarchy of CDFDs.
- All the given types are defined precisely. In other words, no given types are allowed in the formal specification, for their values are not defined precisely.
- If applicable, the pre and postconditions of every process and function in modules are written in the SOFL language, not in any informal language.
- All the invariants are defined formally using SOFL.
Example: we take a top-down approach to design the architecture of the simplified Hotel Research System.

The top level CDFD:
module SYSTEM_Hotel_Reservation;

type
   FullName = given;
   Customer = composed of
      name: FullName
      address: string
      tel: nat0
      pass_no: string
      reservation_no: nat0
   end;

   Room = composed of
      room_no: nat0
      room_type: {<Single>, <Twin>, <Double>}
      status: {<Reserved>, <Check In>, <Check Out>}
   end;

   ReservationList = map Customer to Room;
   Rooms = set of Room;
   RoomNo = nat0;
var

ext #rlist: ReservationList;

ext #rooms: Rooms;

inv

forall[x: RoomNo] | 1 <= x <= 200;
forall[x: rooms] | x.status = <Check Out>;
forall[x: dom(rlist)] | rlist(x).status = <Reserved> or rlist(x).status = <Check In>;
behav CDFD_No1;
process Hote_Reservation(res_req: ReservationRequest | cancel_req: CancellationRequest | change_req: ChangeRequest | check_in_req: Customer | check_out_req: RoomNo)
    no_vacancy: string | res_no: nat | confirmation: string | no_vacancy_mes: string | confirmation_mes: string | no_reservation_mes: string | room_no: RoomNo | warning_mes: string | bill_receipt: string
ext rw rlist
    rw rooms
post (bound(res_req) => bound(no_vacancy) or bound(res_no)) and
    (bound(cancel_req) => bound(confirmation)) and
    (bound(change_req) => bound(no_vacancy_mes) or
        bound(confirmation_mes))
    and
    (bound(check_in_req) => bound(no_reservation_mes) or
        bound(room_no))
    and
    (bound(check_out_req) => bound(warning_mes) or bound(bill_receipt))
This process only specifies the relation between the input data flows and output data flows. In other words, it only specifies which input data flows are consumed to produce which output data flows. The details of how the input data flows are used to produce the output data flows are spelled out in its decomposition.

end_process;
end_module;
The process **Hotel_Reservation** is decomposed into the module **Hotel_Reservation_Decom** and its CDFD is:
module Hotel_Reservation_Decom / SYSTEM_Hotel_Reservation;

type
Date = nat0 * nat0 * nat0;
ReservationRequest = composed of
   name: Customer.FullName
   address: string
   tel: nat0
   period: Date * Date
   room_type: {<Single>, <Twin>, <Double>}
end;

CancellationRequest = composed of
   reservation_no: nat0
   name: Customer.FullName
end;

ChangeRequest = composed of
   reservation_no: nat0
   name: Customer.FullName
   room_type: {<Single>, <Twin>, <Double>}
end;
var
  ext #rlist: ReservationList;
  ext #rooms: Rooms;
behav CDFD_No2;

process Reserve(res_req: ReservationRequest)
  no_vacancy: string | res_no: nat
  ext wr rlist
    rw rooms
  pre not exists[c: dom(rlist)] | c.name = res_req.name and
    c.address = res_req.address
  decom Reserve_Decom
  comment
    This process reserves a room according to the request, if
    there is vacancy. Since this process has a decomposition,
    the postcondition is given as true, which is omitted in the
    specification.
end_process;
process Check_out(check_out_req: RoomNo)
    warning_mes: string |
    bill_receipt: string

ext rw rlist
    rw rooms
post (exists[c: dom(~rlist)] |
    (~rlist(c).room_no = check_out_req and
    rlist = domrb({c}, ~rlist) and
    rooms = union(~rooms,
        {modify(~rlist(c), status -> <Check Out>) } }
    and
    bill_receipt = Print_Bill_Receipt(c ))) or
    warning_mes = "The room number is wrong."
end_process;
function Print_Bill_Receipt(c: Customer): string == undefined end_function

/* this function returns a receipt that may be printed out on a printer. */
end_module;
The process Reserve is decomposed further into the module Reserve_Decom whose CDFD is as follows:
module Reserve_Decom / Hotel_Reservation_Decom;

var
  ext rlist: ReservationList;
  ext rooms: Rooms;

behav CDFD_No2;

process Check_Vacancy(res_req: ReservationRequest)
  no_vacancy: string |
  res_req: ReservationRequest

  ext rd rooms
  post (exists[r: rooms] | r.room_type = res_req.room_type) and
    res_req = ~res_req) or
  (not exists[r: rooms] | r.room_type = res_req.room_type) and
    no_vacancy = "No vacancy"

comment
  No specific precondition is required. If there exists a room whose type is
  the same as the required type of the reservation request, pass the
  reservation request ref_req to the output of this process. Otherwise,
  produce a "No vacancy" message as its output.

end_process;
process Make_Reservation(
    res_req: ReservationRequest)
cust: Customer

... 
end_process;

process Issue_Reservation_Number(cust: Customer)
    res_no: nat0

... 
end_process;

end_module;
14.4 Evolution

Transformations from informal to semi-formal, and then to formal specifications are in general an evolutionary process. An evolution of a specification can be one of the following three activities:

- Refinement
- Extension
- Modification
Refinement is an activity of improving a specification by resolving non-determinism.

Extension of a specification means the addition of new components to the specification. It can also mean the addition of new functionality to a process. A component can be a module, CDFD, process, or even a data type definition, and so on.

Modification of a specification is a change, either in syntax or semantics, without conforming to any formalized standard.
14. 5 Detailed design

The goal of the detailed design is twofold:

- transform the implicit specifications of processes, and functions, defined in modules, into explicit specifications in order that the algorithmic information provided by such explicit specifications serves as a foundation for implementation in a specific programming language.

- transform the structured abstract design specification into an object-oriented detailed design specification in order to achieve good quality of final implementation (e.g., encapsulation, information hiding, reusability, and maintainability).
It is worth mentioning that such a transformation needs to keep the hierarchy of CDFDs in the abstract design specification, thus we can leave as much freedom as possible to the programmer in deciding the strategy for the implementation of the specification. For example, the programmer can decide how to implement each CDFD in the specification based on the programming language he or she uses for implementation.
14.5.1 Transformation from implicit to explicit specifications

Only the lowest level processes (with no decompositions) need to be transformed, because each high level process is defined by its decomposition. The transformation from an implicit specification into an explicit specification is in fact a functional refinement.

Note that during a transformation from an implicit specification to an explicit specification, there may be a need to adjust or modify the definitions of some types given in the abstract design, but data refinement should not be emphasized because this issue will be addressed during the implementation of design specifications. Thus, it can help to avoid additional cost possibly caused by performing strict data refinement in specifications.
Definition 14.1 Let P and Q be two processes. Q is a refinement of P if and only if the following two conditions hold:

1. $\text{pre}_P \Rightarrow \text{pre}_Q$
2. $\text{pre}_P \text{ and } \text{post}_Q \Rightarrow \text{post}_P$
Example: let process $P$ be defined as:

```plaintext
process P()
  ext wr x: int
  pre x > 0
  post x > \sim x
end_process
```

This specification is refined into the following explicit specification:
process P()
ext wr x: int
pre x > 0
post x > ~x    /* usually the pre and
               postconditions are kept as a
               record of the specification
development */

explicit
  x := x + 1   /* implementing the postcondition */
end_process
Another definition of refinement that treats a process either in implicit or explicit style as a relation may be more straightforward in facilitating the application of conventional but practical verification techniques, such as testing and inspection.

**Definition 14.2** Let $P$ and $Q$ be the implicit and explicit specifications of a process, respectively. $Q$ is an refinement of $P$ if and only if the following condition holds:

$$\forall x : \text{dom}(P), \forall y : \text{rng}(Q) \mid \text{pre}_P(x) \land x Q y \Rightarrow \text{post}_P(x, y)$$

Where $\text{dom}(P)$ and $\text{rng}(Q)$ denote the domain of $P$ and the range of $Q$, respectively, and $x Q y$ means that $x$ and $y$ have relation $Q$.

Verification of this refinement obligation can be done by testing.
14.5.2 Transformation from structured to object-oriented specifications

We need to do the following for the transformation:

- **Convert a composite type into a class definition** in the way that the field variables of the composite type are defined as the attribute variables of the class and its methods are formed based on the operations on the values of the composite type in process specifications.

- **Define an appropriate class for each data store** occurring in a CDFD and **convert the store into an object of the class**.

- **Create new classes to meet the need of developing the abstract design specification** (e.g., developing the function of a process).

- **Transform the implicit specification of a process and function into an explicit specification in which objects, if any, are manipulated in the way that the principle of information hiding is not violated** (i.e., all the attributes of an object are accessed through its methods).
Example:

The process `Check_out` in the specification of Hotel Reservation System is transformed into the following explicit specification:
process Check_out(check_out_req: RoomNo)
    warning_mes: string | bill_receipt: string
ext rw rlist
    rw rooms
explicit
begin
    cus: Customer;
    cus := new Customer;
    cus := get({c: dom(rlist) | (rlist(c).room_no = check_out_req});
    if cus <> nil then
        begin
            rooms := union(rooms, {(rlist(cus)).setStatus(<Check Out>)});
            bill_receipt = Print_Bill_Receipt(cus);
            rlist = domdl({res}, rlist)
        end
    else
        warning_mes := "The room number is wrong."
    end
end_process;
class Customer;

type
    FullName = string * string * string;
/*first name, middle name, and family name*/
var
    name: FullName
    address: string
    tel: nat0
    pass_no: string
    reservation_no: nat0
method Init()
    ...
end_class;
class Room;
  var
    room_no: nat0;
    room_type: {<Single>, <Twin>, <Double>};
    status: {<Reserved>, <Check In>, <Check Out>};

method Init()
  ...
method setStatus(st: {<Reserved>, <Check In>, <Check Out>})
  explicit
    status := st;
end_method;

...
14.6 Program

Program is an implementation of the detailed design in a specific programming language. It is desirable to ensure that transforming a detailed design, an explicit specification, into a program that satisfies the specification.

This transformation needs to deal with the refinement of abstract data types defined in the design specification into concrete data types available in the programming language.
In general four level transformations are necessary:

1. Transformation of the abstract data types.
2. Transformation of explicit specifications of processes, methods, and functions.
3. Transformation of modules.
4. Transformation of classes.
Data refinement:

Let **abs** and **con** denote the abstract and concrete data types, respectively, then **con** is a refinement of **abs** if and only if there exists a retrieve function, say **Retr**, such that:

\[
\forall a: \text{abs} \ \exists c: \text{con} \ |
\text{Retr}(c) = a
\]

That is, every value in the abstract type must be implemented by at least one value in the concrete type.
Transformation from SOFL to Java:

- A module in the design is transformed to a class in the program.
- A process of the module in the design is implemented as a method in the corresponding class in the program.
- The CDFD of the module in the design is implemented as an independent method in the class in the program.
- A class in the design is transformed to a class in the program.
14.7 Validation and verification

(1) **Validation**: the validation of a specification is an activity to ensure that the written specification reflects the user's requirements accurately and completely. It can be done by means of specification testing and/or rigorous reviews.

(2) **Verification**: the verification of a specification aims to ensure that the specification is internally consistent, satisfiable, and really met by their implementations (or programs). For example, the precondition of a process must not violate the related type invariant.
Exercise 14

1. Give an example to explain the difference between evolution and refinement of processes.

2. Construct a formal design specification of library system by taking the three steps: informal, semi-formal, and formal specification. The system is required to provide the services: Borrow, Return, and Search. Each of these services should be implemented by a process. The process Borrow registers the data of the borrowed book; Return removes the registered information about the borrowed book; and Search provides the requested information of the wanted books, if they are available.

3. Refine the implicit specifications of all the three processes in the library system into explicit specifications.