11. Hierarchical CDFDs and Modules

The motivation for building hierarchical CDFDs:

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

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

This process can be decomposed into the CDFD on the next slide:
module Check_Password_decom / SYSTEM_ATM;
var ext account_file: set of Account;
behav CDFD_2; /* Assume the CDFD in Figure 2 is numbered 2. */
process Init()
end_process;

process Confirm_Accout(card_id: nat0, pass: nat0)
    account: Account | pr_meg: string
...
end_process;

process Transfer_Accout(sel: bool, account: Account)
    account1: Account | account2: Account
...
end_process;
end_module;
Definition If process $A$ is decomposed into a CDFD, we call the CDFD the decomposition of process $A$ and process $A$ the high level process of the CDFD.

Decompositions can be carried out until all the lowest level processes are simple enough to be formally defined. As a result, a hierarchical CDFDs will be constructed.
The associated module hierarchy of this CDFD hierarchy is outlined as follows:

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

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

process A2;

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

process A4;
end_module;
```
module A1_decom;
...
var ext s1: Type1;
behav CDFD_2;
process Init;
process A11;
process A12;
process A13;
end_module;
module A3_decom;
...
behav CDFD_3;
process Init;
process A31;
process A32;
process A33
decom: A33_decom; /*Module A33_decom is associated with the decomposition of A33. */
end_process;
end_module;
module A33_decom;
...
var s2: Type1;
behav CDFD_4;
process Init;
process A331;
process A332;
process A333;
end_module.
Handling stores in decomposition

Generally speaking, a store accessed by a high level process must also be drawn in the decomposition of the process, and must be accessed in the same way, probably by several processes.
Definition An external store of a CDFD is a store that is accessed by a high level process of the entire specification.

There are two kinds of external stores:
(1) a store local to a high level CDFD. Usually it is declared after the keyword `ext`.
(2) existing external store that is global to all the CDFD in the hierarchy. Usually such stores are declared with the sharp mark, such as `ext #file: int`

Definition A local store of a CDFD is a store that is introduced for the first time in the CDFD.
Input and output data flows

**Definition** an input data flow of a starting node of a CDFD is treated as an input data flow of the CDFD. An output data flow of a terminating node is treated as an output data flow of the CDFD.

**Definition** a process and its decomposition are **structurally consistent** if the three conditions hold:

1. all the input data flows and the output data flows of the decomposition are the input data flows and the output data flows of the process, respectively.
2. all the input and output data flows of the process appear in its decomposition.
3. any store connected to the process (i.e., the process may read or write the store) is accessed in the same manner by the process and the child processes in its decomposition.
Example:

High level process

The decomposition of process W

Rules for consistent access to a store:
(1) \( \text{rd } s \) at the high level CDFD \( \rightarrow \) \( \text{rd } s \) at the decomposition.
(2) \( \text{wr } s \) at the high level CDFD \( \rightarrow \) \( \text{rd} \) and \( \text{wr } s \) in the decomposition.
The correctness of decomposition

The decomposition of a high level process can be perceived as an implementation of the process, in the sense that the decomposition is intended to provide the functionality of the high level process in a more "executable" style.

Definition Let $G$ be a CDFD. Then a data flow path of $G$ is a sequence of data flow groups traversed by an execution of $G$ from a starting node to some of the terminating nodes of $G$. 

Example: the possible data flow paths of the decomposition of process $W$ are:

1. $[x_1, \{[d_1, y_1], [d_2, y_2]\}]$
2. $[x_2, \{[d_1, y_1], [d_2, y_2]\}]$
Let $A$ be a high level process:

```
process A (x1: Ti_1 | x2: Ti_2) y1: To_1, y2: To_2
ext wr s: Ts
pre pre_A
post post_A
end_process
```

Let $G$ denote the decomposition of $A$. Then, the correctness of $G$ with respect to $A$ is defined as follows:
Definition If $A$ and $G$ are structurally consistent, and the following condition holds:

\[
\text{forall}[x_1: Ti_1, ~s: Ts] \mid \text{pre}_A(x_1, x_2, ~s) \Rightarrow \text{post}_A(x_1, x_2, G(x_1), ~s, s)
\]

or

\[
\text{forall}[x_2: Ti_2, ~s: Ts] \mid \text{pre}_A(x_1, x_2, ~s) \Rightarrow \text{post}_A(x_1, x_2, G(x_2), ~s, s)
\]

we say that $G$ satisfies $A$, or $G$ is correct with respect to $A$. We use $G(x_1)$ (or $G(x_2)$) to denote the output data flows generated by $G$ (it may be represented by a set of data flows like $\{y_1, y_2\}$), through an execution taking $x_1$ (or $x_2$) as its input.
11.2 Scope

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

To give a formal definition of the scope rule, we need the following definitions.
**Definition** Let process $A$ be defined in module $M1$. If $A$ is decomposed into a CDFD associated with module $M2$, we say that process $A$ is decomposed into module $M2$.

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

**Definition** Let $A_1, A_2, ..., A_n$ be a sequence of modules where $n > 1$. If $A_1$ is the parent module of $A_2$, and $A_2$ is the parent module of $A_3$, ..., and $A_{n-1}$ is the parent module of $A_n$, then we call $A_1$ ancestor module of $A_n$ and $A_n$ descendant module of $A_1$. 
Definition If module $A$ is neither ancestor module nor descendant module of module $B$, $A$ and $B$ are called relative modules.

Example:

(1) $M_2$ is the parent module of $M_{12}$.
(2) $M_2$ is an ancestor module of $M_{122}$.
(3) $M_{11}$ and $M_{13}$ are relative modules. $M_{13}$ and $M_{121}$ are relative modules.
Scope rules:

Let $M_c$ be a type or constant identifier declared in module $M_1$. Then, the scope of the effectiveness of this declaration is $M_1$ and its all descendant modules.

Let $M_c$ be declared in both module $M_1$ and its ancestor module $M$. Then, $M_c$ declared in $M_1$ has higher priority in its scope than the same $M_c$ declared in $M$.

Let $M_f$ be a function defined in $M_1$. Then, the scope of $M_f$ is $M_1$.

Let $M_c$ be a type, constant identifier, or function declared in module $M_1$. Then, if $M_c$ is used in its relative module $M_2$, it must be used in the form:

$$M_1.M_c$$
Example: suppose type `Person` is defined in module `M1`. Then, we can use `Person` to declare a store variable `s` in a relative module `M2` as follows:

```
s: M1.Person;
```

The following is a list of references of function `fact` used in module `M2`, pre- and postconditions of process `A`, and constant identifier `Age`, which are all assumed to have been defined and declared in module `M1`.
- `M1.fact(5)`
- `M1.pre-A`
- `M1.post-A`
- `M1.Age`
11.3. Approaches to constructing specifications

The outline of this part:

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

Two strategies can be taken in the top-down approach: the CDFD-module-first strategy and the CDFD-hierarchy-first strategy.
(1) The CDFD-module-first strategy

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

1. If the relation between the input and output data flows of a process cannot be expressed without further information, the decomposition of this process should be considered.

2. If the behavior of a process involves a sequence of actions, this process needs to be decomposed.

3. If the postcondition of a process is too complex to be written in a concise manner, it may need decomposition.
(2) The CDFD-hierarchy-first strategy

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

(b) Bottom-up for data flows and stores
11.3.2 The middle-out approach

Constructing a specification by the middle-out approach usually starts with the building of the CDFDs and modules modeling the functions that are most familiar to the developer and crucial to the system. Then, the system is built by decomposing some of the introduced processes and synthesizing some of those processes to form high level processes.
When synthesizing CDFDs into high level processes, we can follow the following guidelines:

- If there are more than two input data flows to different starting processes of a CDFD, the CDFD needs to be abstracted into a high level process that defines precisely the relationship among those input data flows.

- If two processes in a CDFD access the same store in both reading and writing manner, this CDFD needs to be considered for abstraction to define that concurrent execution of the two processes is impossible (e.g., processes A1_1 and A1_2 in CDFD_2).

- If two CDFDs have relations in terms of data flows, they need to be abstracted into high level processes and the connections between these processes need to be formed in the high level CDFD (e.g., process A1_3 in CDFD_2 and process A2_1 in CDFD_3 share the same data flow a2).
11.3.3 Comparison of the approaches

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

(1) Advantages:

- It is effective and intuitive in providing sub-goals or sub-tasks to support the current goal or task, and in developing ideas with little information (abstraction) into ideas with more information (decomposition).

- It provides a good global view of data flows and stores that may be used across CDFDs at different levels, thus the consistency in using data flows and stores can be well managed during the decomposition of high level processes.
(2) Weakness:

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

(1) Advantages:

- It may be more **effective and natural than the top-down approach**, because it always starts with modeling the most familiar and crucial functions.

- It also takes a **flexible way to utilize the top-down and the bottom-up approaches**. Taking which approach usually stems from natural demands during the construction of the entire specification.
(2) Weakness:
The developer may not be easy to take a global view of the specification in the early stage, thus data flows, stores, and processes created in different CDFDs may overlap or defined inconsistently.
3. How to use the top-down and the middle-out approach?

- Use the **middle-out approach** for requirements analysis and requirements specification constructions, especially for the semi-formal ones, because the most familiar and important functional requirements are often focused in the early stage of requirements analysis.

- Use the **top-down approach** for design, because the designer usually has a fair understanding of the functional requirements after studying the semi-formal requirements specification, and needs to take a global view in structuring the entire system.
Class exercise 10

1. Answer the questions:
   a. what is a hierarchy of CDFDs?
   b. what is a hierarchy of modules?
   c. what is the relation between module hierarchy and CDFD hierarchy?
   d. what is the relation between a CDFD and its high level process in a CDFD hierarchy?
   e. what does it mean by a CDFD being consistent with its high level process?
   f. what does it mean by saying that module M1 is the ancestor module of M2?
   g. what does it mean by saying that modules M1 and M2 are relative modules?
   h. what is the scope of a variable, type identifier, constant identifier, invariant, function, and a process?

2. Explain whether the CDFD in Figure 5 is structurally consistent with its high level process $W$? Is the CDFD possible to be correct with respect to process $W$? If so, explain why.
High level process

The decomposition of process W

Figure 5
11.4 The SOFL Three-Step Approach to Constructing Formal Specifications

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

The SOFL three-step approach is a solution!
Three-step formal specification:

- Informal specification
- Semi-formal specification
- Formal specification

Requirements analysis

Abstract design
Tasks for informal specification: Capture desired functions, necessary data resources, and constraints on both functions and data resources.
Tasks for semi-formal specification: (1) **Grouping** related functions, data resources, and constraints into SOFL modules. (2) **Defining** necessary data types and variables. (3) **Defining** the function of each process using pre- and post-conditions at informal level.
Tasks for formal specification: (1) Describe system architecture using hierarchical CDFDs (condition data flow diagram). (2) Formalize the specification of each process in pre- and post-conditions.
1. Informal specification (requirements analysis)

(1) Functions:

- Receive commands from the customer
- Confirm the customer’s card id and password
- Withdraw
- Inquire about the customer’s account balance
(2) **Data resources:**

- Customers’ account information

(3) **Constraints:**

- A customer can only access to his or her own account.
- No customer is allowed to borrow money from the bank.
1. The functions of the system:
   (1) F1
   (2) F2
   (3) F3
   (4) F4

1.1 F1
   (1) F11
   (2) F12
   (3) F13

1.2 F2
   (1) F21
   (2) F22

2. Recourses:

3. Constraints:
2. Semi-formal specification (requirements analysis):

module SYSTEM_ATM

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

var
  account_file: set of Account;

inv
  Account balance must be greater than or equal to zero.

behav CDFD_No1;
...

process Withdraw(amount: real, account1: Account)
e_msg: string | cash: real

ext wr account_file

pre account1 is a member of account_file

post if amount is not greater than the balance of account1
then supply cash with the same amount as amount, and
reduce the amount from the balance of the account.
else output an appropriate error message.

end_process;

...

end_module
3. Formal specification (abstract design)

The top level CDFD:
module SYSTEM_ATM

type

Account = composed of

    account_no: nat

    password: nat

    balance: real

end

var

account_file: set of Account;

inv

    forall[x: Account] | x.balance >= 0;

behav CDFD_No1;

...
process Withdraw(amount: real, account1: Account)
    e_msg: string | cash: real
ext wr account_file
pre account1 inset account_file
post if amount <= account1.balance
    then
        cash = amount and
        let Newacc =
            modify(account1, balance -> account1.balance – amount)
        in
            account_file = union(diff(~account_file, {account1}), {Newacc})
    else
        e_msg = "The amount is over the limit. Reenter your amount."
comment
...