19. Transformation from designs to programs

The outline of this part:

- The principle of transformation
- Transformation of data types
- Transformation of modules and classes, including process and method specifications
- Transformation of CDFD
- Important points
19.1 The principle of transformation

Transformation from specification S to program P should aims to achieve, at least, the following qualities:

(1) Correctness: program P satisfies S.
(2) Efficiency: less memory and high speed when P executes.
(3) Maintainability: program P is easy to be evolved (extension, modification).
To reach the goal of achieving the above qualities, we should take the following issues into account in the transformation:

(1) How to properly transform data types
(2) How to properly transform process specifications
(3) How to properly map CDFDs defining the architecture of the specification to the program structure
19.2 Transformation of data types

1. The principle

Let $T_a$ denote a type in specification $S$ and $T_c$ denote a type in program $P$. Then the condition for transforming $T_a$ into $T_c$ is:

$$\text{forall}[a: T_a] \text{ exists}[c: T_c] \mid \text{Retr}(c) = a$$

where \text{Retr} is known as retrieve function:

$$\text{Retr}: T_c \rightarrow T_a$$
This condition requires that when abstract type $T_a$ is transformed into concrete type $T_c$, all the elements of $T_a$ be represented by the elements of $T_c$, that is, $T_c$ should contain sufficient elements to represent all the elements of $T_a$, because this will eliminate the possibility of inappropriate data structures causing the program using $T_c$ not to satisfy the required functions in the specification.
2. The programming language
   Java!

3. Mapping between the built-in types in SOFL and types in Java (only a guideline!)
<table>
<thead>
<tr>
<th>type in SOFL</th>
<th>type in Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>nat0</td>
<td>int</td>
</tr>
<tr>
<td>nat</td>
<td>int</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>real</td>
<td>double</td>
</tr>
<tr>
<td>bool</td>
<td>boolean</td>
</tr>
<tr>
<td>char</td>
<td>char</td>
</tr>
<tr>
<td>Enumeration</td>
<td>Enumeration or array of String</td>
</tr>
<tr>
<td>set type</td>
<td>Set, array, vector, or file</td>
</tr>
<tr>
<td>seq type</td>
<td>List, array, vector, or file</td>
</tr>
<tr>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>map type</td>
<td>Map, array, vector, or file</td>
</tr>
<tr>
<td>composite type</td>
<td>class</td>
</tr>
<tr>
<td>product type</td>
<td>class</td>
</tr>
<tr>
<td>union type</td>
<td>Object class</td>
</tr>
</tbody>
</table>
19.3 Transformation of modules and classes

Definition: A module or class in a specification is called source module or source class.
Definition: A class in a Java program is called target class.

Overall transformation strategy

- Source module
- Source class
- Target class
Specific transformation guidelines

- Transform the module name to the class name.
- Transform a constant declaration to a constant declaration (using the keyword final prior to the constant variable in the declaration).
- Transform a type declaration to either a corresponding basic type or a proper target class.
- Transform a store variable declared in the var part to an instance variable (or field) of the class.
- Transform a source process (a process of the source module) to a target method (a method of the target class).
- Transform a function, if any, defined in the module to a target method that does not change the state of the target class.
- When transforming a process or a function, make sure that the invariants defined in the inv part of the module are not violated.
- Transform the CDFD of the module to a target method of the class in which all the related methods are integrated accordingly through their invocations.
transformation

module m

store s

instance variable s

data flows z, u, v, t, w

attributes z, u, v, t, w

process Init

constructor m

CDFD

method G

process A

method A

process B

method B

process C

method C

function □

method □

}
Examples

A source module:
module Student_Management / Faculty_System;
  const
    PI = 3.14159;
  type
    Students = set of Student;
  var
    students: Students;
  inv
    forall[x: Student] | x.total_credit <= 135;
  behav CDFD_e1;
process Init()
    post students = { }
end_process;

process Find(x: Student) x2: Student | x1: Student
    ext rd students
    post if (exists[y: students] | x.id = y.id)
        then x2 = x
        else x1 = x
end_process;
process Add(x1: Student)
  ext wr students
  post students = union(~students, {x1})
end_process;
process Update(x2: Student) confirm: bool
  ext wr students
  post if x2 inset students
    then students = diff(~students, {x2}) and
       res = true
    else students = union(~students, {x2}) and
       res = false
  end_process;
end_module;
The above source module is transformed into the following target class:

```java
public class StudentManagement {
    static final double PI = 3.14159;
    private Set students;

    public Student_Management() {
        students = new HashSet();
    }

    public boolean Find(Student x) {
        if (students.contains(x))
            return true;
        else
            return false;
    }
}
```
public void Add(Student x)
{
    students.add(x);
}

public boolean Update(Student x2)
{
    //create an array objs containing all the elements
    //of the set students.
    Student objs[] = students.toArray();
    for (int i = 0; i < objs.length; i++)
        if (objs[i].getID() == x2.getID())
        {
            //Remove the identified element from the set students
            students.remove(objs[i]);
            break;
        }
    //Add the element x2 to the set students
    students.add(x2);
}
//A method derived from the CDFD of the module in the specification

public boolean CDFD(Student x)
{
    if (Find(x))
    {
        Add(x);
    }
    else
    {
        return Update(x);
    }
}
} //the end of the target class
A source class:

```plaintext
class Student / Person;
var
  id: string;
  name: string;
  total_credit: nat0;
method Init()
  post id = 0 and name = "" and total_credit = 0
end_method;

method Set_Name(name1: string)
  ext wr name    post name = name1
end_method;
end_class;
```
The source class Student is transformed into the following target class:

```java
public class Student extends Person {
    int id;
    String name;
    int totalCredit;
    public Student()
    {
        id = 0;
        name = "";
        totalCredit = 0;
    }
    public void SetName(String name1)
    {
        name = name1;
    }
}
```
public int getID()  
{      return id;      }  

public String getName()  
{      return name;      }  

public int getTotalCredit()  
{      return totalCredit;      }  

} //the end of the target class
Transformation of processes

There are two kinds of processes:

(1) single-port process (a process with both one input and output port)
(2) multiple-port process

Transformation strategy

Single-port process

Multiple-port process

Method

Multiple methods
(1) Transformation of single-port process
Let A be a single-port process:

```pascal
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
pre pre_A
post post_A
end_process
```

Then we transform process A into method A in the target class Transformation1:
class Transformation1 {
    To_1 y_1;
    To_2 y_2;
    ...
    To_m y_m;
    ...
    public void A(Ti_1 x_1, Ti_2 x_2, ..., Ti_n x_n) {
        if (pre_A)
        {
            Tran(post_A)  //transformation of post_A
        }
        ...
        // other parts of the class
    }
} // end of the class
If process A is given in an explicit form:

\[
\text{process } A(x_1: \text{T}_1, x_2: \text{T}_2, \ldots, x_n: \text{T}_n) \\
y_1: \text{T}_1, y_2: \text{T}_2, \ldots, y_m: \text{T}_m \\
\text{pre pre}_A \\
\text{explicit} \\
S \\
\text{end_process}
\]

Then we transform it into method A of the target class Transformation2:
class Transformation2 {
    To_1 y_1;
    To_2 y_2;
    ...
    To_m y_m;
    ...
    public void A(Ti_1 x_1, Ti_2 x_2, ..., Ti_n x_n) {
        if (pre_A) {
            Tran(S)  // transformation of statement S
        }
        ...
    }
}
(2) Transformation of multiple-port process

Let B be a multiple-port process in module M:

```plaintext
process B(x_1: Ti_1 | x_2: Ti_2)
  y_1: To_1 | y_2: To_2
pre pre_B
post post_B
end_process
```

and assume that output y_1 is generated based on input x_1 and y_2 is based on x_2. Then we transform B into two methods B1 and B2 in the target class AnotherTransformation that is derived by transforming module M:
class AnotherTransformation {
    // corresponding to the source module containing process B in its CDFD.
    
    ... public To_1 B1(Ti_1 x_1)
    {  if (pre_B(x_1))
        Tran(post_B(y_1))  //post_B(y_1) is part of post_B
    }
    public To_2 B2(Ti_2 x_2)
    {  if (pre_B(x_2))
        Tran(post_B(y_2))  //post_B(y_2) is part of post_B
    }

    ...
}
Another more general transformation method is to generate a class, say ProcessClass, which contains the definitions of methods B1 and B2. And then in the method derived from the CDFD of module M in the class AnotherTransformation, the two methods B1 and B2 are invoked properly. For example, we can get the following transformation:
class ProcessClass {

    ...

    public To_1 B1(Ti_1 x_1)
    {
        if (pre_B(x_1))
        {
            Tran(post_B(y_1))
        }
    }

    public To_2 B2(Ti_2 x_2)
    {
        if (pre_B(x_2))
        {
            Tran(post_B(y_2))
        }
    }

    ...

}
class AnotherTransformation {
    ...
    public CDFD(...) {
        ProcessClass processOBJ =
            new ProcessClass();
        ...
        processOBJ.B1(x_1); //invoke method B1 of //the object
        processOBJ
        ...
        processOBJ.B2(x2); //invoke method B2 of //the object
        processOBJ
        ...
    }
    ...
}
19.4 Transformation of CDFD

We discuss the guidelines for transformation of CDFD through examples of all the possible structures of a CDFD.

A CDFD may contain the structures:

- Sequential structure
- Conditional structure
- Nondeterministic structure
- Broadcasting structure
- Parallel structure
**Definition:** Let $T_c$ be a function from the set of CDFDs to the set of algorithms written in Java:

$$T_c : S_{cdfd} \mapsto A_{java}$$

Thus, when applying $T_c$ to a specific CDFD, say $S$, it will yield an algorithm in Java, that is,

$$T_c(S)$$

denotes an algorithm.

Now the most important issue is how to define $T_c$ for every possible CDFD structure.
Guideline 1. Let $S$ denote the sequential structure in Figure 1. Then

$$T_{\{c\}}(S) == TY_1 y_1;$$
$$TY_2 y;$$
$$A1(x, s);$$
//take $x$ and $s$ to produce
//$y_1$ and update $s$, which is
//implemented in the definition
//of $A1$
$$A2(y_1, s);$$
//take $y_1$ and $s$ to produce
//$y$ and update $s$, which is
//implemented in the definition
//of $A2$

Where we use == to mean "is defined as" and = to mean the assignment operator in Java.
Guideline 2. Let $S$ denote the conditional structure in Figure 2.

Then

\[ T_c(S) = TY_1 s; \]

//store $s$ is treated as a global variable

... if ($P(y)$)
{  $y_1 = y$;
  $w_1 = B(y_1)$;
} else
{  $y_2 = y$;
  $w_2 = C(y_2)$;
}
Guideline 3. Let $S$ denote the single condition structure in Figure 3.

Then

$T_c(S) == TY_1 s$;

//store $s$ is declared
//as a global
//variable

... 

if ($P(y)$)

{ $y_1 = y$;
  $w = B(y_1, s);$}
Guideline 4. Let $S$ denote the CDFD containing a multiple condition structure given in Figure 4.

Then

$T\{c\}(S) ==$

$TY_1 s;$
//declaring s as a global
//variable

... 

if (P1(y))
{  y1 = y;
   w1= B(y1,s); }
else if (P2(y))
{  y2 = y;
   w2 = C(y2, s);} 
else if (!(P1(y) || P2(y)))
{  y3 = y;
   w3 = D(y3,s);}
Guideline 5. Let $S$ denote the CDFD of Figure 5 that involves a nondeterministic structure.

Then

$$T_{\{c\}}(S) = TY_1 s;$$

//treating store $s$ as a global variable

... 

if $(P(x))$

{ $x_1 = x$;  
  $y_1 = B(x_1,s)$; }

else if $(P_1(x))$

{ $x_2 = x$;  
  $y_2 = C(x_2,s)$; }

else if $(P_2(x))$

{ $x_3 = x$;  
  $y_3 = D(x_3,s)$; }

Figure 5
Guideline 6. Let $S$ denote the broadcasting structure in Figure 6. Then

$T_{c}(S) == TY_1 s$;

//s is declared as a global //variable

... 

x1 = x;
x2 = x;
x3 = x;
y1 = B(x1,s);
y2 = C(x2,s);
y3 = D(x3,s);
**Guideline 7.** Let $S$ denote the iteration structure in Figure 7. Then
\[
T_c(S) ==
TY_1 s;
//declare s as a global
//variable
... 
  y = E(x);
  while (P(y))
    {
      y = E1(y);
    }
  y1 = E2(x, y);
\]
Guideline 8. Let \( S \) denote the parallel structure in Figure 8, where the executions of process A and B are independent of each other. Then

\[
T_{c}(S) ==
TY_1 s;
//store s is treated as a
//global variable
... 
\]

\[
y_1 = A(x_1,s);
y_2 = B(x_2,s);
\]
19.6 Important points

- The guidelines given above show only one possibility of transformation from CDFDs to programs. There may be more efficient transformation approaches.

- A method for automatically transforming any CDFD into a program is still an open problem.

- It is impossible to automatically transform a process specification into a program in general, but it is possible to do so only for an executable subset of process specifications.

Let’s consider the following examples.
process P(x, y: real) z, d: real
post x ** 5 + z ** 8 > y ** 9 + d ** 3 and
    z * d < x * y
end_process

It is difficult to provide a translation system that can convert this specification into an equivalent and executable program.

However, it is possible for the following specification:
process P(x, y: real) z, d: real
post z = x + y and d = x * y
end_process

The important features of this specification in relation to the executability are:
(1) Output variables z and d are defined independently of each other.
(2) Each output variable is defined based only on input variables.
(3) The definition of each output variable is deterministic (e.g., z = x + y, not something like z > x + y and z < x * y).
Exercise 19

1. Give another way of transforming a source module and class that differs from that of the one given in this lecture notes.

2. Give a transformation of process A in the way that the target method A produces an error message when the precondition is not satisfied by the inputs, where process A is assumed to have the following format:

```
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
pre pre_A
post post_A
end_process
```
3. Give a different transformation of process B with two input and output ports from the one given in this lecture notes. The format of process B is as follows:

\[
\text{process } B(x_1: Ti_1 \mid x_2: Ti_2) \\
\quad y_1: To_1 \mid y_2: To_2 \\
\text{pre } \text{pre}_B \\
\text{post } \text{post}_B \\
\text{end_process}
\]
4. Suppose process A is decomposed into a CDFD. Give a transformation of A that utilizes the CDFD in defining the body of the target method A.