Lecture 3

User Defined Classes

Class Definition

A class definition is an abstract data type. It has two main sections.

- abstract section
  - declare abstract variables
  - class invariants
- interface section used to declare public:
  - functions,
  - schemas,
  - operators
  - constructors

Class Template

```
class <name> ^=
abstract
    // Add variable, invariant and
    //private method declarations
interface
        // public methods here
   build{}
        post ?;
end;
```

Constructor

build{}

- The name of the constructor method
- Multiple constructors with different parameter lists are allowed

post ?;

- Describes the condition that must be satisfied after the constructor is invoked.

Class Counter

```
class Counter ^=
   abstract
      var count : int;
   interface
      function getCount : int
         ^= count;
       function count;
   build{}
       post count! = 0;
end;
```

Constructor

- The postcondition in the constructor `build` states that the attribute `count` changes such that its final value is `0`.
- Also possible to write the constructor with an argument

```
build{x : int}
post count! = x;
```

Symbol !

- The symbol ! after a name indicates that a change of value occurs
- It can also be used as a prefix to the name of an abstract variable in the constructor.

```
build{!count}
```

Creating a Counter

- To create a Counter use the expression

```
Counter{}
```

or

```
Counter{5}
```

- Outside the class declaration add the definition

```
const c1 ^= Counter{};
```

Functions

- Functions
  - return information about the abstract variables in a class
  - Cannot change the values of abstract variables

- Abstract variables can be re-declared as functions in the interface section. They provide read-only access.
  - Use this approach as opposed to method `getCount`

Schemas

A schema is a method that can change one or more abstract variables in a class.

A schema is declared as follows:

```
schema !<name>
  post ?;
schema !<name>(param_list)
  post ?;
```

Schemas

For example,

```
schema !inc
  post count! = count + 1;

schema !inc(x:int)
  post count! = count + x;
```

Schemas

post ?;

Describes the condition that must be satisfied after the schema is invoked.

Given an instance c1 of type Counter to invoke the schema inc use:

c1!inc

c1!inc(5)

Class Invariants

A class invariant is a constraint that must be true for every instance of the class. It is a global constraint on the values of abstract variables.

In the Counter class we might want to impose a lower bound.

```
abstract
    var count : int;
    invariant count >= 0;
```

Class Invariants

The invariant is a global constraint and will have consequences when writing methods that change the values of abstract variables.

The verifier has to be able to prove that all change methods satisfy the global invariant.

```
class Counter ^=
abstract
    var count : int;
    invariant count >= 0;
interface
    schema !inc
            post count! = count + 1;
    schema !inc(x : int)
            post count! = count + x;
    build{}
           post count! = 0;
    build {x : int}
            post count! = x;
end;
```

Verifier Report

Counter.pd

(14,12): Information! Confirmed: Class invariant satisfied

(17,11): Warning! Unable to prove: Class invariant satisfied

..cannot prove: 0 <= self'.count.

(20,12): Information! Confirmed: Class invariant satisfied

(24,14): Information! Confirmed: Class invariant satisfied

(27,11): Warning! Unable to prove: Class invariant satisfied

Generating verification output files ... 0 seconds

0% of capacity used

0 errors, 2 warnings found.

PD: Job completed but warnings were detected!

Verifier Report

The causes of the warnings were:

```
schema !inc(x : int)
    post count! = count + x;

build {x : int}
    post count! = x;
```

Preconditions

The problems can be solved by adding a precondition that restricts the values of the parameter x.

The keyword `pre` followed by one or more comma separated Boolean expressions is used to denote preconditions.

Preconditions

Preconditions represent conditions that must be satisfied when a

- schema,
- constructor, or
- function

is invoked

Adding preconditions to Counter

```
schema !inc(x : int)
      pre x >= 0
      post count! = count + x;

build {x : int}
      pre x >= 0
      post count! = x;
```

```
class Counter ^=
abstract
    var count : int;
    invariant count >= 0;
interface
    schema !inc
        post count! = count + 1;
    schema !inc(x : int)
        pre x >= 0
        post count! = count + x;
    function getCount : int
        ^= count;
    redefine function toString : string
        ^= "Counter: " ++ count.toString;
     build{}
        post count! = 0;
     build {x : int}
        pre x >= 0
        post count! = x;
end;
```

toString function

Inherited by all classes from root class anything

Must be redefined in user-defined classes to use it

Example,

```
redefine function toString : string
    ^= "Counter: " ++ count.toString;
```

Combining Postconditions

Postconditions can be combined using:

- conjunction(&)

  swap x, y <==> x! = y & y! = x

  note: satisfied in parallel

- schema composition(sequence)

  denoted by key word then

- conditional expression

  allows choice in postcondition

Primed Abstract Variables

A primed variable refers to the final value of a variable after a change event. Primed variables may be used in postconditions and in post assertions. For example,

count' refers to the final value of count after dec or inc schemas,

or

x! = e & y = x' <==> x! = e & y = e

Meaning of Postconditions in Schemas

The meaning of the post condition is defined by

`change` var-list `satisfy condition`

where each var in var-list must appear primed in condition

Meaning of Postconditions in Schemas

By this definition

count! = count + 1

<==>

change count satisfy count' = count + 1

Equality

Two instances of the Counter class `c1`, `c2` can be compared for equality of content using the equals operator (`=`).

`c1 = c2` is defined over equality of attributes for all instances of the class

Comparison

To impose an ordering on instances of the Counter class we must specify a meaning for the comparison operator ~~

```
operator ~~(other)
    ^= count ~~ other.count;
```

Note: ~~ between instances of the `int` class is already defined.

Comparison Test

```
const c1 : Counter
 ^= Counter{5};
const c2: Counter
 ^= Counter{5};
const c3 : Counter
 ^= Counter{4};
property assert c1 ~~ c2 = same@rank;
property assert c3~~c2 = below@rank;
```

Post Assertions

A post assertion is a boolean expression that must be satisfied after a change operation on the state of the class.

After !inc the value of count must be greater than 0.

This constraint can be inserted as a post assertion on schema inc as follows:

```
schema !inc
  post count! = count + 1
  assert count' > 0;
```

Post Assertions

A post assertion for !dec might be:

```
schema !dec
    pre count > 0
    post count! = count -1
    assert count' >= 0;
```

Properties of a class

Properties are used to verify expected behaviours of all instances of a class.

Properties can be expressed by using the

`property assert` clause

or by writing

`ghost` schemas

Note: A `ghost` schema is one for which no code is generated.

Properties of Counter class

The value of the abstract variable count after !inc is greater than 0

```
property
  assert (self after it!inc).count > 0;
```

Properties of Counter class

The value of count is unchanged when !inc is followed by !dec

```
ghost schema !addDec
    post !inc then !dec
    assert count' = count;
```

Properties of Counter class

!inc followed by !inc gives a final value of count greater than 1

```
ghost schema !incInc
    post !inc then !inc
    assert count' > 1;
```

Complete Counter class

```
class Counter ^=
abstract
   var count : int;
   invariant count >= 0;
interface
   schema !inc
     post count! = count + 1
     assert count' > 0;
   schema !inc(x : int)
    pre x >= 0
    post count! = count + x
    assert count' >= 0;
```

Complete Counter class

```
schema !dec
  pre count > 0
  post count! = count -1
  assert count' >= 0;

function count;

redefine function toString : string
     ^= "Counter: " ++ count.toString;
operator ~~(other)
 ^= count ~~ other.count;
```

Complete Counter class

```
build{}
      post count! = 0;
build {x : int}
 pre x >= 0
 post count! = x;

 property
  assert (self after it!inc).count > 0;
```

Complete Counter class

```
  ghost schema !addDec
    post !inc then !dec
    assert count' = count;
  ghost schema !incInc
    post !inc then !inc
    assert count' > 1;
end;
```

Cursor Control

A computer terminal has fixed dimensions given by the number of lines and columns in the display. It also has a cursor that indicates the current position of the typeface in the display. The movement of this cursor is restricted to the limits of the dimensions of the terminal. The movement of the cursor is controlled by the following list of public methods:

home

set cursor to top left hand corner

return

generates a carriage return and linefeed when cursor is not currently at the bottom of the terminal: otherwise a carriage return

Cursor Control

down

generates a linefeed when cursor not on bottom row; otherwise cursor wraps to top of terminal

up

cursor moves up one row when not at top row; otherwise wraps to bottom row

left

cursor moves left one position when not at leftmost column; otherwise wraps to end of line

right

cursor moves right one position when not at rightmost column; otherwise wraps to start of line

Terminal class

```
class Terminal ^=
abstract
   var
    line : nat,
    column : nat,
    numCols : nat,
    numLines : nat;
   invariant
    (1 <= line <=numLines) & (1 <= column <= numCols);
```

Terminal class

Constructor

```
build{lines : nat, cols : nat}
    pre cols > 1 & lines >= 1
    post  line! = 1, column! = 1,
          numCols! = cols,
          numLines! = lines;
```

Interface section

```
schema !home
    post line! = 1, column! = 1;
```

Interface section

```
schema !return
    post
        ([ line < numLines ]:
            line! = line + 1,
         [ ]:
            line! = line
        )
        &
        (column! = 1);
```

Interface section

```
schema !down
    post
        ([ line < numLines ]:
              line! = line + 1,
         [ ]:
             line! = 1
        )
        &
        (column! = column);
```

Interface section

```
schema !left
    post
        ([column > 1]:
            column! = column - 1,
         []:
            column! = numCols
        )
        &
        (line! = line);
```

Interface section

```
schema !right
    post
       ([column < numCols]:
          column! = column+1,
        []:
          column! = 1
       )
       &
       (line! = line);
```

Property

!left followed by !right leaves the cursor position unchanged.

```
ghost schema !leftRight
  post !left then !right
  assert column' = column & line' = line;
```