Assignment 3: Testing

ETH Zurich

Hand-out: 7 June 2006
Due: 20 June 2006

1 Code coverage

Goal

Understand the different measures of code coverage provided in the lecture.

Tasks

Consider the following routine which uses Euclid's algorithm to compute the greatest common divisor of its arguments:

```
gcd(a, b: INTEGER): INTEGER is
      -- Greatest common divisor of 'a' and 'b'.
   local
      i, j: INTEGER
   do
      from
         i := a
         j := b
      until
         i = j
      loop
         if i > j then
            i := i - j
         else
            j := j - i
         end
      end
      Result := i
   end
```

1. Write a test case which achieves 100% statement coverage for this routine and passes.
2. Write a test case (or several if necessary) that achieve(s) path coverage, considering that the presence of a loop opens 2 paths: one is covered if the loop is not executed at all and the other if it is executed one or more times.

3. Write a test case for this routine that fails (hint: a test case whose execution does not terminate is also considered to fail).

4. Fix the bug(s) that your test case has just revealed by adding appropriate contracts to the routine. Now run again all the test cases that you developed so far, to make sure that the bug(s) has/have been eliminated. (The test cases that were triggering the bug should now be filtered out by the routine's contract).

5. Provide an example of a piece of code for which path coverage cannot be achieved. If you are using any loops, state the definition of path coverage that you are using. Explain why the paths in your code cannot be covered 100%.

Solution

1.
```
print (gcd (6, 4).out)
```

2.
```
print (gcd (6, 4).out + "%N")
print (gcd (4, 4).out)
```

3.
```
print (gcd (−6, 4).out + "%N")
```

4. Add the precondition:
```
require
  a_greater_than_0: a > 0
  b_greater_than_0: b > 0
```

5.
```
a, b: INTEGER
 ...
if a > 0 then
  b := b + 1
end
print (a.out)
if a > 0 then
  b := b + 1
end
```

   There are 4 paths in this piece of code, but only 2 of them can be executed (one for any $a <= 0$ and the other for any $a > 0$).

2 Mutation testing and contract-based testing

Goal

Understand how mutation works and what it tries to achieve. Add a postcondition to a routine so that we can perform contract-based testing on it.

Tasks

2.1 Mutation operators

The examples of mutation operators provided in the lecture are: operator replacement, replacing a variable by another one, replacing a variable used in an expression by a constant. There are also mutation operators specific for object-oriented programming, such as:

- Creation procedure call with child class type: changes the dynamic type with which an object is created.
- Deletion of routine redefinition: deletes the redefinition of a routine in a sub-class.

Give examples of other mutation operators specific for OOP. Remember that the code must still compile after the mutation operator has been applied and that a tool must be able to apply the operator automatically on any piece of code where it's applicable.

Solution

Here are some examples:

- Access modifier (visibility) change - changes the visibility level of features
- Deleting a call to *Precursor*
- Replacing a call to a modifier by a call to another modifier
- Reference assignment and content assignment replacement

2.2 Creating mutants

Consider the following routine, which should return the index of the first occurrence of the maximum element in a non-empty list of integers:

```
index_of_max (list: LIST [INTEGER]): INTEGER is
        -- Index of the first occurrence of the maximum element in list 'list'
    require
        list_not_void: list /= Void
        list_not_empty: not list.empty
    local
        max: INTEGER
    do
        from
            list.start
            Result := 1
            max := list.item
        until
            list.after
        loop
            if list.item >= max then
        Result := list.index
        max := list.item
      end
      list . forth
    end
  end
```

A tester has written the following test case for the above routine (located in the same class, so you can assume that the code compiles):

```
test_index_of_max is
    -- Test routine 'index_of_max'.
  local
    l: LIST [INTEGER]
  do
    create {ARRAYED_LIST [INTEGER]} l.make (0)
    l.extend (10)
    l.extend (20)
    l.extend (30)
    if index_of_max (l) = 3 then
      print ("test passes%N")
    else
      print ("test fails%N")
    end
  end
```

Routine *index_of_max* does not fulfill its specification as stated informally in its header comment, but the given test case does not expose the problem.

Provide a non-equivalent mutant of routine *index_of_max* that is not killed by the given test case. Explain why the mutant is not equivalent to the original through an example.

Solution

The following mutant is obtained by replacing the >= operator by > (and hence actually fixes the bug present in the implementation). The given test case will not distinguish its output from that of the original routine. However, there exist inputs (such as the list [1, 3, 3]) for which the original routine and the mutant will have different outputs, hence they are not equivalent.

```
index_of_max (list: LIST [INTEGER]): INTEGER is
    -- Index of the first occurrence of the maximum element in list 'list'.
    -- 0 if 'list' is empty.
  require
    list_not_void: list /= Void
    list_not_empty: not list.empty
  local
    max: INTEGER
  do
    from
      list . start
      Result := 1
      max := list.item
    until
      list . after
    loop
      if  list .item > max then
        Result := list.index
        max := list.item
      end
      list . forth
    end
  end
```

2.3 Contract-based testing

Performing contract-based testing of routine *index_of_max* would not bring us any useful information about its correctness, because the routine does not have a proper postcondition. Add a postcondition that states that the result is indeed the index of the maximum element in the list. Why is it difficult to state in this postcondition that the index found is that of the *first* occurrence of the maximum element? Try to come up with a solution.

Solution

We add the following postcondition to routine *index_of_max*:

```
ensure
  list . for_all  (agent greater_than ( list . i_th  (Result), ?))
```

where

```
greater_than  (max, elem: INTEGER): BOOLEAN is
    -- Is 'max' >= 'elem'?
  do
    Result := max >= elem
  end
```