Verslag OGO 2.2 voorjaar 2009

Reflectie specificatie & implementatie

Technische informatica, TU/e

Eindhoven, February 13, 2009

Groep 2
Etienne van Delden, 0618959
Edin Dudojevic, 0608206
Jeroen Habraken, 0586866
Dion Jansen, 0590077
Stef Louwers, 0590864
Anson van Rooij, 0596312

Contents

1 Discussion specification group A

1.1 General

- The use of for example A and A' before- and after-states of variable A is not part of Hoare-tripling. It's part of the specification language $\mathbb{Z}$ and is meaningless in the way the group used it in their specification.
- As stated before, the group used A and A' to denote before- and after-states, but they used it wrong. Instead of assigning a value to A' to denote an after-state they assigned a value to A in the post-conditions.

1.2 Add

- $(\neg n \lessdot A)$ was used in the post-condition, where $n \in \mathbb{Z}$. What is a non-integer?

1.3 Sort

- As stated before A and A' were used wrong. The post-condition of this procedure required the ordering of A, but since A' is the after-state A' is supposed to be ordered instead. Now we could order the array A and return A' which is, according to the precondition, the same as the unordered A in the pre-condition.

1.4 Median

- No A' used in post-condition, so no after-state?
- Function will return the wrong element from the array A. When the number of elements in A is even the function will return the element just above the middle element.

2 Discussion implementation group A

2.1 Code

See algorithm 1 and 2.

2.2 Comments

No comments. Due to our small fault in the specification, this implementation is correct $\ddot\smile$.

3 Discussion specification group B

- There is one crucial error in the post of the specification.
  $(((\#B \neq 0) \Rightarrow \textbf{return}\ B(\lfloor(\#B)/2\rfloor)) \vee ((\#B = 0) \Rightarrow \textbf{return}\ -1))$ is true, when **return** -1 is true (**false** $\Rightarrow$ **true** $=$ **true**). Concluding: returning -1 satisfies the *entire* post condition.
- The specification is given in exactly 1 Hoare Triple. The given problem is relatively trivial and can be given in 1 Hoare Triple, it is preferable to split it up into sub problems.

Algorithm 1 Framework for group A

```
program mdm;

uses
  SysUtils;

type
  ainteger = array of integer;

procedure ReadFile(var f: Text; var B: ainteger);
var
  n, i: integer;
begin
  i := 0;
  while not Eof(f) do begin
    while not Eoln(f) do begin
      read(f, n);
      setlength(B,i+2);
      B[i] := n;
      i := i + 1;
    end;
  end;
end;

var
  A: ainteger;
  dm: Integer;
  inFile: Text;
  outFile: Text;
begin
  AssignFile(inFile, 'invoer.txt');
  AssignFile(outFile, 'uitvoer.txt');
  Rewrite(outFile);
  Reset(inFile);

  ReadFile(inFile, A);
  dm := middelste_derde_macht(A);

  write(dm);
  write(outFile, dm);

  CloseFile(inFile);
  CloseFile(outFile);
end.
```

Algorithm 2 Implementation for group A

```
function middelste_derde_macht(A: Array of Integer): Integer;
begin
    middelste_derde_macht := -1
end;
```

4 Discussion implementation group B

4.1 Code

See algorithm 3 and 4.

Algorithm 3 Framework for group B

```
program Input (Output);

uses
    Process;

var
    F: text;
    c: char;
    i: integer;
    mid: integer;

begin
    if (ParamCount <> 1) then begin
        WriteLn('Please specify an input file');
        Exit
    end;

    Assign (F, ParamStr (1));
    Reset (F);

    Initialize ();
    while not EOF (F) do begin
        Read (F, i);
        Read (F, c);
        Add (i);
    end;
    Sort ();
    mid := Median ();
    if (mid = nil) then begin
        WriteLn('No third power in the input file');
    end else begin
        WriteLn(mid);
    end;

    Close (F);
end.
```

4.2 Comments

No comments.

Algorithm 4 Implementation for group B

```
unit implementation;

interface

uses
  Math;

var
  A: array of integer;

implementation

procedure Add(var n: Integer);
var
  i : Integer;
begin
  i := 0;
  while ( Power(i,3) < abs(n) ) do i := i + 1;
  if (Power(i,3) = abs(n)) then begin
    i := 0;
    while ( i < length(A)) and ( A[i] <> n ) do i := i + 1;
    if (i = length(A)) then begin
      setlength(A, length(A)+1);
      A[length(A)-1] := n;
    end;
  end;
end;

procedure Sort;
  var i, j, key: Integer;
begin
  for j:= 1 to length(A)-1 do
  begin
    key := A[j];
    i := j - 1;
    while  (i>=0) and (A[i] > key) do begin
      A[i+1] := A[i];
      i := i - 1;
    end;
    A[i+1] := key;
  end;
end;

function Median(): Integer;
begin
  if (length(A)=0) then result := -1
  else result := A[floor(length(A)/2)];
end;

end.
```

5 Reflection

We learned two big lessons while working on this project in the past few weeks, they were:

1. There are two types of specification: one can specify as little as possible, giving only pre- and postconditions, or one can specify as much as possible, dividing the program up into multiple smaller functions and procedures, and giving specifications for these, or dividing them up even further if that is deemed necessary.
   Each of these approaches was used by one of our subteams, and as we have noticed, both approaches have advantages and drawbacks.

   A big advantage of specifying as little as possible is of course that it is less work. It is also more elegant, leaving more freedom for the implementing team. Specifying as little as possible will probably tend to become unwieldy as the programs to be specified become larger, but for a program as small as the one to be specified in assignment one, it worked just fine.

   Specifying as much as possible, essentially meaning telling the implementation team more or less exactly what to do, is more feasible when it comes to larger projects. However, a specification team using this approach will probably spend a lot of time solving problems which are not their responsibility.

   Based on our experiences solving the first assignment we are probably going to use a mix of these approaches, staying general when providing specifics would be difficult, and being specific when staying general would be risky.

2. Small errors can lead to big problems. Whatever level of details we are working on, and whatever way we specify our program, mistakes can happen, and, given enough complexity, will happen.

   Our subteams specified the program of the first assignment in completely different ways, yet both made very small mistakes which turned into fatal problems.

   This means that we will have to be watchful for errors at all moments, and also that we will have to find some "insurance" against oversights. We are probably going to use scenarios to provide this protection: by specifying not only rules but also specific sequences of actions which should be possible.