

Cleanroom Verification

- In Cleanroom, constructed programs can be checked by a parser for syntax errors, **but may not be executed by the developer**
 - No debugging \Rightarrow cheap and predictable process
- Verification is performed by a team review driven by a set of *verification conditions*
 - Questions to ask about the program code
 - Specific questions are asked about each occurrence of different kinds of program statements
- Productivity
 - 3x–5x improvement in verification over debugging

Formal Inspections

- Although program proving is always an option, it involves intensive work requiring mathematical sophistication
- An alternative, used by Cleanroom software engineering, is to structure a team code inspection in terms of *program functions* and *verification conditions* and then undertake a semi-formal review confirming that all verification conditions are satisfied

Functional Verification Steps

1. Program is specified by pre and post conditions
2. *Prime program decomposition*
 - Parse program control flow into nested single entry/exit constructs (SESEs)
3. Proceeding top down, determine the program function for all SESEs
 - *Program function*: Description of the function of a prime program
4. Define *verification conditions* for each program point
 - Things to check for each SESE
5. Inspect, answering all verification conditions

Program Function

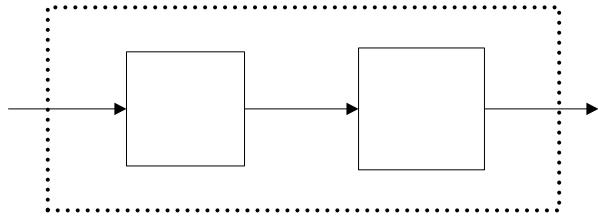
- Conditions under which the prime program can legally execute (preconditions)
- Expression of the effect of program execution on the state of the system (postconditions)
- Expressed in terms of the program's input arguments, return value, instance variables, global variables, and side effects on the environment (disk writes, printing, etc.) but not referring to local program variables

Sort Program Function

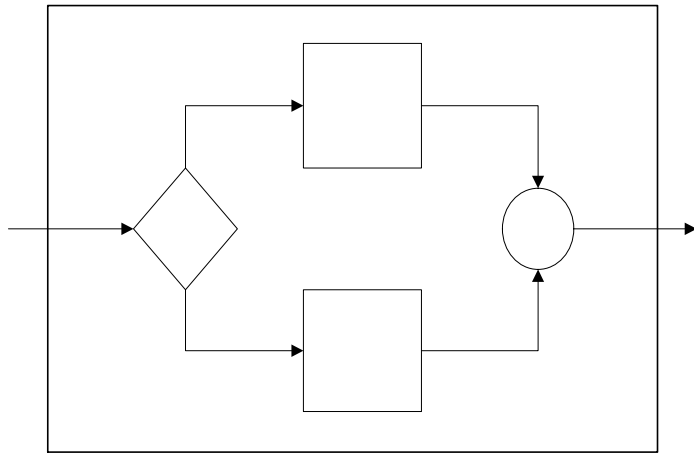
```
context IntVector::sort()  
  -- assumes that IntVector is  
  -- represented with an OCL sequence  
post:  
  this@pre->asBag() = this->asBag()  
  and  
  forAll(i : Integer |  
    1 <= i and  
    i < this->size() and  
    this->at(i) <= this->at(i + 1))
```

Program Parse

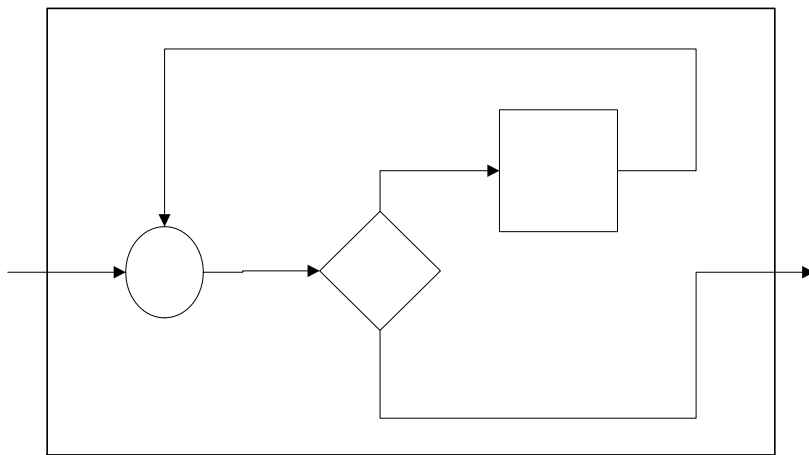
- Modern programming languages support the concept of nested blocks
 - A block is normally enclosed in braces or keyword pairs (`begin-end`)
- In structured programs (programs without `GOTO` statements), the nesting is always well formed
 - That is, there is only ever one way for control to enter the block and one way to exit. That is, they have the property of being *single-entry, single exit (SESE)*
 - Programs with `GOTO`s can be handled using special methods
- The process of determining the SESEs for a program involves parsing its control flow graph



Sequential Composition



Conditional Composition



Iterative Composition

Typical SESEs

Composition of SESEs

- Each SESE can be thought of as being itself a small program with its own program function
- The overall program function is the logical composition of the program functions of its constituent SESEs
- The lowest level SESE is the single assignment statement

Verification Conditions

- If we were proving a program correct, we would construct the proof by composing the proofs of each of the SESEs
- Instead of a proof, Cleanroom uses an informal review that examines each program statements to determine its logical validity
- In particular, each type of statement has a set of questions that should be asked about it every time that it occurs in the program

Verification Conditions - 2

- Imagine a program p with a program function f
- p is composed of smaller programs (SESEs) $q, r, s \dots$ each with their own program functions $f_q, f_r, f_s \dots$
- In general, we have to devise verification conditions $VC_q, VC_r, VC_s \dots$ such that demonstrating that the verification conditions all hold demonstrates that f also holds
- This will depend on how the statements are composed
- There are three ways of composing SESEs
 - Sequentially, conditionally and iteratively

Sequence

- The simplest control structure is a sequence of two other statements or control structures
- There is one verification condition per sequence:
 - Do the constituent statements together accomplish the sequence's goal?
 - That is, if you come across the following situation
[f] do g; h enddo
 - Then generate the following verification condition
Does g followed by h do f?
- This idea can readily be extended to three or more constituent statements

Sequential Composition

1. Is the post assertion of the sequence equivalent to the logical composition of the first part followed by second part?

Conditional

- An `if-then-else` has two arms
 - Does each arm acting by itself accomplish the control structure's post condition, assuming the control structure's precondition and that the tested condition is true (or false)?
 - That is, if you come across the following situation

```
[f] if p then g else h endif
```
 - Then generate the following verification conditions
 - When `p` is true does `g` do `f`?
 - When `p` is false does `h` do `f`?
- `If-then` is treated as `if-then-else` with a null arm. `Case` is like a multi-armed `if`

Conditional

2. Does taking the `true` branch imply the post assertion?

- The predicate of the conditional can be assumed to be `true`

3. Does taking the `false` branch imply the post assertion?

- The predicate can be assumed to be `false`

Iteration

- There are three questions to ask about an iterative construct such as a while loop:
 - Does it terminate in all circumstances?
 - Does it accomplish its purpose when it does not execute?
 - Does it accomplish its purpose when its body is executed followed by its own execution?
 - That is, if you come across the following situation
`[f] while p do g endo`
 - Then generate the following verification conditions
 - Is termination guaranteed?
 - When `p` is true does `g` followed by `f` do `f`?
 - When `p` is false does doing nothing guarantee `f`?
- `for` loops and `repeat` loops can be defined in terms of `while` loops

Iteration

4. Does the loop terminate?

5. If the predicate is `false`, does the post assertion follow from the pre assertion?

6. If the predicate is `true`, is the post assertion of the loop equivalent to the post assertion of the body followed by the post assertion of the loop?

- Recursive!

Implications

- As teams become more experienced in Cleanroom, they begin to write their programs more directly
 - This typically results in very small program segments with few control structures each
- Example: 3300 lines \Rightarrow 600 control structures, 1000 correctness conditions

Results Of Independent Empirical Evaluation

- 15 3-person teams; 10 of them used Cleanroom
- 6/10 delivered 91% of functionality
- Requirements better met and less failures
- More comments, less dense control flow
- Better adherence to schedule
- Developers expressed satisfaction with process

Example

- Consider the following program segment
- What are its pre and post conditions?

```
SUM(b[0 .. n] : int)
  returns t : int

  int b[0 : n];

  i := 1;

  t := b[0];

  while i <= n do
    t := t + b[i];
    i := i + 1
  end;

end;
```

Program Function

- Signature:

`SUM (b [0 .. n] : int)`

`returns t : int`

- Precondition: $n \geq 0$

– "n is non-negative"

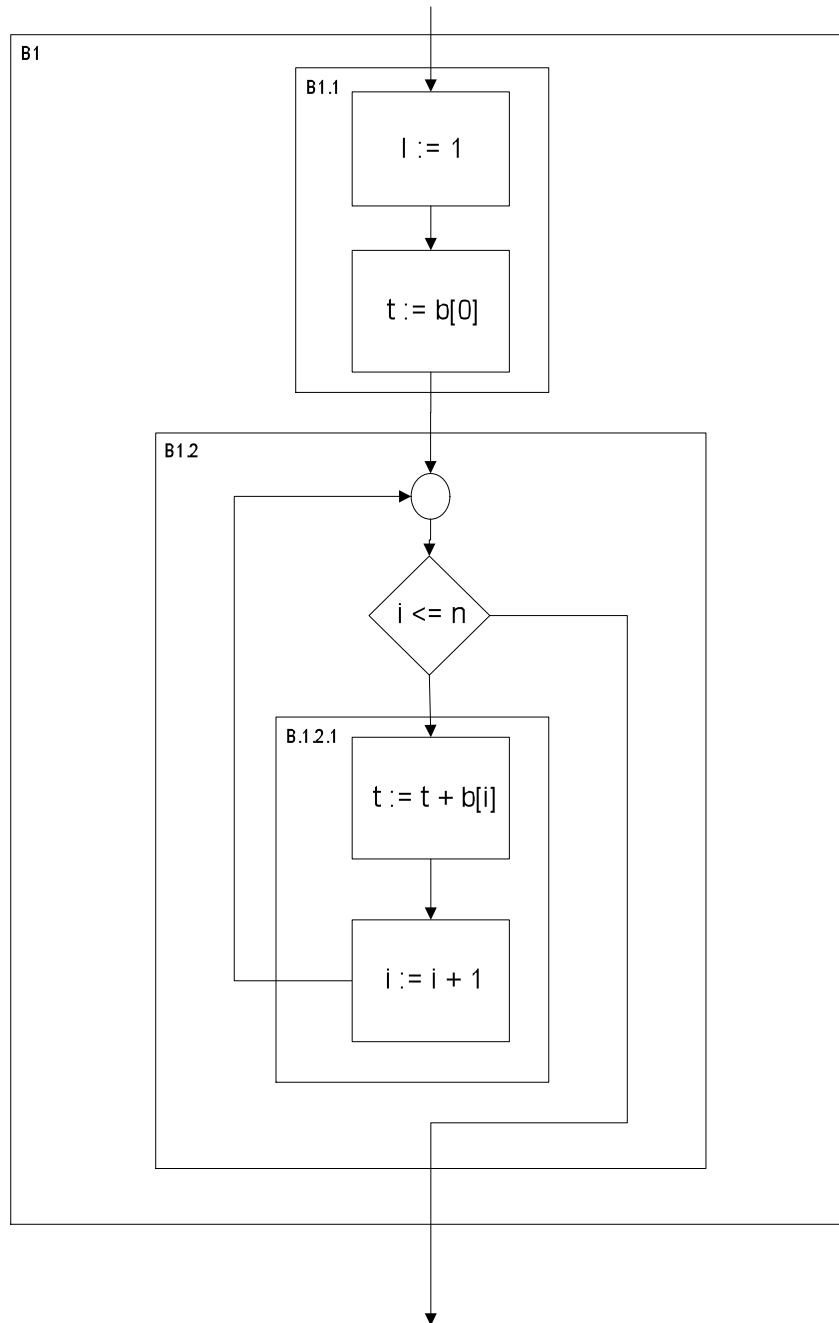
- Postcondition: $t = \sum_{k=0}^n b[k]$

– "t is the sum of the elements of b"

Program Parse

```
Seq(  
  Seq(  
    i := 1;  
    t := b[0];  
  )  
  Loop(  
    Seq(  
      t := t + b[i];  
      i := i + 1  
    )  
  )  
)
```

Program Decomposition



Program Function

<----- { $n \geq 0$ }

```
int b[0 : n];
```

```
i := 1;
```

```
t := b[0];
```

```
while i <= n do
```

```
    t := t + b[i];
```

```
    i := i + 1;
```

```
end;
```

<----- { $t = \sum_{k=0}^n b[k]$ }

Informal Specification

- B1: Sum the contents of the vector b of length n
 - B1.1: Initialize program variables
 - Initialize loop counter
 - Initialize partial sum
 - B1.2: Loop through tail of vector adding elements
 - B1.2.1: Process each element
 - Increase partial sum
 - Increase loop counter

Formal Specification

```
<----- {n >= 0}
int b[0 : n]; i := 1; t := b[0];

<----- {n >= 0 ∧ i = 1 ∧ t = b[0]}
while i <= n do
  <----- {1 <= i <= n ∧ t =  $\sum_{k=0}^{i-1} b[k]$ }
  t := t + b[i];
  <----- {1 <= i <= n ∧ t = t' + b[i] =
            $\sum_{k=0}^{i-1} b[k] + b[i] = \sum_{k=0}^i b[k]$ }
  i := i + 1
  <----- {1 <= i - 1 <= n ∧ t =  $\sum_{k=0}^{i-1} b[k]$ }
end;
<----- {i > n ∧ t =  $\sum_{k=0}^n b[k]$ }
```

Verification Conditions

- B1: Sum the contents of the vector b of length n
 1. Does initializing the program variables followed by looping through the remaining vector elements compute the sum of all elements?
- B1.1: Initialize program variables
 2. Does initializing the loop counter followed by initializing the partial sum successfully initialize the program variables?
- Initialize loop counter
 3. Does setting i to 1 successfully initialize the loop counter?
- Initialize partial sum
 4. Does setting t to $b[0]$ compute the initial partial sum?

Specification - 2

- B1.2: Loop through tail of vector adding elements
 5. Does the loop terminate in all circumstances?
 6. Does the loop add up all the elements of an empty tail?
 7. After the loop has successfully added some number of elements in the tail, does the next iteration add in the next element?
- B1.2.1: Process each element
- Increase partial sum
 8. Does adding the current element to the partial sum increase the partial sum appropriately?
- Increase loop counter
 9. Does adding 1 to i successfully complete the iteration?

Exercise

```
procedure search(id, i: integer;
                emp(1..n): array of integer)
  bot, top, mid: integer
  i := 0; bot := 1; top := n;
  while bot <= top & i = 0 do
    mid := (bot + top) / 2
    if emp(mid) = id then
      i := mid
    elseif emp(mid) < id then
      bot := mid + 1
    else // emp(mid) > id
      top := mid - 1
    endif
  enddo
endprocedure
```