FRAMA-C: INTRODUCTION

Dr.S.Padmavathi

CSE, Amrita School of Engineering

Coimbatore

Maximum

- Specify and prove the following program:
- // returns the maximum of x and y
- int max (int x, int y) {
- if (x >=y)
  - return x ;
- return y ;
- }

Specification

- The following program is proved. Do you see any error?
- /*@ ensures \result >= x && \result >= y;
- */
- int max (int x, int y) {
- if (x >=y)
  - return x ;
- return y ;
- }

Our specification is incomplete
Should say that the returned value is one of the arguments

Corrected Specification

- This is the completely specified program:
- /*@ ensures \result >= x && \result >= y;
- ensures \result == x || \result == y;
- assigns \nothing ;
- */
- int max (int x, int y) {
- if (x >=y)
  - return x ;
- return y ;
- }

Example pointers

- Specify and prove the following program:
- // returns the maximum of *p and *q
- int max_ptr (int *p, int *q) {
- if (*p >= *q)
  - return *p ;
- return *q ;
- }

Specification pointer

- Explain the proof failure for the following program:
- /*@ ensures \result >= *p && \result >= *q;
- ensures \result == *p || \result == *q;
- */
- int max_ptr (int *p, int *q) {
- if (*p >= *q)
  - return *p ;
- return *q ;
- }

Nothing ensures that pointers p, q are valid
It must be ensured either by the function, or by its precondition

Safety warnings: invalid memory accesses

- An invalid pointer or array access may result in a segmentation fault or memory corruption.
- They ensure that each pointer (array) access has a valid offset (index)
- If the function assumes that an input pointer is valid, it must be stated in its precondition, e.g.
  - \valid(p) for one pointer p
  - \valid(p+0..2) for a range of offsets p, p+1, p+2

Example: specification pointer

- The following function is given a contract to specify that it increments the value pointed to by the pointer given as argument.
- /*@ **requires \valid**(p);
- @ **assigns** *p;
- @ **ensures** *p <==> **\old**(*p) + 1;
- @*/
- void incrstar(int *p);

- The contract means that the function must be called with a pointer p that points to a safely allocated memory location (\valid built-in predicate).
- It does not modify any memory location but the one pointed to by p.
- Finally, the ensures clause specifies that the value *p is incremented by one.

Specification pointer

- The following program is proved. Do you see any error?
- /*@ requires \valid (p) && \valid (q);
- ensures \result >= *p && \result >= *q;
- ensures \result == *p || \result == *q;
- */
- int max_ptr (int *p, int *q) {
- if (*p >= *q)
  - return *p ;
- return *q ;
- }

Incorrect implementation

- This is a wrong implementation that is also proved. Why?
- /*@ requires \valid (p) && \valid (q);
- ensures \result >= *p && \result >= *q;
- ensures \result == *p || \result == *q;
- */
- int max_ptr (int *p, int *q) {
- *p = 0;
- *q = 0;
  - return 0 ;
- }

Our specification is incomplete
Should say that the function cannot modify *p and *q

Frame rules

- The clause assigns v1, v2, ... , vN;
  - Part of the post condition
  - Specifies which (non local) variables can be modified by the function
  - No need to specify local variable modifications in the post condition
    - a function is allowed to change local variables
    - a postcondition cannot talk about them anyway, they do not exist after the function call
  - Avoids to state that for any unchanged global variable v, we have ensures \old(v) == v
  - Avoids to forget one of them: explicit permission is required
  - If nothing can be modified, specify assigns \nothing

Corrected specification

- This is the completely specied program:
- /*@ requires \valid (p) && \valid (q);
- ensures \result >= *p && \result >= *q;
- ensures \result == *p || \result == *q;
- assigns \nothing ;
- */
- int max_ptr (int *p, int *q) {
- if (*p >= *q)
  - return *p ;
- return *q ;
- }

Contracts and function calls

- Function calls are handled as follows:
  - Suppose function g contains a call to a function f
  - Suppose we try to prove the caller g
  - Before the call to f in g, the precondition of f must be ensured by g
    - VCs is generated to prove that the precondition of f is respected
  - After the call to f in g, the post condition of f is supposed to be true
    - the post condition of f is assumed in the proof below
    - modular verification: the code of f is not checked at this point
    - only a contract and a declaration of the callee f are required
- Pre/post of the caller and of the callee have dual roles in the caller's proof
  - Pre of the caller is supposed, Post of the caller must be ensured
  - Pre of the callee must be ensured, Post of the callee is supposed

Example multiple functions

- Specify and prove the function max_abs
- int abs (int x);
- int max (int x, int y);
- // returns maximum of absolute values of x and y
- int max_abs (int x, int y) {
- x=abs(x);
- y=abs(y);
- return max (x,y);
- }

Specify what is wrong

```
#include <limits.h>
/*@ requires    x > INT_MIN ;
ensures ( x >= 0 ==> \result == x ) && ( x < 0 ==> \result == -x ) ;
assigns  \nothing ; */
int   abs ( int   x ) ;
/*@ ensures   \result >= x && \result >= y ;
ensures   \result == x   || \result == y ;
assigns   \nothing ; */
int max ( int  x , int  y ) ;
/*@ ensures  \result >= x &&  \result >= -x && \result >= y &&  \result >= -y ;
ensures   \result == x  || \result == -x ||  \result == y || \result == -y ;
assigns  \nothing ; */
int max abs ( int  x , int  y ) {
x=abs ( x ) ;
y=abs ( y ) ;
return max ( x , y ) ;
}
```

Specify what is wrong

```
#include <limits.h>
/*@ requires   x > INT_MIN ;
ensures ( x >= 0 ==> \result == x ) && ( x < 0 ==> \result == -x ) ;
assigns  \nothing ; */
int   abs ( int   x ) ;
/*@ ensures   \result >= x && \result >= y ;
assigns   \nothing ; */
int max ( int  x , int  y ) ;
/*@ requires   x > INT_MIN ;
requires   y > INT_MIN ;
ensures  \result >= x &&  \result >= -x && \result >= y &&  \result >= -y ;
ensures   \result == x  || \result == -x ||  \result == y || \result == -y ;
assigns  \nothing ; */
int max abs ( int  x , int  y ) {
x=abs ( x ) ;
y=abs ( y ) ;
return max ( x , y ) ;
}
```

Correct specification

```
#include <limits.h>
/*@ requires    x > INT_MIN ;
ensures ( x >= 0 ==> \result == x ) && ( x < 0 ==> \result == -x ) ;
assigns  \nothing ; */
int   abs ( int   x ) ;
/*@ ensures   \result >= x && \result >= y ;
ensures   \result == x   || \result == y ;
assigns   \nothing ; */
int max ( int  x , int  y ) ;
/*@ requires    x > INT_MIN ;
requires    y > INT_MIN ;
ensures  \result >= x &&  \result >= -x && \result >= y &&  \result >= -y ;
ensures   \result == x  || \result == -x ||  \result == y || \result == -y ;
assigns  \nothing ; */
int max abs ( int  x , int  y ) {
x=abs ( x ) ;
y=abs ( y ) ;
return max ( x , y ) ;
}
```

Loop invariants - some hints

- How to find a suitable loop invariant? Consider two aspects:
  - identify variables modified in the loop
    - variable number of iterations prevents from deducing their values (relationships with other variables)
    - define their possible value intervals (relationships) after k iterations
    - use loop assigns clause to list variables that (might) have been assigned so far after k iterations
  - identify realized actions, or properties already ensured by the loop
    - what part of the job already realized after k iterations?
    - what part of the expected loop results already ensured after k iterations?
    - why the next iteration can proceed as it does? . . .
- A stronger property on each iteration may be required to prove the final result of the loop
- Some experience may be necessary to find appropriate loop invariants

Loop invariants - more hints

- Remember: a loop invariant must be true
  - before (the first iteration of) the loop, even if no iteration is possible
  - after any complete iteration even if no more iterations are possible
  - in other words, any time before the loop condition check
- In particular, a for loop
  - for (i =0; i<n ; i++) { / body /}
- should be seen as
  - i =0; // action before the first iteration
  - whi le (i<n) // an iteration starts by the condition check
  - {
  - / body /
  - i++; // last action in an iteration
  - }

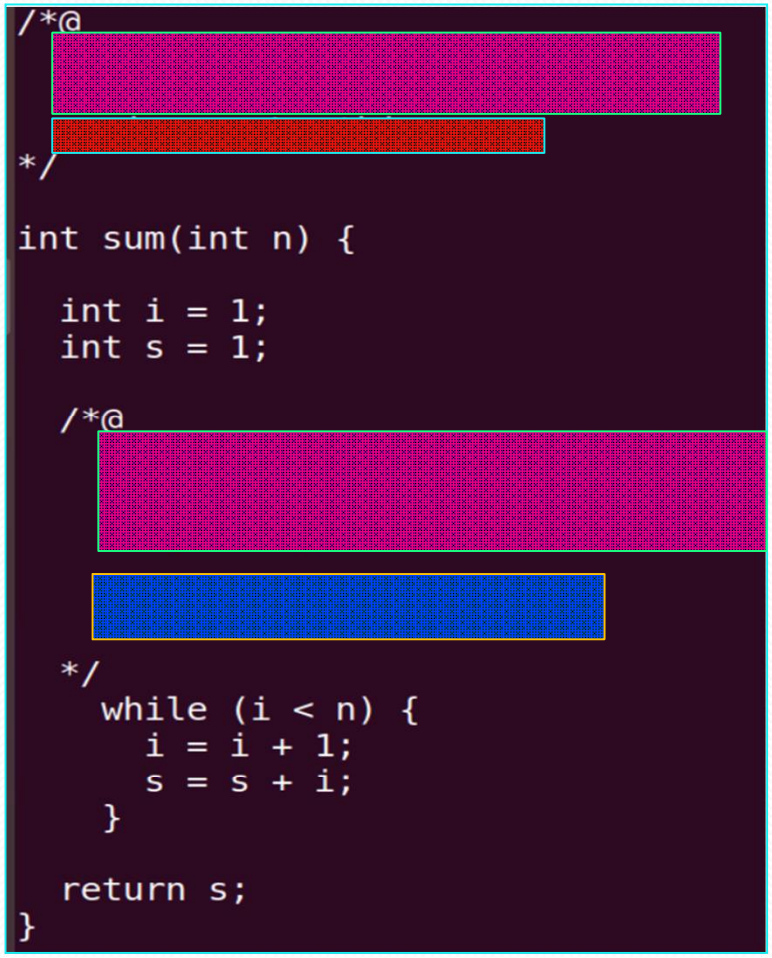
Loop termination

- Loop termination
  - Program termination is undecidable
  - A tool cannot deduce neither the exact number of iterations, nor even an upper bound
  - If an upper bound is given, a tool can check it by induction
  - An upper bound on the number of remaining loop iterations is the key idea behind the loop variant
- Terminology
  - Partial correctness: if the function terminates, it respects its specification
  - Total correctness: the function terminates, and it respects its specification

Loop variants - some hints

- Unlike an invariant, a loop variant is an integer expression, not a predicate
- Loop variant is not unique: if V works, V + 1 works as well
- No need to find a precise bound, any working loop variant is OK
- To find a variant, look at the loop condition
  - For the loop while(exp1 > exp2), try loop variant exp1-exp2;
- In more complex cases: ask yourself why the loop terminates, and try to give an integer upper bound on the number of remaining loop iterations

‘Sum 1 to N’ in Frama-C



```
/*@
   [redacted]
   [redacted]
*/

int sum(int n) {

  int i = 1;
  int s = 1;

  /*@
     [redacted]
     [redacted]
  */
    while (i < n) {
      i = i + 1;
      s = s + i;
    }

  return s;
}
```

1. Contract and Loop Invariant given within /*@ ... */

2. \result is function’s result.

3. Must explicitly state what changes using `assigns’ clause.

4. Must bound the loop index variable on both sides, e.g.

1 <= i <= n

5. Use loop variant for stating termination function.

Running Frama-C

```
bharat@bharat-VB: ~/FRAMA-C
File  Edit  View  Search  Terminal  Help
bharat@bharat-VB:~/FRAMA-C$
[kernel] Parsing FRAMAC_SHARE/libc/__fc_builtin_for
_normalization.i (no preprocessing)
[kernel] Parsing sum2.c (with preprocessing)
[wp] warning:
[wp] 9 goals scheduled
[wp]

bharat@bharat-VB:~/FRAMA-C$
bharat@bharat-VB:~/FRAMA-C$
bharat@bharat-VB:~/FRAMA-C$
bharat@bharat-VB:~/FRAMA-C$
```

RTE = Run-Time Environment

We will see an example where we turn on the RTE guard.