UNIVERSITÉ DE LORRAINE

TELECOM nancy
Ingénieurs du numérique • Inspiring your digital future

MVSI Course
Modelling and Verification
of Computer Systems

Lecture 5 – Mechanised Verification of Contracts (II)
The ANSI/ISO C Specification Language (ACSL)

Dominique Méry
Telecom Nancy, Université de Lorraine
November 20, 2026

Plan

1. Generation of Verification Conditions
   - WP calculus in Frama-c
   - First annotation
   - Second annotation
2. Annotations
3. Contracts
   - Extending C programming language by contracts
   - Playing with variables
   - Ghost Variables

Part 1

Generation of Verification Conditions

Verification Condition for Assignment

Listing 1: an1.c

```
1 //@ assert l1: P(x);
2   x = e(x);
3 //@ assert l2: Q(x);
```

Verification Condition for Assignment

Listing 2: an1.c

```
1 //@ assert l1: P(x);
2   x = e(x);
3 //@ assert l2: Q(x);
```

- $P(x) \Rightarrow WP(x := e(x))(Q(x))$

Verification Condition for Assignment

Listing 3: an1.c

```
1 //@ assert l1: P(x);
2   x = e(x);
3 //@ assert l2: Q(x);
```

- $P(x) \Rightarrow WP(x := e(x))(Q(x))$
- $P(x) \Rightarrow Q[x \mapsto e(x))]$

Verification Condition for Assignment

Listing 4: an1.c

```
1  //@ assert l1: P(x);
2    x = e(x);
3  //@ assert l2: Q(x);
```

- $P(x) \Rightarrow WP(x := e(x))(Q(x))$
- $P(x) \Rightarrow Q[x \mapsto e(x))]$
- $P(x1) \Rightarrow Q[x \mapsto e(x1))]$ (renaming of free occurences of x by x1)

Verification Condition for Assignment

Listing 5: an1.c

```
1 //@ assert l1: P(x);
2   x = e(x);
3 //@ assert l2: Q(x);
```

- $P(x) \Rightarrow WP(x := e(x))(Q(x))$
- $P(x) \Rightarrow Q[x \mapsto e(x))]$
- $P(x1) \Rightarrow Q[x \mapsto e(x1))]$ (renaming of free occurences of x by x1)
- $P(x1) \wedge x = e(x1) \Rightarrow Q(x)$

Verification Condition for Assignment

Listing 6: an1.c

```
1 //@ assert l1: P(x);
2   x = e(x);
3 //@ assert l2: Q(x);
```

- $P(x) \Rightarrow WP(x := e(x))(Q(x))$
- $P(x) \Rightarrow Q[x \mapsto e(x))]$
- $P(x1) \Rightarrow Q[x \mapsto e(x1))]$ (renaming of free occurences of x by x1)
- $P(x1) \wedge x = e(x1) \Rightarrow Q(x)$
- $P(x1) \wedge x = e(x1) \Rightarrow Q(x)$

Verification Condition for Assignment

Listing 7: an1.c

```
1 //@ assert l1: P(x);
2   x = e(x);
3 //@ assert l2: Q(x);
```

- $P(x) \Rightarrow WP(x := e(x))(Q(x))$
- $P(x) \Rightarrow Q[x \mapsto e(x))]$
- $P(x1) \Rightarrow Q[x \mapsto e(x1))]$ (renaming of free occurences of x by x1)
- $P(x1) \wedge x = e(x1) \Rightarrow Q(x)$
- $P(x1) \wedge x = e(x1) \Rightarrow Q(x)$
- $\vdash P(x1) \wedge x = e(x1) \Rightarrow Q(x)$

Verification Condition for Assignment

Listing 8: an1.c

```
1 //@ assert l1: P(x);
2   x = e(x);
3 //@ assert l2: Q(x);
```

- $P(x) \Rightarrow WP(x := e(x))(Q(x))$
- $P(x) \Rightarrow Q[x \mapsto e(x))]$
- $P(x1) \Rightarrow Q[x \mapsto e(x1))]$ (renaming of free occurences of x by x1)
- $P(x1) \wedge x = e(x1) \Rightarrow Q(x)$
- $P(x1) \wedge x = e(x1) \Rightarrow Q(x)$
- $\vdash P(x1) \wedge x = e(x1) \Rightarrow Q(x)$
- $P(x1) \wedge x = e(x1) \vdash Q(x)$

Verification Condition for Assignment

Listing 9: an1.c

```
1 //@ assert l1: P(x);
2   x = e(x);
3 //@ assert l2: Q(x);
```

- $P(x) \Rightarrow WP(x := e(x))(Q(x))$
- $P(x) \Rightarrow Q[x \mapsto e(x))]$
- $P(x1) \Rightarrow Q[x \mapsto e(x1))]$ (renaming of free occurences of x by x1)
- $P(x1) \wedge x = e(x1) \Rightarrow Q(x)$
- $P(x1) \wedge x = e(x1) \Rightarrow Q(x)$
- $\vdash P(x1) \wedge x = e(x1) \Rightarrow Q(x)$
- $P(x1) \wedge x = e(x1) \vdash Q(x)$

  `Assume` $\{$
  $P(x1)$
- $x = e(x1)$
  $\}$
  `Prove:` $Q(x)$
-

Simple annotation (I)

Listing 10: an1.c

```
1  int main(void){
2    signed long int x,y,z; //     int x,y,z;
3    x = 1;
4    /*@  assert x == 1; */
5    y = 2;
6    /*@ assert x == 1 && y == 2; */
7    z = x *y;
8    /*@  assert x == 1 && y == 1 && z == 2; */
9    return 0;
10   }
```

Simple annotation (I)

```
[kernel] Parsing an1.c (with preprocessing)
[wp] Running WP plugin...
[wp] Warning: Missing RTE guards
[wp] 2 goals scheduled
[wp] Proved goals:    4 / 4
  Terminating:      1
  Unreachable:      1
  Qed:              2
```

Simple annotation (I)

```
Goal Assertion '11' (file an1.c, line 3):
Assume {
  Type: is_sint32(x) /\ is_sint32(y).
  (* Initializer *)
  Init: x = 12.
  (* Initializer *)
  Init: y = 24.
}
Prove: (2 * x) = y.
Prover Qed returns Valid
```

Simple annotation (I)

```
------------------------------------------------------------

Goal Assertion 'l2' (file an1.c, line 5):
Assume {
  Type: is_sint32(x) /\ is_sint32(x_1) /\ is_sint32(y).
  (* Initializer *)
  Init: x_1 = 12.
  (* Initializer *)
  Init: y = 24.
  (* Assertion 'l1' *)
  Have: (2 * x_1) = y.
  Have: (1 + x_1) = x.
}
Prove: (2 + y) = (2 * x).
Prover Qed returns Valid
```

Simple annotation (I)

```
[kernel] Parsing an1.c (with preprocessing)
[wp] Running WP plugin...
[wp] Warning: Missing RTE guards
[wp] 2 goals scheduled
[wp] Proved goals:    4 / 4
  Terminating:      1
  Unreachable:      1
  Qed:              2
```

Simple annotation (II)

Listing 11: an2.c

```
1  void ex(void) {
2    int x=12,y=24;
3    //@ assert l1: 2*x == y;
4    x = x+1;
5    //@ assert l2:  y == 2*(x-1);
6    x = x+2;
7    //@ assert l3:  y+6 == 2*x;
8  }
```

Simple annotation (II)

```
[kernel] Parsing an2.c (with preprocessing)
[wp] Running WP plugin...
[wp] Warning: Missing RTE guards
[wp] 3 goals scheduled
[wp] Proved goals:    5 / 5
  Terminating:      1
  Unreachable:      1
  Qed:              3
```

Simple annotation (II)

```
Goal Assertion '11' (file an2.c, line 3):
Assume {
  Type: is_sint32(x) /\ is_sint32(y).
  (* Initializer *)
  Init: x = 12.
  (* Initializer *)
  Init: y = 24.
}
Prove: (2 * x) = y.
Prover Qed returns Valid
```

Simple annotation (II)

```
Goal Assertion '12' (file an2.c, line 5):
Assume {
  Type: is_sint32(x) /\ is_sint32(x_1) /\ is_sint32(y).
  (* Initializer *)
  Init: x_1 = 12.
  (* Initializer *)
  Init: y = 24.
  (* Assertion '11' *)
  Have: (2 * x_1) = y.
  Have: (1 + x_1) = x.
}
Prove: (2 + y) = (2 * x).
Prover Qed returns Valid
```

Simple annotation (II)

```
Goal Assertion 'l3' (file an2.c, line 7):
Assume {
  Type: is_sint32(x) /\ is_sint32(x_1) /\ is_sint32(x_2) /\ is_s
  (* Initializer *)
  Init: x_2 = 12.
  (* Initializer *)
  Init: y = 24.
  (* Assertion 'l1' *)
  Have: (2 * x_2) = y.
  Have: (1 + x_2) = x_1.
  (* Assertion 'l2' *)
  Have: (2 + y) = (2 * x_1).
  Have: (2 + x_1) = x.
}
Prove: (6 + y) = (2 * x).
Prover Qed returns Valid
```

Listing 12: an2bis.c

```
1  void ex(void) {
2    int x=12,y=24;
3    //@ assert l1: 2*x == y;
4    x = x+1;
5    //@ assert l2:  y == 2*(x-1);
6     x = x+2;
7    //@ assert l3:  y+6 == 2*x;
8  }
```

Part 2

Annotations

Writing a simple contract

variables x
requires $x >= 0 \wedge x <= 10;$
ensures $\begin{cases} x\%2 = 0 \Rightarrow 2 \cdot \text{result} = x+; \\ x\%2 \neq 0 \Rightarrow 2 \cdot \text{result} = x-1; \end{cases}$

begin
$int\ y;$
$y = x/2;$
$return(y);$
end

- result is the value returned by the command `rezturn(y)`.
- `return(y)` is equivalent to result:= y.

(Writing a simple contract.)

Listing 13: project-divers/annotation.c

```
1 /*@ requires x >= 0 && x <= 10;
2   @ assigns \nothing;
3   @ ensures x % 2 == 0 ==> 2*\result == x;
4  @ ensures x % 2 != 0 ==> 2*\result == x-1;
5 */
6 int f(int x)
7 {
8 int y;
9 y = x/2 ;
```

Writing a simple contract

(Writing a simple contract.)

Listing 14: project-divers/annotationwp.c

```
1  /*@ requires 0 <= x  && x <=  10;
2    @ assigns \nothing;
3    @ ensures  x % 2 == 0 ==> 2*\result == x;
4    @ ensures  x % 2 != 0 ==> 2*\result == x-1;
5    @*/
6
7  int f(int x)
8  {
9  /*@ assert x % 2 == 0 ==> 2* (x / 2) == x; */
10 /*@ assert   x % 2 != 0 ==> 2* (x / 2) == x-1; */
11   int y;
12 /*@ assert x % 2 == 0 ==> 2* (x / 2) == x; */
13 /*@ assert   x % 2 != 0 ==> 2* (x / 2) == x-1; */
14   y  = x / 2;
15 /*@ assert x % 2 == 0 ==> 2*y == x; */
16 /*@ assert   x % 2 != 0 ==> 2*y == x-1; */
17   return(y);
18 /*@ assert x % 2 == 0 ==> 2*y == x; */
19 /*@ assert   x % 2 != 0 ==> 2*y == x-1; */
20
21   }
```

Writing a simple contract

(Checking the precondition.)

Listing 15: project-divers/annotation0.c

```
1  /*@ requires x >= 0 && x <  0;
2    @ assigns \nothing;
3    @ ensures \result == 0;
4    @*/
5  int annotation0(int x)
6  {
7    int y;
8    y  =  y / (x-x);
9    return(y);
10   }
```

Writing a simple contract

(Checking the precondition.)

Listing 16: project-divers/annotation0wp.c

```
1  /*@ requires x >= 0 && x <  0;
2    @ assigns \nothing;
3    @ ensures \result == 0;
4    @*/
5  int annotation(int x)
6  {
7    /*@ assert y / (x-x) == 0; */
8    int y;
9    /*@ assert y / (x-x) == 0; */
10   y  =  y / (x-x);
11   /*@ assert y == 0; */
12   return(y);
13   /*@ assert y == 0; */
14   }
```

Property to check

$x \geq 0 \wedge < 0 \Rightarrow y/(x-x) = 0$

Transformations of annotated programs (1)

$//@$ assert $P(v0, v)$:
$S1; S2$
$//@$ assert $Q(v0, v)$:

- Applying the property: $wp(S1; S2)(A) = wp(S1)(wp(S2)(A))$
-

$//@$ assert $P(v0, v)$:
$S1;$
$//@$ assert $wp(S2)(Q(v0, v))$:
$S2;$
$//@$ assert $Q(v0, v)$:

$//@$ assert $P(v0, v)$:
$//@$ assert $xp(S1)(wp(S2)(Q(v0, v)))$:
$S1;$
$//@$ assert $wp(S2)(Q(v0, v))$:
$S2;$
$//@$ assert $Q(v0, v)$:

Transformations of annotated programs (2)

//@ assert $P(v0, v)$:
IF B THEN
  $S1$
ELSE
  $S2$
FI
//@ assert $Q(v0, v)$:

- Applying the property: $wp(if(B, S1, S2)(A) = b \wedge wp(S1)(A) \vee \neg B \wedge wp(S2)(A)$.
-

//@ assert $P(v0, v)$:
IF B THEN
  $S1$
ELSE
  $S2$
FI
//@ assert $Q(v0, v)$:

//@ assert $P(v0, v)$:
IF B THEN
  $S1$
//@ assert $Q(v0, v)$:
ELSE
  $S2$
//@ assert $Q(v0, v)$:
FI
//@ assert $Q(v0, v)$:

Transformations of annotated programs (2)

//@ assert $P(v0, v)$:
IF B THEN
  $S1$
//@ assert $Q(v0, v)$:
ELSE
  $S2$
//@ assert $Q(v0, v)$:
FI
//@ assert $Q(v0, v)$:

Transformations of annotated programs (2)

$//@$ assert $P(v0, v)$:
IF B THEN
  $S1$
$//@$ assert $Q(v0, v)$:
ELSE
  $S2$
$//@$ assert $Q(v0, v)$:
FI
$//@$ assert $Q(v0, v)$:

$//@$ assert $P(v0, v)$:
IF B THEN
$//@$ assert $B \wedge wp(S2)(Q(v0, v))$:
  $S1$
$//@$ assert $Q(v0, v)$:
ELSE
$//@$ assert $\neg B \wedge wp(S2)(Q(v0, v))$:
  $S2$
$//@$ assert $Q(v0, v)$:

Transformations of annotated programs (2)

//@ assert $P(v0, v)$:
IF B THEN
  $S1$
//@ assert $Q(v0, v)$:
ELSE
  $S2$
//@ assert $Q(v0, v)$:
FI
//@ assert $Q(v0, v)$:

//@ assert $P(v0, v)$:
IF B THEN
//@ assert $B \wedge wp(S2)(Q(v0, v))$:
  $S1$
//@ assert $Q(v0, v)$:
ELSE
//@ assert $\neg B \wedge wp(S2)(Q(v0, v))$:
  $S2$
//@ assert $Q(v0, v)$:

//@ assert $P(v0, v)$:
IF B THEN
//@ assert $b \wedge wp(S1)(Q(v0, v))$:
  $S1$
//@ assert $Q(v0, v)$:
ELSE
//@ assert $\neg b \wedge wp(S2)(Q(v0, v))$:
  $S2$
//@ assert $Q(v0, v)$:
FI
//@ assert $Q(v0, v)$:

Transformations of annotated programs (2)

//@ assert $P(v0, v)$:
IF B THEN
  $S1$
//@ assert $Q(v0, v)$:
ELSE
  $S2$
//@ assert $Q(v0, v)$:
FI
//@ assert $Q(v0, v)$:

//@ assert $P(v0, v)$:
IF B THEN
//@ assert $B \wedge wp(S2)(Q(v0, v))$:
  $S1$
//@ assert $Q(v0, v)$:
ELSE
//@ assert $\neg B \wedge wp(S2)(Q(v0, v))$:
  $S2$
//@ assert $Q(v0, v)$:

//@ assert $P(v0, v)$:
IF B THEN
//@ assert $b \wedge wp(S1)(Q(v0, v))$:
  $S1$
//@ assert $Q(v0, v)$:
ELSE
//@ assert $\neg b \wedge wp(S2)(Q(v0, v))$:
  $S2$
//@ assert $Q(v0, v)$:
FI
//@ assert $Q(v0, v)$:

- $b \wedge P(v0, v) \Rightarrow$
  $b \wedge wp(S1)(Q(v0, v))$
- $\neg b \wedge P(v0, v) \Rightarrow$
  $\neg b \wedge wp(S2)(Q(v0, v))$

Transformations of annotated programs (3)

//@ assert $P(v0, v)$:
//@ loop invariant $I(v0, v)$:
WHILE B THEN
  S
OD
//@ assert $Q(v0, v)$:

- Applying the iteration rule of Hoare Logic:

Transformations of annotated programs (3)

```
//@ assert P(v0, v) :
//@ loop invariant I(v0, v) :
WHILE B THEN
   S
OD
//@ assert Q(v0, v) :
```

- Applying the iteration rule of Hoare Logic:

```
//@ assert P(v0, v) :
//@ loop invariant I(v0, v) :
//@ assert I(v0, v) :
WHILE B THEN
//@ assert b ∧ I(v0, v) :
   S
//@ assert I(v0, v) :
OD
//@ assert Q(v0, v) :
```

Transformations of annotated programs (3)

```
//@ assert P(v0, v) :
//@ loop invariant I(v0, v) :
WHILE B THEN
   S
OD
//@ assert Q(v0, v) :
```

- Applying the iteration rule of Hoare Logic:

```
//@ assert P(v0, v) :
//@ loop invariant I(v0, v) :
//@ assert I(v0, v) :
WHILE B THEN
//@ assert b ∧ I(v0, v) :
   S
//@ assert I(v0, v) :
OD
//@ assert Q(v0, v) :
```

- $b \wedge I(v0, v) \Rightarrow wp(S)(I(v0, v))$
- $P(v0, v) \Rightarrow I(v0, v))$
- $\neg b \wedge I(v0, v) \Rightarrow Q(v0, v)$

Summary of transformations

- Checking the preservation of invariant.
- Applying the wps on assertions according to startements.

Assertions

- Assertions at a control point of the program

```
/*@  assert pred; */

//@  assert pred;
```

- Assertions at a control point of the program components.

```
/*@  for id1,id2, ..., idn: assert pred; */
```

Verification using WLP

(Incrementing a number)

Listing 17: project-divers/compwp0.c

```
1 #define x0 5
2 /*@ assigns \nothing;*/
3 int  exemple() {
4   int x=x0;
5   //@ assert x == x0;
6   x = x + 1;
7 //@ assert  x== x0+1;
8 return x;
9 }
```

(Incrementing a number)

Listing 18: project-divers/compwp0wp.c

```
1 #define x0 5
2 /*@ assigns \nothing; */
3 int  exemple() {
4   //@ assert x0 == x0;
5 //@ assert  x0+1 == x0+1;
6   int x=x0;
7   //@ assert x == x0;
```

Summary on annotations and assertions

- requires
- assigns
- ensures
- decreases
- predicate
- logic
- lemma

Part 3

Contracts

Programming by contract

- The calling function should garantee the required condition or precondition introduced by the clauses requires $P1 \wedge \ldots \wedge Pn$ at the calling point.
- The called function returns results that are ensured by the clause ensures $E1 \wedge \ldots \wedge Em$; ensures clause exporess a relatrionship between the initial values of variables and the final values.
- initial values of a variable v is denoted $\backslash old(v)$
- The variables which are not in the set $L1 \cup \ldots \cup Lp$ are not modified.

Listing 19: contrat

```
1 /*@ requires P1;...;requires Pn;
2   @ assigns L1;...;assigns Lm;
3   @ ensures E1;...;ensures Ep;
4   @*/
```

Definition of a contract (specification)

- Define the mathematical fucntion to compute (what to compute?)
- Define an inductive method for computing the mathematical function and using axioms.

(Factorial: what)

Listing 20: project-factorial/factorial.h

```
1  #ifndef _A_H
2  #define _A_H
3  /*@ axiomatic mathfact {
4    @ logic integer mathfact(integer n);
5    @ axiom mathfact_1: mathfact(1) == 1;
6    @ axiom mathfact_rec: \forall integer n; n > 1
7    ==> mathfact(n) == n * mathfact(n-1);
8    @ } */
9
10 /*@ requires n > 0;
11   decreases n;
12   ensures \result == mathfact(n);
13   assigns \nothing;
14 */
15 int codefact(int n);
16 #endif
```

Definition of a contract (programming)

- Define the program codefact for computing mathfact (How to compute?)
- Define the algorithm computing the function mathfact

(Factorial: how)

Listing 21: project-factorial/factorial.c

```
1  #include "factorial.h"
2
3  int codefact(int n) {
4    int y = 1;
5    int x = n;
6    /*@ loop invariant x >= 1 && x <= n && mathfact(n) == y *
         mathfact(x);
7      loop assigns x, y;
8      loop variant x;
9     */
10   while (x != 1) {
11     y = y * x;
12     x = x - 1;
13   };
14   return y;
15
16 }
```

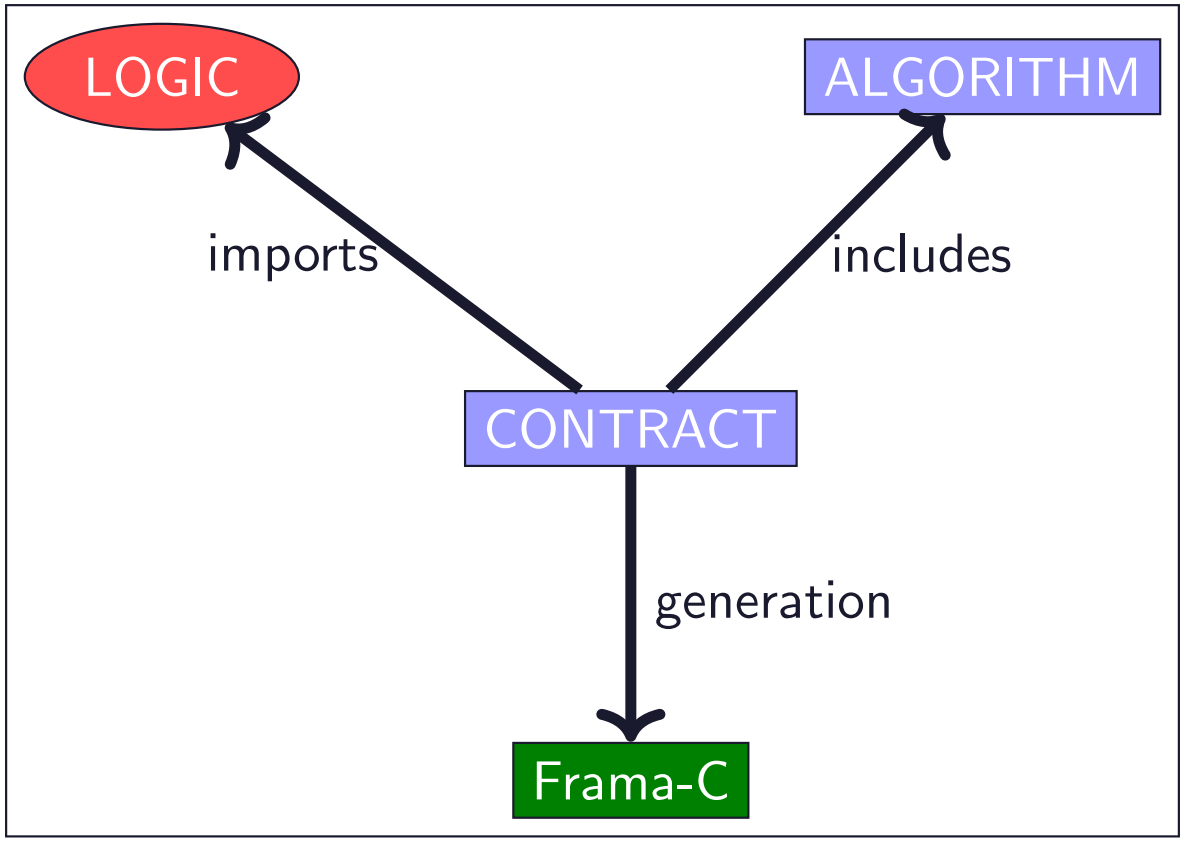
Definition of a contract (approach)

- The specification of a function (mathfact) to compute requires to define it mathematically.
- The definition is stated in an axtiomatic framework and is preferably inductive (mathfact) which is used in assrtions or theorems or lemmas.
- The relationship between the ciomputed value (\result) and the mathematical value (mathfact(n)) is stated in the ensures clause:

$$\backslash\text{result} == \text{mathfact(n)}$$

- The main property to prove is codefact(n)==mathfact(n): Calling codefact for n returns a value equal to mathfact(n).

Tryptic Pattern



Contracts with named behaviours

Listing 22: contrat

```
1  /*@ requires P;
2  @ behavior b1:
3    @ assumes A1;
4    @ requires R1 ;
5    @ assigns L1;
6    @ ensures E1;
7  @ behavior b2:
8    @ assumes A2;
9    @ requires R2;
10   @ assigns L2;
11   @ ensures E2;
12 @*/
```

Division should not return silly expressions!

(Pairs of integers)

Listing 23: project-divers/structures.h

```
1 #ifndef _STRUCTURE_H
2
3
4 struct s {
5     int q;
6     int r;
7 };
8
9 #endif
```

Division should not return silly expressions!

(Specification)

Listing 24: project-divers/division.h

```
1  #ifndef _A_H
2  #define _A_H
3  #include "structures.h"
4  /*@ requires a >= 0 && b >= 0;
5    @ requires  b != 0;
6    @ ensures 0 <= \result.r;
7    @ ensures \result.r < b;
8    @ ensures  a ==  b * \result.q + \result.r;
9  */
10 struct s  division(int a, int b);
11 #endif
```

Division should not return silly expressions!

(Algorithm)

Listing 25: project-divers/division.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "division.h"
4
5  struct s  division(int a, int b)
6  {  int rr = a;
7     int qq = 0;
8     struct s  silly = {-1,-1};
9     struct s resu;
10    if (b == 0) {
11      return silly;
12    }
13    else
14      {
15   /*@
16     loop invariant
17     ( a ==  b*qq + rr) &&
18     rr >= 0;
19     loop assigns rr,qq;
20     loop variant rr;
21    */
22    while (rr >= b) { rr = rr - b; qq=qq+1;};
23    resu.q= qq;
24    resu.r = rr;
```

Management of loop

Iteration Rule for PC

If $\{P \wedge B\}\mathbf{S}\{P\}$, then $\{P\}$**while B do S od**$\{P \wedge \neg B\}$.

- Prove $\{P \wedge B\}\mathbf{S}\{P\}$ or $P \wedge B \Rightarrow \{\mathbf{S}\}(P)$.
- By the iteration rule, we conclude that $\{P\}$**while B do S od**$\{P \wedge \neg B\}$ without using WLP.
- Introduction of LOOP INVARIANTS in the notation.

Listing 26: loop.c

```
1  /*@ loop invariant I1;
2      loop invariant I2;
3      ...
4      loop invariant In;
5      loop assigns X;
6      loop variant E;
7    */
```

(Loop invariant)

Listing 27: project-divers/anno6.c

```
1
2  /*@ requires a >= 0 && b >= 0;
3    ensures 0 <= \result;
4    ensures \result < b;
5    ensures \exists integer k; a == k * b + \result;
6  */
7  int rem(int a, int b) {
8    int r = a;
9    /*@
10     loop invariant
11     (\exists integer i; a == i * b + r) &&
12     r >= 0;
13     loop assigns r;
14    */
15   while (r >= b) { r = r - b;};
16   return r;
17 }
```

Initial value of x $\backslash old(x)$

- $\backslash old(x)$ is the value of the variable when the function is called.
- It can be used in the postcondition of the *ensures* clause.

(Modifying variables while calling)

Listing 28: project-divers/old1.c

```
1
2  /*@ requires \valid(a) && \valid(b);
3     @ assigns *a,*b;
4      @ ensures   *a == \at(*a,Pre) +2;
5      @ ensures  *b == \at(*b,Pre)+\at(*a,Pre)+2;
6
7           @ ensures  \result == 0;
8  */
9  int old(int *a, int *b) {
10   int x,y;
11   x = *a;
12   y = *b;
13   x=x+2;
14    y = y +x;
15
16   *a = x;
17   *b = y;
18   return 0 ;
19 }
```

Expression $\backslash at(e, id)$

- $\backslash at(e, id)$ is the value of e at the control point id.
- id should occur before $\backslash at(e, id)$
- id is one of the possible expressions: Pre, Here, Old, Post, LoopEntry, LoopCurrent, Init
- $\backslash old(e)$ is equivalent to $\backslash at(e, Old)$

Example for $\backslash at(e, id)$

(label Pre)

Listing 29: project-divers/at1.c

```
1  /*@
2    requires \valid(a) && \valid(b);
3    assigns *a,*b;
4    ensures  *a == \old(*a)+2;
5    ensures  *b == \old(*b)+\old(*a)+2;
6  */
7  int at1(int *a, int *b) {
8  //@ assert *a == \at(*a,Pre);
9    *a = *a +1;
10 //@ assert *a == \at(*a,Pre)+1;
11   *a = *a +1;
12 //@ assert *a == \at(*a,Pre)+2;
13   *b = *b +*a;
14 //@ assert *a == \at(*a,Pre)+2 && *b == \at(*b,Pre)+\at(*a,Pre)
       +2;
15   return 0;
16 }
```

-

Example for $\backslash at(e, id)$

(Another label)

Listing 30: project-divers/at2.c

```
1 void f (int n) {
2 for (int i = 0; i < n; i++) {
3 /*@ assert \at(i,LoopEntry) == 0; */
4 int j=0;
5 while (j++ < i) {
6 /*@ assert \at(j,LoopEntry) == 0; */
7 /*@ assert \at(j,LoopCurrent) + 1 == j; */
8 } }
9 }
```

Example for $\backslash at(e, id)$

(otherlabel)

Listing 31: project-divers/change1.c

```
1  /*@ requires \valid(a) && *a >= 0;
2    @ assigns *a;
3    @ ensures   *a == \old(*a)+2 && \result == 0;
4  */
5  int  change1(int *a)
6  {  int x = *a;
7     x = x + 2;
8     *a = x;
9     return 0;
10 }
```

Ghost variable

- A variable called *ghost* allows to model a computed value useful for stating a model property: the ghost variable is hidden for the computer but not for the model.
- It should not change the semantics of others variables and should not change the effective variables.

Wrong use of ghost variable

(Bug)

Listing 32: project-divers/ghost2.c

```
1  int f (int x, int y) {
2    //@ghost int z=x+y;
3  switch (x) {
4  case 0: return y;
5  //@ ghost case 1: z=y;
6  // above statement is correct.
7  //@ ghost case 2: { z++; break; }
8  // invalid, would bypass the non-ghost default
9  default: y++; }
10 return y; }
11
12 int g(int x) { //@ ghost int z=x;
13 if (x>0){return x;}
14 //@ ghost else { z++; return x; }
15 // invalid, would bypass the non-ghost return
16 return x+1; }
```

Programming by contract

(Ghost variable)

Listing 33: project-divers/ghost1.c

```
1  /*@ requires a >= 0 && b >= 0;
2    ensures 0 <= \result;
3    ensures \result < b;
4    ensures \exists integer k; a == k * b + \result; */
5  int rem(int a, int b) {
6    int r = a;
7   /*@ ghost   int q=0;    */
8    /*@
9      loop invariant
10     a == q * b + r &&
11     r >= 0 &&  r <= a;
12     loop assigns r;
13     loop assigns q;
14 // loop variant r;
15 */
16   while (r >= b) {
17     r = r - b;
18 /*@ ghost     q = q+1;    */
19   };
20   return r;
21 }
```