
Cours MALG & MOVEX

Vérification mécanisée de contrats (II) (The ANSI/ISO C Specification Language (ACSL))

Dominique Méry
Telecom Nancy, Université de Lorraine

Année universitaire 2023-2024

① Programs as Predicate

Transformers

② Annotations

③ Contracts

Extending C programming
language by contracts

Playing with variables

Ghost Variables

① Programs as Predicate Transformers

② Annotations

③ Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

① Programs as Predicate Transformers

② Annotations

③ Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

① Programs as Predicate Transformers

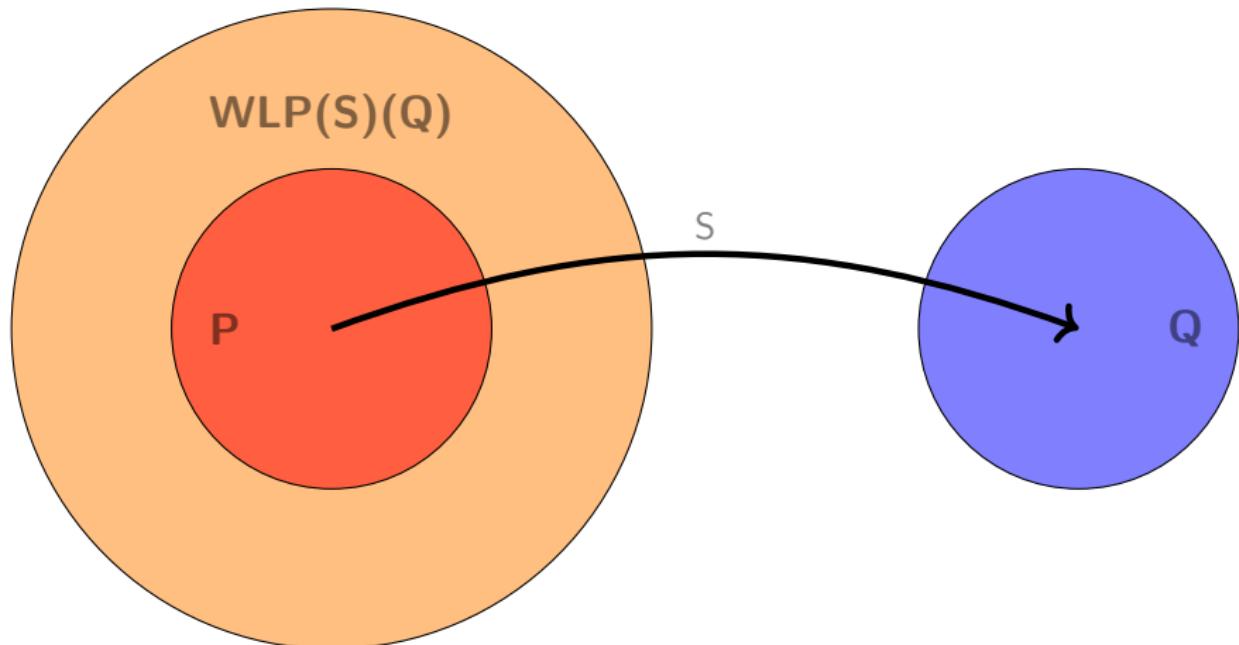
② Annotations

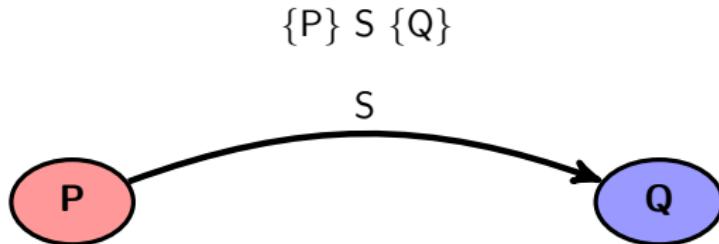
③ Contracts

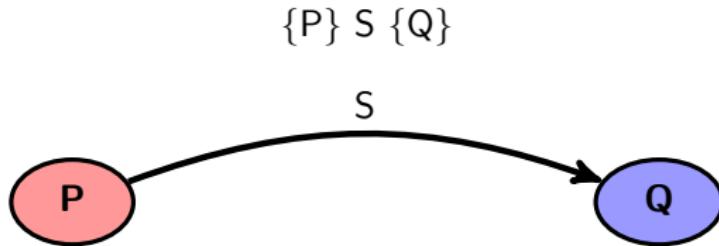
Extending C programming language by contracts

Playing with variables

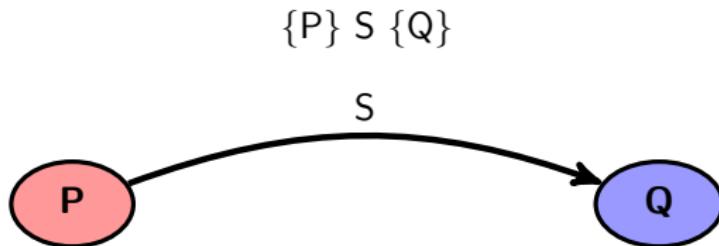
Ghost Variables







$P \Rightarrow WLP(S)(Q)$



$$P \Rightarrow WLP(S)(Q)$$

Computing $WLP(S)(Q)$?

① Programs as Predicate Transformers

② Annotations

③ Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

Writing a simple contract

variables x

requires $x \geq 0 \wedge x \leq 10$;

ensures $\left\{ \begin{array}{l} x \% 2 = 0 \Rightarrow 2 \cdot \text{result} = x +; \\ x \% 2 \neq 0 \Rightarrow 2 \cdot \text{result} = x - 1; \end{array} \right.$

```
begin
```

```
int y;
```

$$y = x/2;$$

```
return(y);
```

end

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

(Writing a simple contract.)

Listing 1 – project-divers/annotation.c

```

/*@ requires x >= 0 && x <= 10;
 @ assigns \nothing;
 @ ensures x % 2 == 0 ==> 2*\result == x;
 @ ensures x % 2 != 0 ==> 2*\result == x-1;
 @*/
int annotation(int x)
{
    int y;
    y = x / 2;
    return(y);
}

```

Writing a simple contract

(Writing a simple contract.)

Listing 2 – project-divers/annotationwp.c

```
/*@ requires 0 <= x && x <= 10;
 @ assigns \nothing;
 @ ensures x \% 2 == 0 ==> 2*\result == x;
 @ ensures x \% 2 != 0 ==> 2*\result == x-1;
 @*/
int annotation(int x)
{
/*@ assert x \% 2 == 0 ==> 2* (x / 2) == x; */
/*@ assert x \% 2 != 0 ==> 2* (x / 2) == x-1; */
    int y;
/*@ assert x \% 2 == 0 ==> 2* (x / 2) == x; */
/*@ assert x \% 2 != 0 ==> 2* (x / 2) == x-1; */
    y = x / 2;
/*@ assert x \% 2 == 0 ==> 2*y == x; */
/*@ assert x \% 2 != 0 ==> 2*y == x-1; */
    return(y);
/*@ assert x \% 2 == 0 ==> 2*y == x; */
/*@ assert x \% 2 != 0 ==> 2*y == x-1; */
}
```

Property to check

$$x \geq 0 \wedge x \leq 10 \Rightarrow \begin{cases} x \% 2 \neq 0 \Rightarrow 2 \cdot (x/2) = x-1 \\ x \% 2 = 0 \Rightarrow 2 \cdot (x/2) = x \end{cases}$$

Writing a simple contract

(Checking the precondition.)

Listing 3 – project-divers/annotation0.c

```
/*@ requires x >= 0 && x < 0;
 @ assigns \nothing;
 @ ensures \result == 0;
 */
int annotation0(int x)
{
    int y;
    y = y / (x-x);
    return(y);
}
```

Writing a simple contract

(Checking the precondition.)

Listing 4 – project-divers/annotation0wp.c

```
/*@ requires x >= 0 && x < 0;
 @ assigns \nothing;
 @ ensures \result == 0;
 */
int annotation(int x)
{
    /*@ assert y / (x-x) == 0; */
    int y;
    /*@ assert y / (x-x) == 0; */
    y = y / (x-x);
    /*@ assert y == 0; */
    return(y);
    /*@ assert y == 0; */
}
```

Property to check

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

```
//@ 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 ∧ wp(S2)(Q(v0,v)) :  
    S1  
  //@ assert Q(v0,v) :  
ELSE  
  //@ assert ¬B ∧ 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 ∧ wp(S2)(Q(v0, v)) :  
    S1  
  //@ assert Q(v0, v) :  
ELSE  
  //@ assert ¬B ∧ wp(S2)(Q(v0, v)) :  
    S2  
  //@ assert Q(v0, v) :
```

```
//@ assert P(v0, v) :  
IF B THEN  
  //@ assert b ∧ wp(S1)(Q(v0, v)) :  
    S1  
  //@ assert Q(v0, v) :  
ELSE  
  //@ assert ¬b ∧ wp(S2)(Q(v0, v)) :  
    S2  
  //@ assert Q(v0, v) :  
//@ 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 ∧ wp(S2)(Q(v0, v)) :  
    S1  
    //@ assert Q(v0, v) :  
  ELSE  
    //@ assert ¬B ∧ wp(S2)(Q(v0, v)) :  
      S2  
      //@ assert Q(v0, v) :
```

```
//@ assert P(v0, v) :  
IF B THEN  
  //@ assert b ∧ wp(S1)(Q(v0, v)) :  
    S1  
    //@ assert Q(v0, v) :  
  ELSE  
    //@ assert ¬b ∧ wp(S2)(Q(v0, v)) :  
      S2  
      //@ assert Q(v0, v) :  
//@ assert Q(v0, v) :
```

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

```
//@ assert  $P(v_0, v)$  :  
//@ loop invariant  $I(v_0, v)$  :  
WHILE  $B$  THEN  
     $S$   
OD  
//@ assert  $Q(v_0, v)$  :
```

- ▶ Applying the iteration rule of Hoare Logic :

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

```
//@ 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) :
```

- ▶ 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) :
```

```
//@ 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) :
```

- ▶ Applying the iteration rule of Hoare Logic :

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

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

- ▶ 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 5 – project-divers/compwp0.c

```
#define x0 5
/*@ assigns \nothing; */
int exemple() {
    int x=x0;
    //@ assert x == x0;
    x = x + 1;
    //@ assert x == x0+1;
    return x;
}
```

(Incrementing a number)

Listing 6 – project-divers/compwp0wp.c

```
#define x0 5
/*@ assigns \nothing; */
int exemple() {
    //@ assert x0 == x0;
    //@ assert x0+1 == x0+1;
    int x=x0;
    //@ assert x == x0;
    //@ assert x+1 == x0+1;
    x = x + 1;
    //@ assert x == x0+1;
    return x;
}
```

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

① Programs as Predicate Transformers

② Annotations

③ Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

① Programs as Predicate Transformers

② Annotations

③ Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

- ▶ The calling function should guarantee the required condition or precondition introduced by the clauses requires $P_1 \wedge \dots \wedge P_n$ at the calling point.
- ▶ The called function returns results that are ensured by the clause ensures $E_1 \wedge \dots \wedge E_m$; ensures clause expresses a relationship between the initial values of variables and the final values.
- ▶ initial values of a variable v is denoted $\text{old}(v)$
- ▶ The variables which are not in the set $L_1 \cup \dots \cup L_p$ are not modified.

Listing 7 – contrat

```
/*@ requires P1;...; requires Pn;  
 @ assigns L1;...; assigns Lm;  
 @ ensures E1;...; ensures Ep;  
 @*/
```

Examples of contract (1)

(Division)

Listing 8 – project-divers/annotation.c

```
/*@ requires x >= 0 && x <= 10;
 @ assigns \nothing;
 @ ensures x \% 2 == 0 ==> 2*\result == x;
 @ ensures x \% 2 != 0 ==> 2*\result == x-1;
 */
int annotation(int x)
{
    int y;
    y = x / 2;
    return(y);
}
```

Examples of contract (1)

(Division)

Listing 9 – project-divers/annotationwp.c

```
/*@ requires 0 <= x && x <= 10;
 @ assigns \nothing;
 @ ensures x \% 2 == 0 ==> 2*\result == x;
 @ ensures x \% 2 != 0 ==> 2*\result == x-1;
 @*/
int annotation(int x)
{
    /*@ assert x \% 2 == 0 ==> 2* (x / 2) == x; */
    /*@ assert x \% 2 != 0 ==> 2* (x / 2) == x-1; */
    int y;
    /*@ assert x \% 2 == 0 ==> 2* (x / 2) == x; */
    /*@ assert x \% 2 != 0 ==> 2* (x / 2) == x-1; */
    y = x / 2;
    /*@ assert x \% 2 == 0 ==> 2*y == x; */
    /*@ assert x \% 2 != 0 ==> 2*y == x-1; */
    return(y);
    /*@ assert x \% 2 == 0 ==> 2*y == x; */
    /*@ assert x \% 2 != 0 ==> 2*y == x-1; */
}
```

Examples of contract (1)

Property to check

$$x \geq 0 \wedge x < 0; \Rightarrow \left(\begin{array}{l} x \% 2 = 0 \Rightarrow 2 \cdot (x/2) = x \\ x \% 2 \neq 0 \Rightarrow 2 \cdot (x/2) = x-1 \end{array} \right)$$

Examples of contract (2)

(Precondition)

Listing 10 – project-divers/annotation0.c

```
/*@ requires x >= 0 && x < 0;
 @ assigns \nothing;
 @ ensures \result == 0;
 */
int annotation0(int x)
{
    int y;
    y = y / (x-x);
    return(y);
}
```

Examples of contract (2)

(Precondition)

Listing 11 – project-divers/annotation0wp.c

```
/*@ requires x >= 0 && x < 0;
 @ assigns \nothing;
 @ ensures \result == 0;
 */
int annotation(int x)
{
    /*@ assert y / (x-x) == 0; */
    int y;
    /*@ assert y / (x-x) == 0; */
    y = y / (x-x);
    /*@ assert y == 0; */
    return(y);
    /*@ assert y == 0; */
}
```

Examples of contract (2)

Property to check

$$0 \leq x \wedge x \leq 10 \Rightarrow y/(x-x) = 0$$

Definition of a contract (specification)

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

(facctorial what)

Listing 12 – project-factorial/factorial.h

```
#ifndef _A_H
#define _A_H
/*@ axiomatic mathfact {
    @ logic integer mathfact(integer n);
    @ axiom mathfact_1: mathfact(1) == 1;
    @ axiom mathfact_rec: \forall integer n; n > 1
        ==> mathfact(n) == n * mathfact(n-1);
    @ } */
/*@ requires n > 0;
   decreases n;
   ensures \result == mathfact(n);
   assigns \nothing;
*/
int codefact(int n);
#endif
```

Definition of a contract (programming)

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

(facctrorial how)

Listing 13 – project-factorial/factorial.c

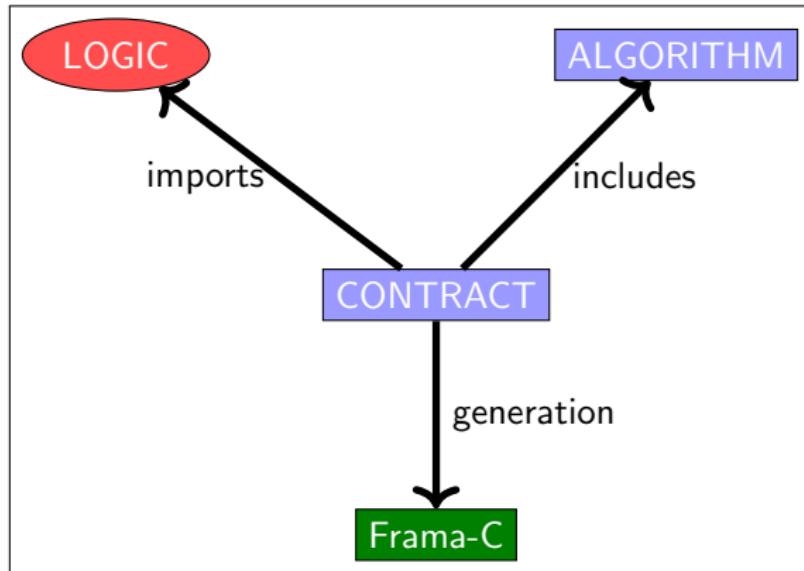
```
#include "factorial.h"

int codefact(int n) {
    int y = 1;
    int x = n;
    /*@ loop invariant x >= 1 && x <= n && mathfact(n) == y * mathfact(x);
       loop assigns x, y;
       loop variant x;
    */
    while (x != 1) {
        y = y * x;
        x = x - 1;
    };
    return y;
}
```

Definition of a contract (approach)

- ▶ The specification of a function (mathfact) to compute requires to define it mathematically.
- ▶ The definition is stated in an axiomatic framework and is preferably inductive (mathfact) which is used in assertions or theorems or lemmas.
- ▶ The relationship between the computed value (`\result`) and the mathematical value (mathfact(n)) is stated in the ensures clause :
$$\result == \text{mathfact}(n)$$
- ▶ The main property to prove is `codefact(n)==mathfact(n)` : Calling codefact for n returns a value equal to $\text{mathfact}(n)$.

Tryptic Pattern



Listing 14 – contrat

```
/*@ requires P;
@ behavior b1:
  @ assumes A1;
  @ requires R1 ;
  @ assigns L1;
  @ ensures E1;
@ behavior b2:
  @ assumes A2;
  @ requires R2;
  @ assigns L2;
  @ ensures E2;
*/

```

(Pairs of integers)

Listing 15 – project-divers/structures.h

```
#ifndef _STRUCTURE_H

struct s {
    int q;
    int r;
};

#endif
```

(Specification)

Listing 16 – project-divers/division.h

```
#ifndef _A_H
#define _A_H
#include "structures.h"
/*@ requires a >= 0 && b >= 0;
@ behavior b :
    @ assumes b == 0;
    @ assigns \nothing;
    @ ensures \result.q == -1 && \result.r == -1 ;
@ behavior B2:
    @ assumes b != 0;
    @ assigns \nothing;
    @ ensures 0 <= \result.r;
    @ ensures \result.r < b;
    @ ensures a == b * \result.q + \result.r;
*/
struct s division(int a, int b);
#endif
```

Division shpuld not return silly expressions !

(Algorithm)

Listing 17 – project-divers/division.c

```
#include <stdio.h>
#include <stdlib.h>
#include "division.h"

struct s division(int a, int b)
{   int rr = a;
    int qq = 0;
    struct s silly = {-1,-1};
    struct s resu;
    if (b == 0) {
        return silly;
    }
    else
    {
        /*@
         * loop invariant
         * ( a == b*qq + rr ) &&
         * rr >= 0;
         * loop assigns rr,qq;
         * loop variant rr;
        */
        while (rr >= b) { rr = rr - b; qq=qq+1;};
        resu.q= qq;
        resu.r = rr;
        return resu;
    }
}
```

Iteration Rule for PC

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

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

Listing 18 – loop.c

```
/*@ loop invariant I1;
   loop invariant I2;
   ...
   loop invariant In;
   loop assigns X;
   loop variant E;
*/

```

(Invariant de boucle)

Listing 19 – project-divers/anno6.c

```
/*@ requires a >= 0 && b >= 0;
ensures 0 <= \result;
ensures \result < b;
ensures \exists integer k; a == k * b + \result;
*/
int rem(int a, int b) {
    int r = a;
    /*@
    loop invariant
    (\exists integer i; a == i * b + r) &&
    r >= 0;
    loop assigns r;
    */
    while (r >= b) { r = r - b; }
    return r;
}
```

① Programs as Predicate Transformers

② Annotations

③ Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

- ▶ \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 20 – project-divers/old1.c

```
/*@ requires \valid(a) && \valid(b);
 @ assigns *a,*b;
 @ ensures  *a == \at(*a,Pre) +2;
 @ ensures  *b == \at(*b,Pre)+\at(*a,Pre)+2;
 @ ensures  \result == 0;
*/
int old(int *a, int *b) {
    int x,y;
    x = *a;
    y = *b;
    x=x+2;
    y = y +x;

    *a = x;
    *b = y;
    return 0 ;
}
```

- ▶ $\backslash at(e, id)$ 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 $\backslash at(e, Old)$

Exemple pour $\text{\textbackslash}at(e, id)$

(label Pre)

Listing 21 – project-divers/at1.c

```
/*@
  requires \valid(a) && \valid(b);
  assigns *a,*b;
  ensures  *a == \old(*a)+2;
  ensures  *b == \old(*b)+\old(*a)+2;
*/
int at1(int *a, int *b) {
    //@ assert *a == \at(*a,Pre);
    *a = *a +1;
    //@ assert *a == \at(*a,Pre)+1;
    *a = *a +1;
    //@ assert *a == \at(*a,Pre)+2;
    *b = *b +*a;
    //@ assert *a == \at(*a,Pre)+2 && *b == \at(*b,Pre)+\at(*a,Pre)+2;
    return 0;
}
```


Example for \at(e, id)

(autre label)

Listing 22 – project-divers/at2.c

```
void f (int n) {
    for (int i = 0; i < n; i++) {
        /*@ assert \at(i, LoopEntry) == 0; */
        int j=0;
        while (j++ < i) {
            /*@ assert \at(j, LoopEntry) == 0; */
            /*@ assert \at(j, LoopCurrent) + 1 == j; */
        }
    }
}
```

(otherlabel)

Listing 23 – project-divers/change1.c

```
/*@ requires \valid(a) && *a >= 0;
 @ assigns *a;
 @ ensures   *a == \old(*a)+2 && \result == 0;
 */
int change1(int *a)
{  int x = *a;
   x = x + 2;
   *a = x;
   return 0;
}
```

① Programs as Predicate Transformers

② Annotations

③ Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

- ▶ 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 24 – project-divers/ghost2.c

```
int f (int x, int y) {
    //@ghost int z=x+y;
    switch (x) {
        case 0: return y;
        //@ ghost case 1: z=y;
        // above statement is correct.
        //@ ghost case 2: { z++; break; }
        // invalid, would bypass the non-ghost default
        default: y++; }
    return y; }

int g(int x) { //@ ghost int z=x;
    if (x>0){return x;}
    //@ ghost else { z++; return x; }
    // invalid, would bypass the non-ghost return
    return x+1; }
```

(Ghost variable)

Listing 25 – project-divers/ghost1.c

```
/*@ requires a >= 0 && b >= 0;
ensures 0 <= \result;
ensures \result < b;
ensures \exists integer k; a == k * b + \result; */
int rem(int a, int b) {
    int r = a;
    /*@ ghost int q=0; */
    /*@
        loop invariant
        a == q * b + r &&
        r >= 0 && r <= a;
        loop assigns r;
        loop assigns q;
    // loop variant r;
    */
    while (r >= b) {
        r = r - b;
    /*@ ghost q = q+1; */
    };
    return r;
}
```