

Cours MALG & MOVEX

MALG

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

Dominique Méry
Telecom Nancy, Université de Lorraine

① Contracts

- Extending C programming language by contracts
- Playing with variables
- Ghost Variables

② Generation of Verification Conditions

- WP calculus in Frama-c
- First annotation
- Second annotation

③ Memory Models in Frama-c

④ Logic Specification

⑤ Organisation of the verification process

⑥ Conclusion

1 Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

2 Generation of Verification Conditions

WP calculus in Frama-c

First annotation

Second annotation

3 Memory Models in Frama-c

4 Logic Specification

5 Organisation of the verification process

6 Conclusion

① Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

② Generation of Verification Conditions

WP calculus in Frama-c

First annotation

Second annotation

③ Memory Models in Frama-c

④ Logic Specification

⑤ Organisation of the verification process

⑥ Conclusion

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

1 Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

2 Generation of Verification Conditions

WP calculus in Frama-c

First annotation

Second annotation

3 Memory Models in Frama-c

4 Logic Specification

5 Organisation of the verification process

6 Conclusion

1 Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

2 Generation of Verification Conditions

WP calculus in Frama-c

First annotation

Second annotation

3 Memory Models in Frama-c

4 Logic Specification

5 Organisation of the verification process

6 Conclusion

- ▶ 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 1 – contrat

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

Examples of contract (1)

(Division)

Listing 2 – 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 3 – 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 4 – 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 5 – 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

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

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

(facctrorial what)

Listing 6 – project-facctrorial/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
```

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

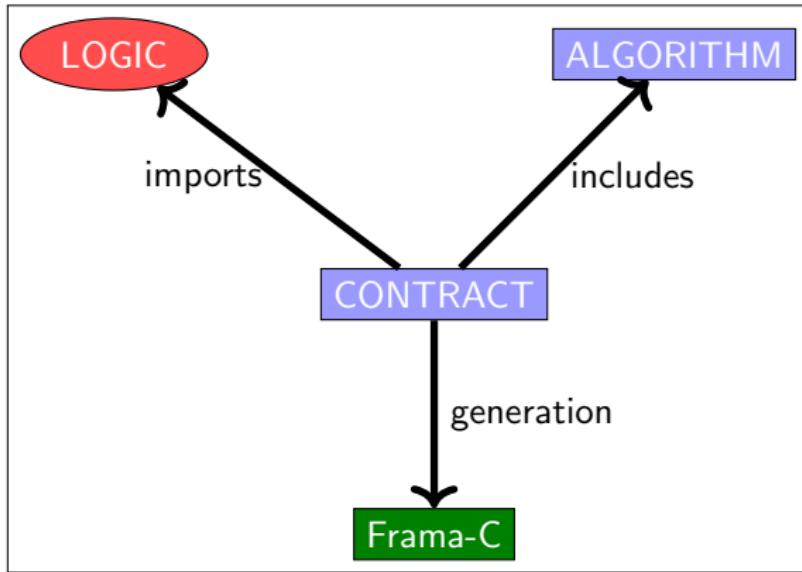
(facctrorial how)

Listing 7 – 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;
}
```

- ▶ 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 `mathfact(n)`.



Listing 8 – 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 9 – project-divers/structures.h

```
#ifndef _STRUCTURE_H

struct s {
    int q;
    int r;
};

#endif
```

(Specification)

Listing 10 – 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
```

(Algorithm)

Listing 11 – 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 12 – loop.c

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

```

(Invariant de boucle)

Listing 13 – 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 Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

2 Generation of Verification Conditions

WP calculus in Frama-c

First annotation

Second annotation

3 Memory Models in Frama-c

4 Logic Specification

5 Organisation of the verification process

6 Conclusion

Initial value of x \old(x)

- ▶ \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 14 – 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 15 – 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;
}
```


(autre label)

Listing 16 – 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 17 – 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 Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

2 Generation of Verification Conditions

WP calculus in Frama-c

First annotation

Second annotation

3 Memory Models in Frama-c

4 Logic Specification

5 Organisation of the verification process

6 Conclusion

- ▶ 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 18 – 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 19 – 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;
}
```

1 Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

2 Generation of Verification Conditions

WP calculus in Frama-c

First annotation

Second annotation

3 Memory Models in Frama-c

4 Logic Specification

5 Organisation of the verification process

6 Conclusion

1 Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

2 Generation of Verification Conditions

WP calculus in Frama-c

First annotation

Second annotation

3 Memory Models in Frama-c

4 Logic Specification

5 Organisation of the verification process

6 Conclusion

Listing 20 – an1.c

```
//@ assert I1 : P(x);
x = e(x);
//@ assert I2 : Q(x);
```

Listing 21 – an1.c

```
//@ assert I1 : P(x);  
x = e(x);  
//@ assert I2 : Q(x);
```

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

Listing 22 – an1.c

```
//@ assert I1 : P(x);  
x = e(x);  
//@ assert I2 : Q(x);
```

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

Listing 23 – an1.c

```
//@ assert I1 : P(x);
  x = e(x);
//@ assert I2 : 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 occurrences of x by $x1$)

Listing 24 – an1.c

```
//@ assert I1 : P(x);
  x = e(x);
//@ assert I2 : 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 occurrences of x by $x1$)
- ▶ $P(x1) \wedge x = e(x1) \Rightarrow Q(x)$

Listing 25 – an1.c

```
//@ assert I1 : P(x);  
x = e(x);  
//@ assert I2 : 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 occurrences of x by $x1$)
- ▶ $P(x1) \wedge x = e(x1) \Rightarrow Q(x)$
- ▶ $P(x1) \wedge x = e(x1) \Rightarrow Q(x)$

Listing 26 – an1.c

```
//@ assert I1 : P(x);  
x = e(x);  
//@ assert I2 : 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 occurrences of x by $x1$)
- ▶ $P(x1) \wedge x = e(x1) \Rightarrow Q(x)$
- ▶ $P(x1) \wedge x = e(x1) \Rightarrow Q(x)$
- ▶ $\vdash P(x1) \wedge x = e(x1) \Rightarrow Q(x)$

Listing 27 – an1.c

```
//@ assert I1 : P(x);
  x = e(x);
//@ assert I2 : 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 occurrences of x by $x1$)
- ▶ $P(x1) \wedge x = e(x1) \Rightarrow Q(x)$
- ▶ $P(x1) \wedge x = e(x1) \Rightarrow Q(x)$
- ▶ $\vdash P(x1) \wedge x = e(x1) \Rightarrow Q(x)$
- ▶ $P(x1) \wedge x = e(x1) \vdash Q(x)$

Listing 28 – an1.c

```
//@ assert I1 : P(x);  
  x = e(x);  
//@ assert I2 : 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 occurrences 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: $Q(x)$

1 Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

2 Generation of Verification Conditions

WP calculus in Frama-c

First annotation

Second annotation

3 Memory Models in Frama-c

4 Logic Specification

5 Organisation of the verification process

6 Conclusion

Listing 29 – an1.c

```
void ex(void) {
    int x=12,y=24;
    /*@ assert l1: 2*x == y;
    x = x+1;
    /*@ assert l2: y == 2*(x-1);
}
```

```
[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
```

Annotation simple (I)

```
Goal Assertion 'l1' (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
```

```
-----  
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
```

```
[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
```

1 Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

2 Generation of Verification Conditions

WP calculus in Frama-c

First annotation

Second annotation

3 Memory Models in Frama-c

4 Logic Specification

5 Organisation of the verification process

6 Conclusion

Listing 30 – an2.c

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

```
[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
```

Annotation simple (ii)

```
Goal Assertion 'l1' (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
```

Annotation simple (ii)

Goal Assertion 'l2' (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 'l1' *)
  Have: (2 * x_1) = y.
  Have: (1 + x_1) = x.
}
Prove: (2 + y) = (2 * x).
Prover Qed returns Valid
```

Annotation simple (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 31 – an2bis.c

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

1 Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

2 Generation of Verification Conditions

WP calculus in Frama-c

First annotation

Second annotation

3 Memory Models in Frama-c

4 Logic Specification

5 Organisation of the verification process

6 Conclusion

- ▶ Hoare Model : `-wp hoare` is the option of `frama-c`
- ▶ Typed Model : default model is typed model

- ▶ It simply maps each C variable to one pure logical variable.
- ▶ Heap cannot be represented in this model, and expressions such as `*p` cannot be translated at all.
- ▶ You can still represent pointer values, but you cannot read or write the heap through pointers.

- ▶ The default model for WP plug-in.
- ▶ Heap values are stored in several separated global arrays, one for each atomic type (integers, floats, pointers) and an additional one for memory allocation.
- ▶ Pointer values are translated into an index into these arrays.
- ▶ all C integer types are represented by mathematical integers and each pointer type to a given type is represented by a specific logical abstract datatype.

1 Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

2 Generation of Verification Conditions

WP calculus in Frama-c

First annotation

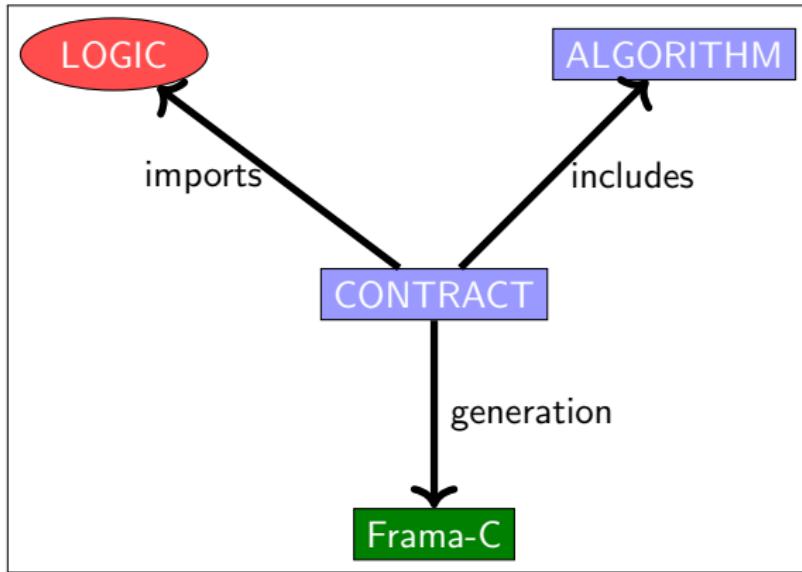
Second annotation

3 Memory Models in Frama-c

4 Logic Specification

5 Organisation of the verification process

6 Conclusion



- ▶ predicate

Predicates - Logic - Lemma (1)

(Predicate)

Listing 32 – project-divers/predicate1.c

```
/*@ predicate is_positive(integer x) = x > 0; */
/*@ logic integer get_sign(real x) = @ x > 0.0?1:(x < 0.0?-1:0);
*/
/*@ logic integer max(int x, int y) = x>=y?x:y;
*/
```

(Lemma)

Listing 33 – project-divers/lemma1.c

```
/*@ lemma div_mul_identity:
@ \forall real x, real y; y != 0.0 => y*(x/y) == x; @*/
/*@ lemma div_qr:
@ \forall int a, int b; a >= 0 && b > 0 =>
\exists int q, int r; a == b*q + r && 0 <= r && r < b; @*/
```

Predicates - Logic - Lemma (2)

(Definition of fibonacci function)

Listing 34 – project-divers/predicate2.c

```
/*@ axiomatic mathfibonacci{
    @ logic integer mathfib(integer n);
    @ axiom mathfib0: mathfib(0) == 1;
    @ axiom mathfib1: mathfib(1) == 1;
    @ axiom mathfibrec: \forall integer n; n > 1
        \implies mathfib(n) == mathfib(n-1)+mathfib(n-2);
    @ } */
```

(Definition of gcd)

Listing 35 – project-divers/predicate3.c

```
/*@ inductive is_gcd(integer a, integer b, integer d) {
    @ case gcd_zero:
        @ \forall integer n; is_gcd(n,0,n);
    @ case gcd_succ:
        @ \forall integer a,b,d; is_gcd(b, a % b, d) \implies is_gcd(a,b,d); @}
    @*/
```

(Definition of function odd/even)

Listing 36 – project-divers/predicate4.c

```
//@ predicate pair(integer x) = (x/2)*2==x;
//@ predicate impair(integer x) = (x/2)*2!=x;
//@ lemma ex: \forall integer a,b; a < b => 2*a < 2*b;

/*@ inductive is_gcd(integer a,integer b , integer c) {
  case zero: \forall integer n; is_gcd(n,0,n);
  case un: \forall integer u,v,w; u >= v => is_gcd(u-v,v,w);
  case deux: \forall integer u,v,w; u < v => is_gcd(u,v-u,w);
}
*/
```

- ▶ The termination is proved by showing that each loop terminates.
- ▶ Any loop is characterized by an expression $\text{expvariant}(x)$ called variant which should decrease each execution of the body :
$$\forall x_1, x_2. b(x_1) \wedge x_1 \xrightarrow{S} x_2 \Rightarrow \text{expvariant}(x_1) > \text{expvariant}(x_2)$$

(Variant)

Listing 37 – project-divers/variant1.c

```
/*@ requires n > 0;
terminates n > 0;

ensures \result == 0;
*/
int code(int n) {
    int x = n;
    /*@ loop invariant x >= 0 && x <= n;
       loop assigns x;
       loop variant x;
    */
    while (x != 0) {
        x = x - 1;
    };
    return x;
}
```

Example of loop variant

(Variant)

Listing 38 – project-divers/variant3.c

```
int f() {
    int x = 0;
    int y = 10;
    /*@
     * loop invariant
     * 0 <= x < 11 && x+y == 10;
     * loop variant y;
     */
    while (y > 0) {
        x++;
        y--;
    }
    return 0;
}
```

(Variant)

Listing 39 – project-divers/variant4.c

```
g/*@ requires n <= 12;
   @ decreases n;
   @*/
int fact(int n){
    if (n <= 1) return 1;
    return n*fact(n-1);
}
```

- ▶ – lemma : 1 VC
- ▶ – axiom : no VC (admitted with no proof)
- ▶ – ensures : 1 VC
- ▶ – exits : 1 VC
- ▶ – disjoint : 1 VC
- ▶ – complete : 1 VC
- ▶ – requires : 1 VC for each call
- ▶ – terminates : 1 VC for each call, 1 VC for each loop without "loop variant"
- ▶ – decreases : 1 VC for each recursive call
- ▶ – assigns : 1 VC for each assigned lvalue
- ▶ – admit : no VC (admitted with no proof)
- ▶ – assert/check : 1 VC
- ▶ – loop invariant : 2 VCs (established, preserved)
- ▶ – loop variant (integer) : 2 VCs (positive, decreasing)
- ▶ – loop variant (general measure) : 1 VC (the measure is assumed to be well-founded) – loop assigns : 1 VC for each assigned lvalue within the loop

1 Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

2 Generation of Verification Conditions

WP calculus in Frama-c

First annotation

Second annotation

3 Memory Models in Frama-c

4 Logic Specification

5 Organisation of the verification process

6 Conclusion

- ▶ Defining the mathematical function to compute *mathf*
- ▶ Stating the postcondition using the mathematical function
- ▶ Evaluating the inductive sequence u_i computing the function *mathf*
- ▶ $\forall i \in \mathbb{N} : u_i = \text{mathf}(i)$
- ▶ Evaluating relationship among variables.

Factorial function

(power2.h)

Listing 40 – project-powers/power21.h

```
#ifndef _A_H
#define _A_H
// Definition of the mathematical function mathpower2
/*@ axiomatic mathpower2 {
    @ logic integer mathpower2(integer n);
    @ axiom mathpower2_0: mathpower2(0) == 0;
    @ axiom mathpower2_rec: \forall integer n; n > 0
        => mathpower2(n) == mathpower2(n-1) + n+n+1;
    @ } */
/*@ axiomatic matheven {
    @ logic integer matheven(integer n);
    @ axiom matheven_0: matheven(0) == 0;
    @ axiom matheven_rec: \forall integer n; n > 0
        => matheven(n) == matheven(n-1) + 2;
    @ } */
// We define v and w in a one shot axiomatic definition
/*@ axiomatic vw {
    @ logic integer v(integer n);
    @ logic integer w(integer n);
    @ axiom v_0: v(0) == 0;
    @ axiom w_0: w(0) == 0;
    @ axiom v_rec: \forall integer n; n > 0
        => v(n) == v(n-1) +n+n+1 && w(n) == w(n-1) + 2;
    @ } */
```

(power2.h)

Listing 41 – project-powers/power22.h

```
/*@ lemma propw:  
@ \forall int n; n >= 0 ==> w(n) == n+n; @*/  
/*@ lemma propv:  
@ \forall int n; n >= 0 ==> v(n) == n*n; @*/  
/*@ lemma prop1:  
@ \forall int n; n >= 0 ==> matheven(n) == n+n; @*/  
/*@ lemma prop2:  
@ \forall int n; n >= 0 ==> mathpower2(n) == n*n; @*/  
/*@ axiomatic auxmath {  
    @ lemma rule1: \forall int n; n >0 ==> n*n == (n-1)*(n-1)+2*(n-1)+1;  
    @ } */  
/*@ requires 0 <= x;  
    assigns \nothing;  
    ensures \result == x*x;  
*/  
int power2(int x);  
#endif
```

(power2.h)

Listing 42 – project-powers/power2.c

```
#include <limits.h>
#include "power2.h"

int power2(int x)
{int r,k,cv,cw,or,ok,ocv,ocw;
r=0;k=0;cv=0;cw=0;or=0;ok=k;ocv=cv;ocw=cw;
/*@ loop invariant 0 <= cv && 0 <= cw && 0 <= k;
 @ loop invariant cv == k*k;
 @ loop invariant k <= x;
 @ loop invariant cw == 2*k;
 @ loop invariant 4*cv == cw*cw;
 @ loop assigns k,cv,cw,or,ok,ocv,ocw; */
while (k<x)
{
    ok=k;ocv=cv;ocw=cw;
    k=ok+1;
    cv=ocv+ocw+1;
    cw=ocw+2;

}
r=cv;
return(r);
}
```

1 Contracts

Extending C programming language by contracts

Playing with variables

Ghost Variables

2 Generation of Verification Conditions

WP calculus in Frama-c

First annotation

Second annotation

3 Memory Models in Frama-c

4 Logic Specification

5 Organisation of the verification process

6 Conclusion

- ▶ Plugin -rte adds specific assertions for each variable
- ▶ Systematic checking of RTE.

- ▶ Defining domain properties (axioms, lemmas, proofs)
- ▶ Defining loop invariants (typing, equation, ...)
- ▶ Analyzing inductive properties
- ▶ Identifying inputs (*requires*) and outputs (*ensures*)