☺

Lectures on

Automated verification of C programs extended by annotations and contracts

Dominique Méry
Université de Lorraine & LORIA

ISCAS Beijing (July 14, 2026)

Summary

1. Introduction of notations
   - First example on proving versus testing
   - Second example on proving versus testing
   - Partial correctness but also runtime error
2. Correctness by contract
   - WP calculus
   - Simple example 1
3. Writing a contract
   - " A la Floyd" Verification Conditions for Contract
   - "A la Hoare" Verification Conditions for Contract
4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c

Summary

4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c
9. Contracts
   - Extending C programming language by contracts
   - Playing with variables
   - Ghost Variables
   - Labels
10. Generation of Verification Conditions
    - WP calculus in Frama-c
    - First annotation
    - Second annotation
11. Memory Models in Frama-c
12. Logic Specification
13. Organisation of the verification process
14. Conclusion

Current Summary

1. Introduction of notations
   - First example on proving versus testing
   - Second example on proving versus testing
   - Partial correctness but also runtime error
2. Correctness by contract
   - WP calculus
   - Simple example 1
3. Writing a contract
   - " A la Floyd" Verification Conditions for Contract
   - "A la Hoare" Verification Conditions for Contract
4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c

Summary

4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c
9. Contracts
   - Extending C programming language by contracts
   - Playing with variables
   - Ghost Variables
   - Labels
10. Generation of Verification Conditions
    - WP calculus in Frama-c
    - First annotation
    - Second annotation
11. Memory Models in Frama-c
12. Logic Specification
13. Organisation of the verification process
14. Conclusion

Example anchina0.c

Listing 1 – anchina0.c

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

Using gcc

```
#include <stdio.h>
#include <assert.h>

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

Using the tool Frama-C

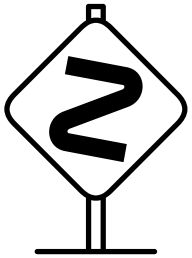
```
[kernel] Parsing anchina0.c (with preprocessing)
[wp] Running WP plugin...
[wp] Warning: Missing RTE guards
[wp] 2 goals scheduled
[wp] Proved goals:    4 / 4
  Terminating:      1
  Unreachable:      1
  Qed:              2
```

Example anchina00.c

Listing 2 – anchina00.c

```
/*@ requires \true;
  @ assigns \nothing;
  @ ensures \true;
*/
void ex(int x,int y) {
  //@ assert l1: 2*x == y;
  x = x+1;
  //@ assert l2:  y == 2*(x-1);
}
```

```
[kernel] Parsing anchina00.c (with preprocessing)
[wp] Running WP plugin...
[wp] Warning: Missing RTE guards
[wp] 4 goals scheduled
[wp] [Timeout] typed_ex_assert_l1 (Alt-Ergo)
[wp] Proved goals:    5 / 6
  Terminating:      1
  Unreachable:      1
  Qed:              3
```



One proved goal is not proved at all !

Using gcc

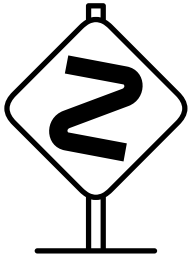
```
#include <stdio.h>
#include <assert.h>

/*@ requires \true;
  @ assigns \nothing;
  @ ensures \true;
*/
void  ex(int x,int y) {
    assert(2*x == y);    //
    //@ assert l1: 2*x == y;
  x = x+1;
  assert( y == 2*(x-1));    //
  //@ assert l2:  y == 2*(x-1);
}
```

Using gcc

```
void main() {

  int a=12,b=2;
  ex(a,b);
}
```

Using gcc

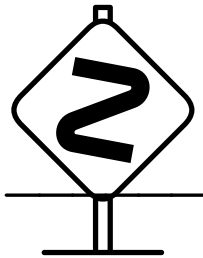
```
void main() {

  int a=12,b=2;
  ex(a,b);
}
```



Abort trap : 6

Using gcc

Frama-C produces the following information



Frama-C produces the following information

```
Goal Assertion 'l1' (file anchina00.c, line 6):
Assume {
  Type: is_sint32(x) /\ is_sint32(y).
  (* Frame In *)
  Have: (ta_x_0=false).
}
Prove: (2 * x) = y.
Prover Alt-Ergo 2.6.0 returns Timeout (2s)
```

Modifying the precondition

```
/*@ requires 2*x == y;
  @ assigns \nothing;
  @ ensures \true;
*/
void ex(int x,int y) {
  //@ assert l1: 2*x == y;
  x = x+1;
  //@ assert l2:  y == 2*(x-1);
}
```

Modifying the precondition

```
/*@ requires 2*x == y;
  @ assigns \nothing;
  @ ensures \true;
*/
void ex(int x,int y) {
  //@ assert l1: 2*x == y;
  x = x+1;
  //@ assert l2:  y == 2*(x-1);
}
```

```
[kernel] Parsing anchina000.c (with preprocessing)
[wp] Running WP plugin...
[wp] Warning: Missing RTE guards
[wp] 4 goals scheduled
[wp] Proved goals:    6 / 6
  Terminating:      1
  Unreachable:      1
  Qed:              4
```

bug0.c

Listing 3 – Bug bug0

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int x, y;
    // Seed the random number generator with the current time
    srand(time(NULL));
    // Generate a random number between 1 and 100
    x = rand() % 100 + 1;
    // Perform some calculations
    y = x / (100 - x);
    printf("Result: %d\n", y);
    return 0;
}
```

bug0.c

Listing 4 – Bug bug0

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int x, y;
    // Seed the random number generator with the current time
    srand(time(NULL));
    // Generate a random number between 1 and 100
    x = rand() % 100 + 1;
    // Perform some calculations
    y = x / (100 - x);
    printf("Result: %d\n", y);
    return 0;
}
```

The statement `y = x / (100 - x);` may lead to a messsage `Floating point exception: 8` . The Floating point exception : 8 message indicates that a program has triggered an arithmetic exception at system level (SIGFPE signal), causing it to stop abruptly.

bug00.c

Listing 5 – Bug bug00

```
// Heisenbug
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
//
int main() {
  int x, y,i=0;
    for (i = 0; i <= 1000000; i++) {
    // Seed the random number generator with the current time
    srand(time(NULL));
    // Generate a random number between 1 and 100
    x = rand() % 100 + 1;
        printf("Result: x=--%d\n",x);
    // Perform some calculations
    y = x / (100 - x);
    printf("Result: i=%d--and y=%d\n",i, y);
    }
    return 0;
}
```

bug00.c

Listing 6 – Bug bug00

```
// Heisenbug
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
//
int main() {
  int x, y,i=0;
    for (i = 0; i <= 1000000; i++) {
    // Seed the random number generator with the current time
    srand(time(NULL));
    // Generate a random number between 1 and 100
    x = rand() % 100 + 1;
        printf("Result: x=--%d\n",x);
    // Perform some calculations
    y = x / (100 - x);
    printf("Result: i=%d--and y=%d\n",i, y);
    }
    return 0;
}
```

The code in bug0.c is repeated 1,000,000 times and does not detect any division by zero.

Detecting overflows of computations

Listing 7 – Function average

```
#include <stdio.h>
#include <limits.h>
int average(int a,int b)
{
  return((a+b)/2);
}

int main()
{
  int x,y;
  x=INT_MAX;y=INT_MAX;
  printf("Average--for-%d-and-%d-is-%d\n",x,y,
          average(x,y));
  return 0;
}
```

Execution

Execution produces a result

```
Average  for 2147483647 and 2147483647 is -1
```

Execution

Execution produces a result

```
Average  for 2147483647 and 2147483647 is -1
```

Using frama-c produces a required annotation

```
int average(int a, int b)
{
  int __retres;
  /*@ assert rte: signed_overflow: -2147483648 <= a + b; */
  /*@ assert rte: signed_overflow: a + b <= 2147483647; */
  __retres = (a + b) / 2;
  return __retres;
}
```

Annotated Example

Listing 8 – Function average.....

```
#include <stdio.h>
#include <limits.h>
/*@  requires  0 <= a;
     requires a <= INT_MAX ;
     requires  0 <= b;
     requires b <= INT_MAX ;
     requires  0 <= a+b;
     requires a+b <= INT_MAX ;
     ensures \result <= INT_MAX;
*/
int average(int a,int b)
{
  return((a+b)/2);
}

int main()
{
  int x,y;
  x=INT_MAX / 2;y=INT_MAX / 2;
  //  printf("Average  for %d and %d is %d\n",x,y,
  //      );
  return average(x,y);
}
```

Current Summary

1. Introduction of notations
   - First example on proving versus testing
   - Second example on proving versus testing
   - Partial correctness but also runtime error
2. Correctness by contract
   - WP calculus
   - Simple example 1
3. Writing a contract
   - " A la Floyd" Verification Conditions for Contract
   - "A la Hoare" Verification Conditions for Contract
4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c

Summary

4 Examples
- Simple annotation (I)
- Simple annotation (II)
- Simple annotation(III)
- Simple annotation(IV)

5 Mechanizing the contract checking

6 Transforming predicates
- Hoare Logic for PC
- Examples in ACSL
- Définition et propriétés du calcul wp

7 Using predicate transformers for checking contracts (next episod)

8 Memory Models in Frama-c

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

10 Generation of Verification Conditions
- WP calculus in Frama-c
- First annotation
- Second annotation

11 Memory Models in Frama-c

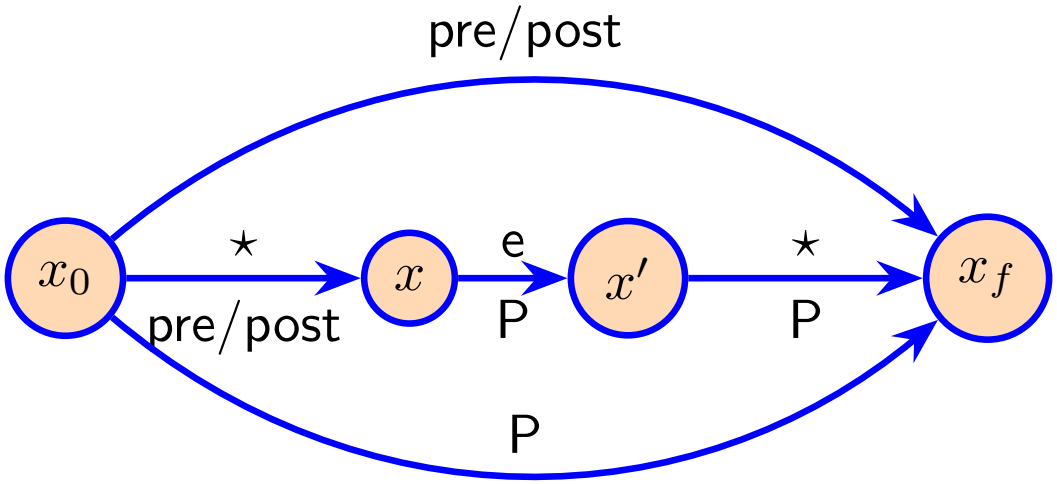
12 Logic Specification

13 Organisation of the verification process

14 Conclusion

Frpm pre/post to algorithm(s)

Refining a pre/post specification into a sequential algorithm



Verification of contract (I)

A program P *satisfies* a contract (pre,post) :

- P transforms a variable x from an initial value x_0 and produces a final value x_f : $x_0 \xrightarrow{P} x_f$
- x_0 satisfait pre : pre(x_0) and x_f satisfait post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{P} x_f \Rightarrow$ post(x_0, x_f)

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X

```
begin
0 : P_0(x_0, x)
instruction_0
...
i : P_i(x_0, x)
...
instruction_{f-1}
f : P_f(x_0, x)
end
```

Floyd style

- $pre(x_0) \wedge x = x_0 \Rightarrow P_0(x_0, x)$
- $pre(x_0) \wedge P_f(x_0, x) \Rightarrow post(x_0, x)$
- For each pair ℓ, ℓ' such that $\ell \longrightarrow \ell'$, one checks that foa any value $x, x' \in \textsc{Memory}$

$$\left(\begin{array}{l} \left(\begin{array}{l} pre(x_0) \wedge P_\ell(x_0, x)) \\ \wedge cond_{\ell,\ell'}(x) \wedge x' = f_{\ell,\ell'}(x) \end{array} \right) \\ \Rightarrow P_{\ell'}(x_0, x') \end{array} \right),$$

Verification du contract (II)

A program P *satisfies* a contract (pre,post) :

- P transforms a variable x from an initial value x_0 and produces a final value x_f : $x_0 \xrightarrow{P} x_f$
- x_0 satisfait pre : pre(x_0) and x_f satisfait post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{P} x_f \Rightarrow$ post(x_0, x_f)

requires $pre(x_0)$
ensures $post(x_0, x_f)$ **Hoare style**
variables X

```
begin
0 : P_0(x_0, x)
instruction_0
...
i : P_i(x_0, x)
...
instruction_{f-1}
f : P_f(x_0, x)
end
```

Verification du contract (II)

A program P *satisfies* a contract (pre,post) :

- P transforms a variable x from an initial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\mathsf{P}} x_f$
- x_0 satisfait pre : pre(x_0) and x_f satisfait post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow$ post(x_0, x_f)

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X

```
begin
0 : P_0(x_0, x)
instruction_0
...
i : P_i(x_0, x)
...
instruction_{f-1}
f : P_f(x_0, x)
end
```

Hoare style

- $\forall x_f, x_0.\text{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f)$

Verification du contract (II)

A program P *satisfies* a contract (pre,post) :

- P transforms a variable x from an initial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\mathsf{P}} x_f$
- x_0 satisfait pre : pre(x_0) and x_f satisfait post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow$ post(x_0, x_f)

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X
begin
$0 : P_0(x_0, x)$
$\texttt{instruction}_0$
...
$i : P_i(x_0, x)$
...
$\texttt{instruction}_{f-1}$
$f : P_f(x_0, x)$
end

Hoare style

- $\forall x_f, x_0.\text{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f)$
- $\forall x_f, x_0.\text{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f))$

Verification du contract (II)

A program P *satisfies* a contract (pre,post) :

- P transforms a variable x from an initial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\mathsf{P}} x_f$
- x_0 satisfait pre : pre(x_0) and x_f satisfait post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow$ post(x_0, x_f)

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X
begin
$0 : P_0(x_0, x)$
$\texttt{instruction}_0$
...
$i : P_i(x_0, x)$
...
$\texttt{instruction}_{f-1}$
$f : P_f(x_0, x)$
end

Hoare style

- $\forall x_f, x_0.\text{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f)$
- $\forall x_f, x_0.\text{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f))$
- $\forall x_0.\text{pre}(x_0) \Rightarrow \forall x_f.(x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f))$

Verification du contract (II)

A program P *satisfies* a contract (pre,post) :

- P transforms a variable x from an initial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\mathsf{P}} x_f$
- x_0 satisfait pre : pre(x_0) and x_f satisfait post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow$ post(x_0, x_f)

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X
begin
$0 : P_0(x_0, x)$
$\texttt{instruction}_0$
...
$i : P_i(x_0, x)$
...
$\texttt{instruction}_{f-1}$
$f : P_f(x_0, x)$
end

Hoare style

- $\forall x_f, x_0.\text{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f)$
- $\forall x_f, x_0.\text{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f))$
- $\forall x_0.\text{pre}(x_0) \Rightarrow \forall x_f.(x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f))$
- $\forall x_0.\text{pre}(x_0) \Rightarrow \forall x.(x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \text{post}(x_0, x))$

Verification du contract (II)

A program P *satisfies* a contract (pre,post) :

- P transforms a variable x from an initial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\mathsf{P}} x_f$
- x_0 satisfait pre : pre(x_0) and x_f satisfait post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow$ post(x_0, x_f)

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X

```
begin
0 : P_0(x_0, x)
instruction_0
...
i : P_i(x_0, x)
...
instruction_{f-1}
f : P_f(x_0, x)
end
```

Hoare style

- $\forall x_f, x_0.\text{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f)$
- $\forall x_f, x_0.\text{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f))$
- $\forall x_0.\text{pre}(x_0) \Rightarrow \forall x_f.(x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f))$
- $\forall x_0.\text{pre}(x_0) \Rightarrow \forall x.(x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \text{post}(x_0, x))$
- $\forall x_0.\text{pre}(x_0) \Rightarrow WLP(P)(\text{post}(x_0, x))$

$//@ \quad assert \quad \ell_1 : P(x);$
$\mathtt{x} = \mathtt{e}(\mathtt{x});$
$//@ \quad assert \quad \ell_2 : Q(x);$

- $\forall x, x'.P(x) \wedge x' = e(x) \Rightarrow Q(x')$
- $\vdash \forall x, x'.P(x) \wedge x' = e(x) \Rightarrow Q(x')$
- $\vdash \forall x1, x.P(x1) \wedge x = e(x1) \Rightarrow Q(x)$
- $\vdash P(x1) \wedge x = e(x1) \Rightarrow Q(x)$
- $P(x1), x = e(x1) \vdash Q(x)$

Example anchina0.c

Listing 9 – anchina0.c

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

Example v1 (1/4)

```
[kernel] Parsing anchina0.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
```

Example v1 (2/4)

```
  Function ex
```

```
Goal Assertion 'l1' (file anchina0.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
```

Example v1 (3/4)

```
Goal Assertion 'l2' (file anchina0.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
```

Example v1 (4/4)

```
[wp:pedantic-assigns] anchina0.c:1: Warning:
  No 'assigns' specification for function 'ex'.
  Callers assumptions might be imprecise.
```

Example v1

Listing 10 – 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;
}
```

Example v1 with wp calculus

Listing 11 – an2bis.c

```
void ex(void) {
 //@ assert l1pre: 2*12 == 24;
  int x=12,y=24;
  //@ assert l1: 2*x == y;
  // =>
   //@ assert l2pre:  y == 2*(x+1-1);
  x = x+1;
  //@ assert l2:  y == 2*(x-1);
  //  =>
  //@ assert l3pre:  y+6 == 2*(x+2);
   x = x+2;
  //@ assert l3:  y+6 == 2*x;
}
```

Example v1 (1/6)

```
[kernel] Parsing anchina1.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
```

F Example v1 (2/6)

```
Function ex
```

```
Goal Assertion 'l1' (file anchina1.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
```

F Example v1 (3/6)

```
Goal Assertion 'l2' (file anchina1.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
```

F Example v1 (4/6)

```
Goal Assertion 'I3' (file anchina1.c, line 7):
Assume {
  Type: is_sint32(x) /\ is_sint32(x_1) /\ is_sint32(x_2)
  (* Initializer *)
  Init: x_2 = 12.
  (* Initializer *)
  Init: y = 24.
  (* Assertion 'I1' *)
  Have: (2 * x_2) = y.
  Have: (1 + x_2) = x_1.
  (* Assertion 'I2' *)
  Have: (2 + y) = (2 * x_1).
  Have: (2 + x_1) = x.
}
Prove: (6 + y) = (2 * x).
Prover Qed returns Valid
```

F Example v1 (5/6)

```
Goal Assertion 'l3' (file anchina1.c, line 7):
Assume {
  Type: is_sint32(x) /\ is_sint32(x_1) /\ is_sint32(x_2)
  (* 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
```

F Example v1 (6/6)

```
[wp:pedantic-assigns] anchina1.c:1: Warning:
  No 'assigns' specification for function 'ex'.
  Callers assumptions might be imprecise.
```

Current Summary

1. Introduction of notations
   - First example on proving versus testing
   - Second example on proving versus testing
   - Partial correctness but also runtime error
2. Correctness by contract
   - WP calculus
   - Simple example 1
3. Writing a contract
   - " A la Floyd" Verification Conditions for Contract
   - "A la Hoare" Verification Conditions for Contract
4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c

Summary

4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c
9. Contracts
   - Extending C programming language by contracts
   - Playing with variables
   - Ghost Variables
   - Labels
10. Generation of Verification Conditions
    - WP calculus in Frama-c
    - First annotation
    - Second annotation
11. Memory Models in Frama-c
12. Logic Specification
13. Organisation of the verification process
14. Conclusion

Writing a contract

$\ell_1 : x = 3 \ \wedge \ y = z{+}x \ \wedge z = 2{\cdot}x$
$y := z{+}x$
$\ell_2 : x = 3 \ \wedge \ y = x{+}6$

A simple assignment is annotated and we have to check that the annotation is correct.

We define the contract as follows :

variables x, y, z
requires $x0 = 3 \wedge y0 = z0{+}x0 \wedge z0 = 2.x0$
ensures $x_f = 3 \wedge y_f = x_f{+}6$

```
begin
ℓ1 : x = 3 ∧ y = z+x ∧ z = 2·x
y := z+x
ℓ2 : x = 3 ∧ y = x+6
end
```

"A la Floyd" Verification Conditions for Contract

We derive the following assertions from annotations :

- $pre(x_0, y_0, z_0) \stackrel{def}{=} x0 = 3 \wedge y0 = z0{+}x0 \wedge z0 = 2.x0$
- $prepost(x_0, y_0, z_0, x, y, z) \stackrel{def}{=} x = 3 \ \wedge \ y = x{+}6$
- $Q_1(x_0, y_0, z_0, x, y, z) \stackrel{def}{=} x = 3 \ \wedge \ y = z{+}x \ \wedge z = 2{\cdot}x$
- $Q_2(x_0, y_0, z_0, x, y, z) \stackrel{def}{=} x = 3 \ \wedge \ y = x{+}6$

"A la Floyd" Verification Conditions for Contract

We derive the following assertions from annotations :

- $pre(x_0, y_0, z_0) \stackrel{def}{=} x0 = 3 \wedge y0 = z0{+}x0 \wedge z0 = 2.x0$
- $prepost(x_0, y_0, z_0, x, y, z) \stackrel{def}{=} x = 3 \ \wedge \ y = x{+}6$
- $Q_1(x_0, y_0, z_0, x, y, z) \stackrel{def}{=} x = 3 \ \wedge \ y = z{+}x \ \wedge z = 2{\cdot}x$
- $Q_2(x_0, y_0, z_0, x, y, z) \stackrel{def}{=} x = 3 \ \wedge \ y = x{+}6$

Three verification conditions for checking the contract :

- (init) $pre(x_0, y_0, z_0) \wedge (x, y, z) = (x_0, y_0, z_0) \Rightarrow Q_1(x_0, y_0, z_0, x, y, z)$
- (concl) $pre(v_0) \wedge Q_2(x_0, y_0, z_0, x, y, z) \Rightarrow prepost(x_0, y_0, z_0, x, y, z)$
- (induct)
  $pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') =$
  $(x, z{+}x, z) \Rightarrow Q_2(x_0, y_0, z_0, x', y', z')$

Proof for the step induct

Proof for the step induct

- $pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') = (x, z{+}x, z) \Rightarrow Q_2(x_0, y_0, z_0, x', y', z')$

Proof for the step induct

- $pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') = (x, z{+}x, z) \Rightarrow Q_2(x_0, y_0, z_0, x', y', z')$
- $\vdash pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') = (x, z{+}x, z) \Rightarrow Q_2(x_0, y_0, z_0, x', y', z')$

Proof for the step induct

- $pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') = (x, z{+}x, z) \Rightarrow Q_2(x_0, y_0, z_0, x', y', z')$
- $\vdash pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') = (x, z{+}x, z) \Rightarrow Q_2(x_0, y_0, z_0, x', y', z')$
- $pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') = (x, z{+}x, z) \vdash Q_2(x_0, y_0, z_0, x', y', z')$

Proof for the step induct

- $pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') = (x, z{+}x, z) \Rightarrow Q_2(x_0, y_0, z_0, x', y', z')$
- $\vdash pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') = (x, z{+}x, z) \Rightarrow Q_2(x_0, y_0, z_0, x', y', z')$
- $pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') = (x, z{+}x, z) \vdash Q_2(x_0, y_0, z_0, x', y', z')$
- $pre(x_0, y_0, z_0), Q_1(x_0, y_0, z_0, x, y, z), TRUE, (x', y', z') = (x, z{+}x, z) \vdash Q_2(x_0, y_0, z_0, x', y', z')$

Proof for the step induct

- $pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') = (x, z{+}x, z) \Rightarrow Q_2(x_0, y_0, z_0, x', y', z')$
- $\vdash pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') = (x, z{+}x, z) \Rightarrow Q_2(x_0, y_0, z_0, x', y', z')$
- $pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') = (x, z{+}x, z) \vdash Q_2(x_0, y_0, z_0, x', y', z')$
- $pre(x_0, y_0, z_0), Q_1(x_0, y_0, z_0, x, y, z), TRUE, (x', y', z') = (x, z{+}x, z) \vdash Q_2(x_0, y_0, z_0, x', y', z')$
- $x0 = 3 \wedge y0 = z0{+}x0, z0 = 2.x0, Q_1(x_0, y_0, z_0, x, y, z), TRUE, (x', y', z') = (x, z{+}x, z) \vdash Q_2(x_0, y_0, z_0, x', y', z')$

Proof for the step induct

- $pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') = (x, z{+}x, z) \Rightarrow Q_2(x_0, y_0, z_0, x', y', z')$
- $\vdash pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') = (x, z{+}x, z) \Rightarrow Q_2(x_0, y_0, z_0, x', y', z')$
- $pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') = (x, z{+}x, z) \vdash Q_2(x_0, y_0, z_0, x', y', z')$
- $pre(x_0, y_0, z_0), Q_1(x_0, y_0, z_0, x, y, z), TRUE, (x', y', z') = (x, z{+}x, z) \vdash Q_2(x_0, y_0, z_0, x', y', z')$
- $x0 = 3 \wedge y0 = z0{+}x0, z0 = 2.x0, Q_1(x_0, y_0, z_0, x, y, z), TRUE, (x', y', z') = (x, z{+}x, z) \vdash Q_2(x_0, y_0, z_0, x', y', z')$
- $x0 = 3 \wedge y0 = z0{+}x0, z0 = 2.x0, x = 3 \ \wedge \ y = z{+}x \ \wedge z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash Q_2(x_0, y_0, z_0, x', y', z')$

Proof for the step induct

- $pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') = (x, z{+}x, z) \Rightarrow Q_2(x_0, y_0, z_0, x', y', z')$
- $\vdash pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') = (x, z{+}x, z) \Rightarrow Q_2(x_0, y_0, z_0, x', y', z')$
- $pre(x_0, y_0, z_0) \wedge Q_1(x_0, y_0, z_0, x, y, z) \wedge TRUE \wedge (x', y', z') = (x, z{+}x, z) \vdash Q_2(x_0, y_0, z_0, x', y', z')$
- $pre(x_0, y_0, z_0), Q_1(x_0, y_0, z_0, x, y, z), TRUE, (x', y', z') = (x, z{+}x, z) \vdash Q_2(x_0, y_0, z_0, x', y', z')$
- $x0 = 3 \wedge y0 = z0{+}x0, z0 = 2.x0, Q_1(x_0, y_0, z_0, x, y, z), TRUE, (x', y', z') = (x, z{+}x, z) \vdash Q_2(x_0, y_0, z_0, x', y', z')$
- $x0 = 3 \wedge y0 = z0{+}x0, z0 = 2.x0, x = 3 \ \wedge \ y = z{+}x \ \wedge z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash Q_2(x_0, y_0, z_0, x', y', z')$
- $x0 = 3 \wedge y0 = z0{+}x0, z0 = 2.x0, x = 3 \ \wedge \ y = z{+}x \ \wedge z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash x' = 3 \ \wedge \ y' = x'{+}6$

- $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash x' = 3 \ \wedge \ y' = x'{+}6$

- $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash x' = 3 \ \wedge \ y' = x'{+}6$
- $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash x' = 3$

- $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash x' = 3 \ \wedge \ y' = x'{+}6$
- $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash x' = 3$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash x' = 3$

- $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash x' = 3 \ \wedge \ y' = x'{+}6$
- $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash x' = 3$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash x' = 3$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash x = 3$

- $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash x' = 3 \ \wedge \ y' = x'{+}6$
- $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash x' = 3$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash x' = 3$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash x = 3$
  - $x = 3$ is an assumption in the left. The sequent is valid.

- $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x'{+}6$

- $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x'{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x'{+}6$

- $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x'{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x'{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x{+}6$

- $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x'{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x'{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash z{+}x = x{+}6$

- $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x'{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x'{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash z{+}x = x{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash 2.x{+}x = x{+}6$

- $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x'{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x'{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash z{+}x = x{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash 2.x{+}x = x{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash 2.3{+}3 = 3{+}6$

- $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x'{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x'{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash z{+}x = x{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash 2.x{+}x = x{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash 2.3{+}3 = 3{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash 9 = 9$

- $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x'{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x'{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash y' = x{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash z{+}x = x{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash 2.x{+}x = x{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash 2.3{+}3 = 3{+}6$
  - $x0 = 3, y0 = z0{+}x0, z0 = 2.x0, x = 3, y = z{+}x, z = 2{\cdot}x, TRUE, (x', y', z') = (x, z{+}x, z) \vdash 9 = 9$
  - Reflexivity of equality

Verification Condition for Assignment

Listing 12 – assert.c

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

Verification Condition for Assignment

Listing 13 – assert.c

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

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

Verification Condition for Assignment

Listing 14 – assert.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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 15 – assert.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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 16 – assert.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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 17 – assert.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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 18 – assert.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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 19 – assert.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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 20 – assert.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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:$Q(x)$
-

Current Summary

1. Introduction of notations
   - First example on proving versus testing
   - Second example on proving versus testing
   - Partial correctness but also runtime error
2. Correctness by contract
   - WP calculus
   - Simple example 1
3. Writing a contract
   - " A la Floyd" Verification Conditions for Contract
   - "A la Hoare" Verification Conditions for Contract
4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c

Summary

4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c
9. Contracts
   - Extending C programming language by contracts
   - Playing with variables
   - Ghost Variables
   - Labels
10. Generation of Verification Conditions
    - WP calculus in Frama-c
    - First annotation
    - Second annotation
11. Memory Models in Frama-c
12. Logic Specification
13. Organisation of the verification process
14. Conclusion

Simple annotation an1.c (I)

Listing 21 – an1.c

```
int main(void){
  signed long int x,y,z; //     int x,y,z;
  x = 1;
  /*@  assert x == 1; */
  y = 2;
  /*@ assert x == 1 && y == 2; */
  z = x *y;
  /*@  assert x == 1 && y == 1 && z == 2; */
  return 0;
  }
```

Example an1.c (1/4)

```
[kernel] Parsing an1.c (with preprocessing)
[wp] Running WP plugin...
[wp] Warning: Missing RTE guards
[wp] 3 goals scheduled
[wp] [Timeout] typed_main_assert_3 (Alt-Ergo)
[wp] Proved goals:    4 / 5
  Terminating:      1
  Unreachable:      1
  Qed:              2
```

Example an1.c (2/4)

```
  Timeout:          1
```

```
  Function main
```

```
Goal Assertion (file an1.c, line 4):
Assume { Type: is_sint64(x). Have: x = 1. }
Prove: x = 1.
Prover Qed returns Valid
```

```
Goal Assertion (file an1.c, line 6):
Assume {
  Type: is_sint64(x) /\ is_sint64(y).
```

Example an1.c (3/4)

```
  Type: is_sint64(x) /\ is_sint64(y).
  Have: x = 1.
  (* Assertion *)
  Have: x = 1.
  Have: y = 2.
}
Prove: (x = 1) /\ (y = 2).
Prover Qed returns Valid
```

```
Goal Assertion (file an1.c, line 8):
Assume {
  Type: is_sint64(x) /\ is_sint64(y) /\ is_sint64(z).
  Have: x = 1.
  (* Assertion *)
  Have: x = 1.
```

Example an1.c (4/4)

```
Have: x = 1.
Have: y = 2.
(* Assertion *)
Have: (x = 1) /\ (y = 2).
```

Simple annotation(II)

Listing 22 – 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;
}
```

Example an2.c (1/4)

```
[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
```

Example an2.c (2/4)

```
  Function ex
```

```
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
```

Example an2.c (3/4)

```
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
```

Example an2.c (4/4)

```
Goal Assertion 'l3' (file an2.c, line 7):
Assume {
```

Simple annotation(III)

Listing 23 – an4.c

```
void ex(void) {
  int x=12,y=24,z;
  //@ assert l1: x==12 && y == 24;
  if (x < y) {
    //@ assert lb1:  x<y && x==12 && y == 24;
    z = y;
    //@ assert lb2:  (z == x || z == y) && x<= z && y <=
  }
  else
    {
    //@ assert la1:  x>=y && x==12 && y == 24;
    z = y;
    //@ assert la2:  (z == x || z == y) && x<= z && y <=
    };
   //@ assert l2:  (z == x || z == y) && x<= z && y <=z;
}
```

Example an4.c (1/4)

```
[kernel] Parsing an4.c (with preprocessing)
[wp] Running WP plugin...
[wp] Warning: Missing RTE guards
[wp] 6 goals scheduled
[wp] Proved goals:    8 / 8
  Terminating:      1
  Unreachable:      1
  Qed:              6
```

Example an4.c (2/4)

```
  Function ex
```

```
Goal Assertion 'l1' (file an4.c, line 4):
Assume {
  Type: is_sint32(x) /\ is_sint32(y).
  (* Initializer *)
  Init: x = 12.
  (* Initializer *)
  Init: y = 24.
}
Prove: (x = 12) /\ (y = 24).
Prover Qed returns Valid
```

Example an4.c (3/4)

```
Goal Assertion 'lb1' (file an4.c, line 6):
Assume {
  Type: is_sint32(x) /\ is_sint32(y).
  (* Initializer *)
  Init: x = 12.
  (* Initializer *)
  Init: y = 24.
  (* Assertion 'l1' *)
  Have: (x = 12) /\ (y = 24).
  (* Then *)
  Have: x < y.
}
Prove: (x = 12) /\ (y = 24) /\ (x < y).
Prover Qed returns Valid
```

Example an4.c (4/4)

```
Goal Assertion 'lb2' (file an4.c, line 8):
```

Simple annotation(III)

Listing 24 – an4.c

```
/*@ requires a >= 0 && b>= 0 && a == -6;
  @ assigns \nothing;
  @ ensures \result == 4;
  @*/
int annotation(int a,int b)
{
  int x,y,z;
  x = a;
//*@  x == a; */
  y = b;
//*@  x == a && y == b; */
  z = a+b-2;
//*@  x == a && y == b && z==4; */
  return(z);
  }
```

Example anchina3.c (1/4)

```
[kernel] Parsing anchina3.c (with preprocessing)
[wp] Running WP plugin...
[wp] Warning: Missing RTE guards
[wp] 4 goals scheduled
[wp] Proved goals:    6 / 6
  Terminating:      1
  Unreachable:      1
  Qed:              4
```

Example anchina3.c (2/4)

```
  Function annotation
```

```
Goal Post-condition (file anchina3.c, line 3) in 'annota
Assume {
  Type: is_sint32(a) /\ is_sint32(annotation_0) /\ is_sin
      is_sint32(z).
  (* Pre-condition *)
  Have: (a = (-6)) /\ (0 <= a) /\ (0 <= b).
  (* Pre-condition *)
  Have: (a = (-6)) /\ (0 <= a) /\ (0 <= b).
  (* Block In *)
  Have: (ta_x_0=false) /\ (ta_y_0=false) /\ (ta_z_0=fals
  Have: (a + b) = (2 + z).
  (* Return *)
```

Example anchina3.c (3/4)

```
  (* Return *)
  Have: z = annotation_0.
}
Prove: annotation_0 = 4.
Prover Qed returns Valid
```

```
Goal Assigns nothing in 'annotation' (1/3):
Effect at line 8
Assume {
  Type: is_sint32(a) /\ is_sint32(b).
  Have: (ta_x_1=true) <-> (ta_x_0=true).
  (* Pre-condition *)
  Have: (a = (-6)) /\ (0 <= a) /\ (0 <= b).
  (* Pre-condition *)
  Have: (a = (-6)) /\ (0 <= a) /\ (0 <= b).
```

Example anchina3.c (4/4)

```
  Have: (a = (-6)) /\ (0 <= a) /\ (0 <= b).
  (* Block In *)
  Have: (ta_x_1=false) /\ (ta_y_0=false) /\ (ta_z_0=fals
}
```

Current Summary

1. Introduction of notations
   - First example on proving versus testing
   - Second example on proving versus testing
   - Partial correctness but also runtime error
2. Correctness by contract
   - WP calculus
   - Simple example 1
3. Writing a contract
   - " A la Floyd" Verification Conditions for Contract
   - "A la Hoare" Verification Conditions for Contract
4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c

Summary

4 Examples
- Simple annotation (I)
- Simple annotation (II)
- Simple annotation(III)
- Simple annotation(IV)

5 Mechanizing the contract checking

6 Transforming predicates
- Hoare Logic for PC
- Examples in ACSL
- Définition et propriétés du calcul wp

7 Using predicate transformers for checking contracts (next episod)

8 Memory Models in Frama-c

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

10 Generation of Verification Conditions
- WP calculus in Frama-c
- First annotation
- Second annotation

11 Memory Models in Frama-c

12 Logic Specification

13 Organisation of the verification process

14 Conclusion

Method for verifying partial correctness and RTE

A program P *satisfies* a contract (x,pre,post) :

- P transforms a variable x from an iinitial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\mathsf{P}} x_f$
- x_0 satisfies pre : pre(x_0) and x_f satisfies post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow$ post(x_0, x_f)
- $\mathbb{D}$ is the domain of x for RTE (No Run Time Errors) .

i

Method for verifying partial correctness and RTE

A program P *satisfies* a contract (x,pre,post) :

- P transforms a variable x from an iinitial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\text{P}} x_f$
- x_0 satisfies pre : pre(x_0) and x_f satisfies post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\text{P}} x_f \Rightarrow$ post(x_0, x_f)
- $\mathbb{D}$ is the domain of x for RTE (No Run Time Errors) .

i

variables x : $\mathbb{D}$
requires $pre(x_0)$
ensures $post(x_0, x_f)$
begin
$0 : P_0(x_0, x)$
S
$f : P_f(x_0, x)$
end

Method for verifying partial correctness and RTE

A program P *satisfies* a contract (x,pre,post) :

- P transforms a variable x from an iinitial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\text{P}} x_f$
- x_0 satisfies pre : pre(x_0) and x_f satisfies post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\text{P}} x_f \Rightarrow$ post(x_0, x_f)
- $\mathbb{D}$ is the domain of x for RTE (No Run Time Errors) .

i

variables x : $\mathbb{D}$
requires $pre(x_0)$
ensures $post(x_0, x_f)$
begin
$0 : P_0(x_0, x)$
S
$f : P_f(x_0, x)$
end

- $pre(x_0) \wedge x = x_0 \Rightarrow P_0(x_0, x)$
- $pre(x_0) \wedge P_f(x_0, x) \Rightarrow post(x_0, x)$
- For any pair ℓ, ℓ' such that $\ell \longrightarrow \ell'$, we verify that for any values $x, x' \in \text{MEMORY}$
$$\left(\begin{array}{l} \left(\begin{array}{l} P_\ell(x_0, x)) \\ \wedge cond_{\ell,\ell'}(x) \wedge x' = f_{\ell,\ell'}(x) \end{array} \right) \\ \Rightarrow P_{\ell'}(x_0, x') \end{array} \right),$$
- For any pair m, n such that $m \longrightarrow n$, we verify that $\forall x, x' \in \text{MEMORY}$:
$pre(x_0) \wedge P_m(x_0, x) \Rightarrow \textbf{DOM}(m, n)(x)$

Checking verificatioon conditions

- $pre(x_0) \wedge x = x_0 \Rightarrow P_0(x_0, x)$
- $pre(x_0) \wedge P_f(x_0, x) \Rightarrow post(x_0, x)$
- For any pair ℓ, ℓ' such that $\ell \longrightarrow \ell'$, we verify that for any values $x, x' \in \text{MEMORY}$
$$\left(\begin{array}{l} \left(\begin{array}{l} P_\ell(x_0, x)) \\ \wedge cond_{\ell,\ell'}(x) \wedge x' = f_{\ell,\ell'}(x) \end{array} \right) \\ \Rightarrow P_{\ell'}(x_0, x') \end{array} \right),$$
- For any pair m, n such that $m \longrightarrow n$, we verify that $\forall x, x' \in \text{MEMORY} : pre(x_0) \wedge P_m(x_0, x) \Rightarrow \textbf{DOM}(m, n)(x)$

Example **DOM**(m,n)(x)

$DOM(\ell_0, \ell_1)(u) = u \in minint..maxint \wedge 5 \in minint..maxint \wedge u{+}5 \in minint..maxint$ where

$\ell_0 : P_{\ell_0}(u);$
u := u+5;
$\ell_1 : P_{\ell_0}(u);$

Intuition

- A program P *produces* results or outputs from inputs according to a (operational or denotational) semantics
  - STATES is the set of states of P : $STATES = \mathrm{x} \rightarrow \mathbb{Z}$ where x designate variables of P.
  - s_0 et s_f two states of STATES : $\mathcal{D}(P)(s_0) = s_f$ means that P is executed from the memory state s_0 and produces a final state s_f.
  - For any current state s of P, $s(\mathrm{x}) = x$ for expressing the value of x in state s :

Intuition

- A program P *produces* results or outputs from inputs according to a (operational or denotational) semantics
  - STATES is the set of states of P : $STATES = \mathsf{x} \rightarrow \mathbb{Z}$ where x designate variables of P.
  - s_0 et s_f two states of STATES : $\mathcal{D}(P)(s_0) = s_f$ means that P is executed from the memory state s_0 and produces a final state s_f.
  - For any current state s of P, $s(\mathsf{x}) = x$ for expressing the value of x in state s :

$$s_0(\mathsf{x}) = x_0,\ s_f(\mathsf{x}) = x_f,\ s'(\mathsf{x}) = x'$$

Intuition

- A program P *produces* results or outputs from inputs according to a (operational or denotational) semantics
  - STATES is the set of states of P : $STATES = \mathsf{x} \rightarrow \mathbb{Z}$ where x designate variables of P.
  - s_0 et s_f two states of STATES : $\mathcal{D}(P)(s_0) = s_f$ means that P is executed from the memory state s_0 and produces a final state s_f.
  - For any current state s of P, $s(\mathsf{x}) = x$ for expressing the value of x in state s :

$$s_0(\mathsf{x}) = x_0,\ s_f(\mathsf{x}) = x_f,\ s'(\mathsf{x}) = x'$$

  - $\mathcal{D}(P)(s_0) = s_f$ defines the computation nrelation over the set of states :

$$x_0 \xrightarrow{\mathsf{P}} x_f$$

Intuition

- A program P *produces* results or outputs from inputs according to a (operational or denotational) semantics
  - STATES is the set of states of P : $STATES = \mathsf{x} \rightarrow \mathbb{Z}$ where x designate variables of P.
  - s_0 et s_f two states of STATES : $\mathcal{D}(P)(s_0) = s_f$ means that P is executed from the memory state s_0 and produces a final state s_f.
  - For any current state s of P, $s(\mathsf{x}) = x$ for expressing the value of x in state s :
    $$s_0(\mathsf{x}) = x_0,\ s_f(\mathsf{x}) = x_f,\ s'(\mathsf{x}) = x'$$
  - $\mathcal{D}(P)(s_0) = s_f$ defines the computation nrelation over the set of states :
    $$x_0 \xrightarrow{\mathsf{P}} x_f$$
- A program P *satisfies* the contract (x,pre,post) :
  - P transforms a variable x from a value x_0 and produces a value x_f : $x_0 \xrightarrow{\mathsf{P}} x_f$
  - x_0 satisfies pre : pre(x_0)
  - x_f satisfies post : post(x_0, x_f)
  - pre$(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow$ post(x_0, x_f)

Checking a contract

A program P *satisfies* a contract (x,pre,post) :

- P transforms a variable x from an iinitial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\text{P}} x_f$
- x_0 satisfies pre : pre(x_0) and x_f satisfies post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\text{P}} x_f \Rightarrow$ post(x_0, x_f)

Checking a contract

A program P *satisfies* a contract (x,pre,post) :

- P transforms a variable x from an iinitial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\text{P}} x_f$
- x_0 satisfies pre : pre(x_0) and x_f satisfies post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\text{P}} x_f \Rightarrow$ post(x_0, x_f)

variables x : $\mathbb{D}$
requires $pre(x_0)$
ensures $post(x_0, x_f)$

```
begin
0 : P_0(x_0, x)
S
f : P_f(x_0, x)
end
```

Checking a contract

A program P *satisfies* a contract (x,pre,post) :

- P transforms a variable x from an iinitial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\text{P}} x_f$
- x_0 satisfies pre : pre(x_0) and x_f satisfies post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\text{P}} x_f \Rightarrow$ post(x_0, x_f)

variables x : $\mathbb{D}$
requires $pre(x_0)$
ensures $post(x_0, x_f)$
begin
$0 : P_0(x_0, x)$
S
$f : P_f(x_0, x)$
end

- $pre(x_0) \wedge x = x_0 \Rightarrow P_0(x_0, x)$
- $pre(x_0) \wedge P_f(x_0, x) \Rightarrow post(x_0, x)$
- For any pair ℓ, ℓ' such that $\ell \longrightarrow \ell'$, we verify that for any values $x, x' \in \textsc{Memory}$

$$\left(\begin{array}{l} \left(\begin{array}{l} P_\ell(x_0, x)) \\ \wedge cond_{\ell,\ell'}(x) \wedge x' = f_{\ell,\ell'}(x) \end{array} \right) \\ \Rightarrow P_{\ell'}(x_0, x') \end{array} \right),$$

Checking a contract using the solver Z3

requires $x0 \geq 0$;
ensures $x_f = x0{+}2$;
variables X

begin
$intX = x0$;
$0 : x = x0$
$X = X{+}2$;
$1 : x = x0{+}2$
end

- $x0 \geq 0 \wedge x = x_0 \Rightarrow x = x0$
- $x = x0{+}2 \Rightarrow x = x0{+}2$
- conditions de vérification $0 \longrightarrow 1$:
  $x = x0 \wedge x' = x{+}2 \Rightarrow x' = x0{+}2$
- $(x0 >= 0, x == x0, x! = x0)$
- $(x == x0{+}2, x! = x0{+}2)$
- $(x == x0, xp == x{+}2, xp! = x0{+}2)$

Listing 25 – z3 en Python

```
from numbers import Real
from z3 import *
x = Real('x')
xp = Real('xp')
x0 = Real('x0')
s = Solver()
s.add(x0 >= 0, x == x0, x != x0)
print(s.check())
s.add(x ==  x0+2, x != x0+2)
print(s.check())
s.add(x == x0,   xp == x + 2,  xp != x0+2)
print(s.check())
```

Current Summary

1. Introduction of notations
   - First example on proving versus testing
   - Second example on proving versus testing
   - Partial correctness but also runtime error
2. Correctness by contract
   - WP calculus
   - Simple example 1
3. Writing a contract
   - " A la Floyd" Verification Conditions for Contract
   - "A la Hoare" Verification Conditions for Contract
4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c

Summary

4 Examples
- Simple annotation (I)
- Simple annotation (II)
- Simple annotation(III)
- Simple annotation(IV)

5 Mechanizing the contract checking

6 Transforming predicates
- Hoare Logic for PC
- Examples in ACSL
- Définition et propriétés du calcul wp

7 Using predicate transformers for checking contracts (next episod)

8 Memory Models in Frama-c

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

10 Generation of Verification Conditions
- WP calculus in Frama-c
- First annotation
- Second annotation

11 Memory Models in Frama-c

12 Logic Specification

13 Organisation of the verification process

14 Conclusion

Transformation of predicates

Transformation of predicates

- $\forall x_0, x_f . \mathsf{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$

Transformation of predicates

- $\forall x_0, x_f.\mathsf{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x_0, x.\mathsf{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \mathsf{post}(x_0, x))$

Transformation of predicates

- $\forall x_0, x_f.\mathsf{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x_0, x.\mathsf{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \mathsf{post}(x_0, x))$
- $\forall x_0, x.\mathsf{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \mathsf{post}(x_0, x))$

Transformation of predicates

- $\forall x_0, x_f.\mathsf{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x_0, x.\mathsf{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \mathsf{post}(x_0, x))$
- $\forall x_0, x.\mathsf{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \mathsf{post}(x_0, x))$
- $\forall x_0.\mathsf{pre}(x_0) \Rightarrow \forall x.x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \mathsf{post}(x_0, x)$

Transformation of predicates

- $\forall x_0, x_f.\mathsf{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x_0, x.\mathsf{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \mathsf{post}(x_0, x))$
- $\forall x_0, x.\mathsf{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \mathsf{post}(x_0, x))$
- $\forall x_0.\mathsf{pre}(x_0) \Rightarrow \forall x.x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \mathsf{post}(x_0, x)$
- $\forall x_0.\mathsf{pre}(x_0) \Rightarrow \{P\}\mathsf{post}(x_0, x)$

Transformation of predicates

- $\forall x_0, x_f.\mathsf{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x_0, x.\mathsf{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \mathsf{post}(x_0, x))$
- $\forall x_0, x.\mathsf{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \mathsf{post}(x_0, x))$
- $\forall x_0.\mathsf{pre}(x_0) \Rightarrow \forall x.x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \mathsf{post}(x_0, x)$
- $\forall x_0.\mathsf{pre}(x_0) \Rightarrow \{P\}\mathsf{post}(x_0, x)$

Weakest Liberal Precondition of S for P

$$\{S\}P(x) \stackrel{def}{=} \forall x_f.x \xrightarrow{\mathsf{S}} x_f \Rightarrow \mathsf{P}(x_f)$$

Transformation of predicates

- $\forall x_0, x_f.\mathsf{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x_0, x.\mathsf{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \mathsf{post}(x_0, x))$
- $\forall x_0, x.\mathsf{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \mathsf{post}(x_0, x))$
- $\forall x_0.\mathsf{pre}(x_0) \Rightarrow \forall x.x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \mathsf{post}(x_0, x)$
- $\forall x_0.\mathsf{pre}(x_0) \Rightarrow \{P\}\mathsf{post}(x_0, x)$

Weakest Liberal Precondition of S for P

$$\{S\}P(x) \stackrel{def}{=} \forall x_f.x \xrightarrow{\mathsf{S}} x_f \Rightarrow \mathsf{P}(x_f)$$

- $\{x := e\}P(x) = P[x \mapsto e]$
- $\{\text{if } b(x) \text{ then } S1 \text{ else } S2\ \}P(x) =$
  $b(x) \wedge \{S1\}P(x) \vee \text{ not } b(x)\ \{S2\}P(x)$

Weakest Liberal Precondition of S for P

- $WLP(S)(P(x))$ is another notation for $\{S\}P(x)$.
- $\{$while $b(x)$ do S end$\}P(x) = \{w\}(P(x))$

Weakest Liberal Precondition of S for P

- $WLP(S)(P(x))$ is another notation for $\{S\}P(x)$.
- $\{$while $b(x)$ do S end$\}P(x) = \{w\}(P(x))$
- $\{$if $b(x)$ then $S; w$ else $skip$ $\}P(x) =$

Weakest Liberal Precondition of S for P

- $WLP(S)(P(x))$ is another notation for $\{S\}P(x)$.
- $\{$while $b(x)$ do S end$\}P(x) = \{w\}(P(x))$
- $\{$if $b(x)$ then $S; w$ else $skip$ $\}P(x) =$
- $b(x) \wedge \{S; w\}P(x) \vee$ not $b(x)$ $\wedge$ $\{skip\}P(x) =$

Weakest Liberal Precondition of S for P

- $WLP(S)(P(x))$ is another notation for $\{S\}P(x)$.
- $\{$while $b(x)$ do S end$\}P(x) = \{w\}(P(x))$
- $\{$if $b(x)$ then $S; w$ else $skip$ $\}P(x) =$
- $b(x) \wedge \{S; w\}P(x) \vee$ not $b(x)$ $\wedge$ $\{skip\}P(x) =$
- $b(x) \wedge \{S\}(\{w\}(P(x))) \vee$ not $b(x) \wedge P(x) = \{w\}(P(x))$
- $F(\{w\})(P(x)) = \{w\}(P(x)$

Examples

- $\{$while $x > 0$ do $x := x{-}1$ end$\}(x = 0) = x \geq 0$
- $\{$while $x > 0$ do $x := x{+}1$ end$\}(x = 0) = x \geq 0$
- $\{$while $x > 0$ do $x := x{+}1$ end$\}(x \leq 0) = x \in \mathbb{Z}$

Expressing a contract using predicate transformer

Computing WLP function

- $\forall x_0.\mathsf{pre}(x_0) \Rightarrow \{P\}\mathsf{post}(x_0, x)$
- $\forall x_0.x = x_0 \wedge \mathsf{pre}(x_0) \Rightarrow \{P\}\mathsf{post}(x_0, x)$
- Hoare Triple : $\{pre(x_0) \wedge x = x_0\}P\{post(x_0, x)\}$

Axiomatisation of Hoare Logic

⊠ Definition(Axioms et rules)

- Axiom for assignment : $\{P(e/x)\}$**X :=E(X)**$\{P\}$.
- Axiome for skip : $\{P\}$**skip**$\{P\}$.
- Rule for composition : If $\{P\}\mathbf{S}_1\{R\}$ and $\{R\}\mathbf{S}_2\{Q\}$, then $\{P\}\mathbf{S}_1$ **;**$\mathbf{S}_2\{Q\}$.
- If $\{P \wedge B\}\mathbf{S}_1\{Q\}$ and $\{P \wedge \neg B\}\mathbf{S}_2\{Q\}$, then $\{P\}$**if B then** $\mathbf{S}_1$ **then** $\mathbf{S}_2$ **fi**$\{Q\}$.
- If $\{P \wedge B\}\mathbf{S}\{P\}$, then $\{P\}$**while B do S od**$\{P \wedge \neg B\}$.
- Rule for strengthening/weakening : If $P' \Rightarrow P$, $\{P\}\mathbf{S}\{Q\}$, $Q \Rightarrow Q'$, alors $\{P'\}\mathbf{S}\{Q'\}$.

Axiomatisation : Hoare Logic

Example of proof $\{x = 1\}$**Z :=X ;X :=Y ;Y :=Z**$\{y = 1\}$

- (1) $x = 1 \Rightarrow (z = 1)[x/z]$ (logic property)
- (2) $\{(z = 1)[x/z]\}$**Z :=X**$\{z = 1\}$ (axiom for assignment)
- (3) $\{x = 1\}$**Z :=X**$\{z = 1\}$ (Règle de renforcement/affaiblissement avec (1) et (2))
- (4) $z = 1 \Rightarrow (z = 1)[y/x]$ (logic property)
- (5) $\{(z = 1)[y/x]\}$**X :=Y**$\{z = 1\}$(axiom for assignment)
- (6) $\{z = 1\}$**X :=Y**$\{z = 1\}$ (Règle de renforcement/affaiblissement avec (4) et (5))
- (7) $z = 1 \Rightarrow (y = 1)[z/y]$ (logic property)
- (8) $\{(z = 1)[x/z]\}$**Y :=Z**$\{y = 1\}$ ((axiom for assignment)
- (9) $\{z = 1\}$**Y :=Z**$\{y = 1\}$ (Règle de renforcement/affaiblissement avec (7) et (8))
- (10) $\{x = 1\}$**Z :=X ;X :=Y ;**$\{z = 1\}$ (Rule for composition with 3 et 6)
- (11) $\{x = 1\}$**Z :=X ;X :=Y ;Y :=Z**$\{y = 1\}$ (Rule for composition avec 11 et 9)

Semantics for Hoare

triples

⊠ Definition

$\{P\}\mathbf{S}\{Q\}$ is defined by $\forall s, t \in STATES : P(s) \wedge \mathcal{D}(S)(s) = t \Rightarrow Q(t)$

☺ PropertySoundness of axiomatic system

- If there exists a proof constructed in accordance with the previous rules of $\{P\}\mathbf{S}\{Q\}$, then $\{P\}\mathbf{S}\{Q\}$ is valid.
- If $\{P'\}\mathbf{S}\{Q'\}$ is valid and if the assertion language is sufficiently expressive, then there exists a proof constructed using the preceding rules of $\{P\}\mathbf{S}\{Q\}$.

⊠ Definition

An assertion language is defined by a set of predicates and composition operators such as disjunction and conjunction ; it is equipped with a partial order relation called implication. Let us denote it $(\textsc{Pred}, \Rightarrow, \textbf{false}, \textbf{true}, \wedge, \vee)$: $(\textsc{Pred}, \Rightarrow, \textbf{false}, \textbf{true}, \wedge, \vee)$ is a complete lattice.

An example of an assertion language is that defined by

Introduction of wlp

- $\{P\}\mathbf{S}\{Q\}$
- $\forall s, t \in STATES : P(s) \wedge \mathcal{D}(S)(s) = t \Rightarrow Q(t)$
- $\forall s \in STATES : P(s) \Rightarrow (\forall t \in STATES : \mathcal{D}(S)(s) = t \Rightarrow Q(t))$

Definition of wlp

$wlp(S)(Q) \stackrel{def}{=} (\forall t \in STATES : \mathcal{D}(S)(s) = t \Rightarrow Q(t))$

$wlp(S)(Q) \equiv \overline{(\exists t \in STATES : \mathcal{D}(S)(s) = t \wedge \overline{Q}(t))}$

Relation between wp and wlp

- $loop(S) \equiv \overline{(\exists t \in STATES : \mathcal{D}(S)(s) = t}$ (set of states which does not permit the termination of S de terminer)
- $wp(S)(Q) \equiv wlp(S)(Q) \wedge \overline{loop(S)}$

Definition of wlp

⊠ Definition

$$WLP(S)(P) = \nu\lambda X.((B \wedge wlp(BS)(X)) \vee (\neg B \wedge P))$$

☺ Property

- Si $P \Rightarrow Q$, then $wlp(S)(P) \Rightarrow wlp(S)(Q)$.

Axiomatisation of Hoare Logic

⊠ DefinitionHoaree triple

$\{P\}\mathbf{S}\{Q\} \stackrel{def}{=} P \Rightarrow wlp(S)(Q)$

Axiomatisation of Hoare Logic

⊠ DefinitionHoaree triple

$\{P\}\mathbf{S}\{Q\} \stackrel{def}{=} P \Rightarrow wlp(S)(Q)$

⊠ Definition(Axioms et rules)

- Axiom of assignment : $\{P(e/x)\}$**X :=E(X)**$\{P\}$.
- Axiom of skip : $\{P\}$**skip**$\{P\}$.
- Composition rule : If $\{P\}\mathbf{S}_1\{R\}$ and $\{R\}\mathbf{S}_2\{Q\}$, then $\{P\}\mathbf{S}_1$ **;**$\mathbf{S}_2\{Q\}$.
- If $\{P \wedge B\}\mathbf{S}_1\{Q\}$ and $\{P \wedge \neg B\}\mathbf{S}_2\{Q\}$, then $\{P\}$**if B then** $\mathbf{S}_1$ **then** $\mathbf{S}_2$ **fi**$\{Q\}$.
- If $\{P \wedge B\}\mathbf{S}\{P\}$, then $\{P\}$**while B do S od**$\{P \wedge \neg B\}$.
- Strengthening/weakening rule : If $P' \Rightarrow P$, $\{P\}\mathbf{S}\{Q\}$, $Q \Rightarrow Q'$, then $\{P'\}\mathbf{S}\{Q'\}$.

Verification

- $\{P\}\mathbf{S}\{Q\}$
- $\forall s \in STATES.P(s) \Rightarrow wlp(S)(Q)(s)$
- $\forall s \in STATES.P(s) \Rightarrow (\forall t \in STATES : \mathcal{D}(S)(s) = t \Rightarrow Q(t))$
- $\forall s, t \in STATES.P(s) \wedge \mathcal{D}(S)(s) = t \Rightarrow Q(t))$
- Soundness : If a proof of $\{P\}\mathbf{S}\{Q\}$ has been constructed using the rules of Hoare logic, then $P \Rightarrow wlp(S)(Q)$
- Semantic completeness : If $P \Rightarrow wlp(S)(Q)$, then one can construct a proof of $\{P\}\mathbf{S}\{Q\}$ using the rules of Hoare logic if one can express $wlp(S)(P)$ in the language of assertions.

Example 1

Listing 26 – difference of two numbers

```
#include <limits.h>
/*@ requires a-b >= INT_MIN && a-b <= INT_MAX;
    assigns \nothing;
    ensures \result == (a - b);
*/
static int difference(int a, int b) {
  return a-b;
}
```

- INT_MIN (resp. INT_MAX) is the smallest codable integer (resp. greatest codable integer).
- $a0-b0 \geq INT_MIN \wedge a0-b0 \leq INT_MAX \wedge a = a0 \wedge b = b0 \Rightarrow [\backslash result = a-b]](\backslash result = (a-b))$

Example 2

Listing 27 – incrément de nombre

```
/*@ requires x0 >= 0;
    assigns \nothing;
    ensures \result == x0+2;
  @*/

int exemple(int x0) {
  int x=x0;
  //@ assert x == x0;
  x = x + 2;
//@ assert x== x0+2;
return x;
}
```

Computing WLP

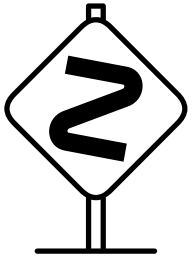
requires $x0 \geq 0$;
ensures $x_f = x0{+}2$;
variables x

```
begin
intx = x0;
0 : x = x0
x := x+2;
1 : x = x0+2
end
```

Conditions de vérification $0 \longrightarrow 1$:

- $x = x0 \wedge x' = x{+}2 \Rightarrow x' = x0{+}2$
- $x = x0 \Rightarrow (x' = x{+}2 \Rightarrow x' = x0{+}2)$
- $x = x0 \Rightarrow (x{+}2 = x0{+}2)$
- $wp(x := x{+}2)(x = x0{+}2) = (x{+}2 = x0{+}2)$
- $x = x0 \wedge x0 \geq 0 \Rightarrow wp(x := x{+}2)(x = x0{+}2)$
- $x = x0 \wedge x0 \geq 0 \Rightarrow x{+}2 = x0{+}2$
- $x = x0 \wedge x0 \geq 0 \Rightarrow x0{+}2 = x0{+}2$

- $x0 \geq 0 \wedge x = x_0 \Rightarrow x = x0$
- $x = x0{+}2 \Rightarrow x = x0{+}2$
- $x = x0 \Rightarrow wp(x := x{+}2)(x = x0{+}2)$



calcul de $wp(X := X{+}2)(x = x0{+}2)$

Example with wp de ACSL/Frama-c before wp

Listing 28 – incrément de nombre

```
/*@ requires x0 >= 0;
    assigns \nothing;
    ensures \result == x0+1;
  @*/

int exemple(int x0) {
  int x=x0;
  //@ assert x == x0;
  x = x + 2;
//@ assert x== x0+2;
return x;
//@ assert \result== x0+2;
}
```

Examplewith wp de ACSL/Frama-c after wp

Listing 29 – incrément de nombre

```
/*@ requires x0 >= 0;
    assigns \nothing;
    ensures \result == x0;
  @*/

int exemple(int x0) {
  int x=x0;
//@ assert x == x0+1;
  x = x + 2;
//@ assert x== x0+2;
return x;

}
```

Opérateur WP

Soit STATES l'ensemble des états sur l'ensemble X des variables. Soit S une instruction de programme sur X. Soit A une partie de STATES.
$s \in WP(S)(A)$, si la condition suivante est vérifiée :

$$\left(\begin{array}{l} \forall t \in STATES : \mathcal{D}(S)(s) = t \Rightarrow t \in A \\ \wedge \\ \exists t \in STATES : \mathcal{D}(S)(s) = t \end{array} \right)$$

- $WP(X := X{+}1)(A) = \{s \in STATES | s[X \mapsto s(X) \oplus 1] \in A\}$
- $WP(X := Y{+}1)(A) = \{s \in STATES | s[X \mapsto s(Y) \oplus 1] \in A\}$
- $WP(while\ X > 0\ do\ X := X{-}1\ od)(A) = \{s \in STATES | (s(X) \leq 0) \vee (s(X) \in A \wedge s(X) < 0)\}$
- $WP(while\ x > 0\ do\ x := x{+}1\ od)(A) = \{s \in STATES | (s(X) \in A \wedge s(X) \leq 0)\}$
- $WP(while\ x > 0\ do\ x := x{+}1\ od)(\varnothing) = \varnothing$
- $WP(while\ x > 0\ do\ x := x{+}1\ od)(STATES) = \{s \in STATES | s(X) \leq 0\}$

Propriétés de WP

Propriétés

- WP est une fonction monotone pour l'inclusion d'ensembles de STATES.
- $WP(S)(\varnothing) = \varnothing$
- $WP(S)(A \cap B) = WP(S)(A) \cap WP(S)(B)$
- $WP(S)(A) \cup WP(S)(B) \subseteq WP(S)(A \cup B)$
- Si S est déterministe, $WP(S)(A \cup B) = WP(S)(A) \cup WP(S)(B)$

- WP est un opérateur avec le profil suivant

  pour toute instruction S du langage de programmation,
  $$WP(S) \in \mathcal{P}(STATES) \nrightarrow \mathcal{P}(STATES)$$
- $(\mathcal{P}(STATES), \subseteq)$ est un treillis complet.
- $(Pred, \Rightarrow)$ est une structure où
  - (1) $Pred$ est une *extension* du langage d'expressions booléennes
  - (2) $Pred$ est une *intension* introduite comme un langage d'assertions
  - $\Rightarrow$ est l'implication
  - $s \in A$ correspond une assertion P vraie en s notée $P(s)$.

Définition structurelle des transformateurs de prédicats

- S est une instruction de STATS.
- T est le type ou les types des variables et D est la constante ou les constantes Définie(s).
- P est un prédicat du langage Pred
- X est une variable de programme
- $E(X, D)$ (resp. $B(X, D)$) est une expression arithmétique (resp. booléenne) dépendant de X et de D.
- x est la valeur de X (X contient la valeur x).
- $e(x, d)$ (resp. $b(x, d)$) est l'expression arithmétique (resp. booléenne) du langage Pred associée à l'expression $E(X, D)$ (resp. $B(X, D)$) du langage des expressions arithmétiques (resp. booléennes) du langage de programmation Prog
- $b(x, d)$ est l'expression arithmétique du langage Pred associée à l'expression $E(X, D)$ du langage des expressions arithmétiques du langage de programmation Prog

Définition structurelle des transformateurs de prédicats

S	wp(S)(P)
X :=E(X,D)	$P[e(x,d)/x]$
SKIP	P
$S_1; S_2$	$wp(S_1)(wp(S_2)(P))$
IF B S_1 ELSE S_2 FI	$(B \Rightarrow wp(S_1)(P)) \wedge (\neg B \Rightarrow wp(S_2)(P))$
WHILE B DO S OD	$\mu.(\lambda X.(B \Rightarrow wp(S)(X)) \wedge (\neg B \Rightarrow P))$

- $wp(X := X{+}5)(x \geq 8) \stackrel{def}{=} x{+}5 \geq 8 \wedge x \geq 3$
- $wp(\text{WHILE } x > 1 \text{ DO } X := X{+}1 \text{ OD})(x = 4) = FALSE$
- $wp(\text{WHILE } x > 1 \text{ DO } X := X{+}1 \text{ OD})(x = 0) = x = 0$

Logique de Hoare Correction Totale

⊠ Definitiontriplets de Hoare Correction Totale

$[P]\mathbf{S}[Q] \stackrel{def}{=} P \Rightarrow wp(S)(Q)$

Logique de Hoare Correction Totale

⊠ Definitiontriplets de Hoare Correction Totale

$[P]\mathbf{S}[Q] \stackrel{def}{=} P \Rightarrow wp(S)(Q)$

⊠ Definition(Axiomes et règles d'inférence)

- Axiome d'affectation : $[P(e/x)]$**X :=E(X)**$[P]$.
- Axiome du saut : $[P]$**skip**$[P]$.
- Règle de composition : Si $[P]\mathbf{S}_1[R]$ et $[R]\mathbf{S}_2[Q]$, alors $[P]\mathbf{S}_1$ **;**$\mathbf{S}_2[Q]$.
- Si $[P \wedge B]\mathbf{S}_1[Q]$ et $[P \wedge \neg B]\mathbf{S}_2[Q]$, alors $[P]$**if B then S**$_1$ **then S**$_2$ **fi**$[Q]$.
- Si $[P(n{+}1)]\mathbf{S}[P(n)]$, $P(n{+}1) \Rightarrow b$, $P(0) \Rightarrow \neg b$, alors $[\exists n \in \mathbb{N}.P(n)]$**while B do S od**$[P(0)]$.
- Règle de renforcement/affaiblissement : Si $P' \Rightarrow P$, $[P]\mathbf{S}[Q]$, $Q \Rightarrow Q'$, alors $[P']\mathbf{S}[Q']$.

Correction

:
Si $[P]$**S**$[Q]$ est dérivé selon les règles ci-dessus, alors $P\wp(S)5Q)$.

- $[P(e/x)]$**X :=E(X)**$[P]$ est valide : $wp(X := E)(P)/x = P(e/x)$.
- $[\exists n \in \mathbb{N}.P(n)]$**while B do S od**$[P(0)]$: si s est un état de P(n) alors au bout de n boucles on atteint un état s_f tel que P(0) est vrai en s_f.

Complétude

:
Si $P \Rightarrow wp(S)(Q)$, alors il existe une preuve de $[P]\mathbf{S}[Q]$ construites avec les règles ci-dessus,

- $P \Rightarrow wp(X := E(X))(Q)$: $P \Rightarrow Q(e/x)$ et $[Q(e/x)]$**X :=E(X)**$[Q]$ constituent une preuve.
- $P \Rightarrow wp(while)(Q)$:
  - On construit la suite de $P(n)$ en définissant $P(n) = W_n$.
  - On vérifie que cela vérifie la règle du while.

Current Summary

1. Introduction of notations
   - First example on proving versus testing
   - Second example on proving versus testing
   - Partial correctness but also runtime error
2. Correctness by contract
   - WP calculus
   - Simple example 1
3. Writing a contract
   - " A la Floyd" Verification Conditions for Contract
   - "A la Hoare" Verification Conditions for Contract
4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c

Summary

4 Examples
- Simple annotation (I)
- Simple annotation (II)
- Simple annotation(III)
- Simple annotation(IV)

5 Mechanizing the contract checking

6 Transforming predicates
- Hoare Logic for PC
- Examples in ACSL
- Définition et propriétés du calcul wp

7 Using predicate transformers for checking contracts (next episod)

8 Memory Models in Frama-c

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

10 Generation of Verification Conditions
- WP calculus in Frama-c
- First annotation
- Second annotation

11 Memory Models in Frama-c

12 Logic Specification

13 Organisation of the verification process

14 Conclusion

Verification of contract (I)

A program P *satisfies* a contract (pre,post) :

- P transforms a variable x from an initial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\mathsf{P}} x_f$
- x_0 satisfies pre : pre(x_0) and x_f satisfait post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow$ post(x_0, x_f)

---

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X

```
begin
0 : P_0(x_0, x)
instruction_0
...
i : P_i(x_0, x)
...
instruction_{f-1}
f : P_f(x_0, x)
end
```

Floyd

- $pre(x_0) \wedge x = x_0 \Rightarrow P_0(x_0, x)$
- $pre(x_0) \wedge P_f(x_0, x) \Rightarrow post(x_0, x)$
- For each pair ℓ, ℓ' such that $\ell \longrightarrow \ell'$, one checks that foa any value $x, x' \in \textsc{Memory}$

$$\left(\begin{array}{l} \left(\begin{array}{l} pre(x_0) \wedge P_\ell(x_0, x)) \\ \wedge cond_{\ell,\ell'}(x) \wedge x' = f_{\ell,\ell'}(x) \end{array} \right) \\ \Rightarrow P_{\ell'}(x_0, x') \end{array} \right),$$

Verification du contract (II)

A program P *satisfies* a contract (pre,post) :

- P transforms a variable x from an initial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\text{P}} x_f$
- x_0 satisfies pre : pre(x_0) and x_f satisfait post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\text{P}} x_f \Rightarrow$ post(x_0, x_f)

---

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X

```
begin
0 : P_0(x_0, x)
instruction_0
...
i : P_i(x_0, x)
...
instruction_{f-1}
f : P_f(x_0, x)
end
```

Verification du contract (II)

A program P *satisfies* a contract (pre,post) :

- P transforms a variable x from an initial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\mathsf{P}} x_f$
- x_0 satisfies pre : pre(x_0) and x_f satisfait post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow$ post(x_0, x_f)

---

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X

```
begin
0 : P_0(x_0, x)
instruction_0
...
i : P_i(x_0, x)
...
instruction_{f-1}
f : P_f(x_0, x)
end
```

- $\forall x_f, x_0.\text{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f)$

Verification du contract (II)

A program P *satisfies* a contract (pre,post) :

- P transforms a variable x from an initial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\text{P}} x_f$
- x_0 satisfies pre : pre(x_0) and x_f satisfait post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\text{P}} x_f \Rightarrow$ post(x_0, x_f)

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X

```
begin
0 : P_0(x_0, x)
instruction_0
...
i : P_i(x_0, x)
...
instruction_{f-1}
f : P_f(x_0, x)
end
```

- $\forall x_f, x_0.\text{pre}(x_0) \wedge x_0 \xrightarrow{\text{P}} x_f \Rightarrow \text{post}(x_0, x_f)$
- $\forall x_f, x_0.\text{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\text{P}} x_f \Rightarrow \text{post}(x_0, x_f))$

Verification du contract (II)

A program P *satisfies* a contract (pre,post) :

- P transforms a variable x from an initial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\text{P}} x_f$
- x_0 satisfies pre : pre(x_0) and x_f satisfait post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\text{P}} x_f \Rightarrow$ post(x_0, x_f)

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X

```
begin
0 : P_0(x_0, x)
instruction_0
...
i : P_i(x_0, x)
...
instruction_{f-1}
f : P_f(x_0, x)
end
```

- $\forall x_f, x_0.\text{pre}(x_0) \wedge x_0 \xrightarrow{\text{P}} x_f \Rightarrow \text{post}(x_0, x_f)$
- $\forall x_f, x_0.\text{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\text{P}} x_f \Rightarrow \text{post}(x_0, x_f))$
- $\forall x_0.\text{pre}(x_0) \Rightarrow \forall x_f.(x_0 \xrightarrow{\text{P}} x_f \Rightarrow \text{post}(x_0, x_f))$

Verification du contract (II)

A program P *satisfies* a contract (pre,post) :

- P transforms a variable x from an initial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\mathsf{P}} x_f$
- x_0 satisfies pre : pre(x_0) and x_f satisfait post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow$ post(x_0, x_f)

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X

begin
$0 : P_0(x_0, x)$
$\texttt{instruction}_0$
...
$i : P_i(x_0, x)$
...
$\texttt{instruction}_{f-1}$
$f : P_f(x_0, x)$
end

- $\forall x_f, x_0.\text{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f)$
- $\forall x_f, x_0.\text{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f))$
- $\forall x_0.\text{pre}(x_0) \Rightarrow \forall x_f.(x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f))$
- $\forall x_0.\text{pre}(x_0) \Rightarrow \forall x.(x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \text{post}(x_0, x))$

Verification du contract (II)

A program P *satisfies* a contract (pre,post) :

- P transforms a variable x from an initial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\mathsf{P}} x_f$
- x_0 satisfies pre : pre(x_0) and x_f satisfait post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow$ post(x_0, x_f)

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X

begin
$0 : P_0(x_0, x)$
$\texttt{instruction}_0$
...
$i : P_i(x_0, x)$
...
$\texttt{instruction}_{f-1}$
$f : P_f(x_0, x)$
end

- $\forall x_f, x_0.\text{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f)$
- $\forall x_f, x_0.\text{pre}(x_0) \Rightarrow (x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f))$
- $\forall x_0.\text{pre}(x_0) \Rightarrow \forall x_f.(x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \text{post}(x_0, x_f))$
- $\forall x_0.\text{pre}(x_0) \Rightarrow \forall x.(x_0 \xrightarrow{\mathsf{P}} x \Rightarrow \text{post}(x_0, x))$
- $\forall x_0.\text{pre}(x_0) \Rightarrow WLP(P)(\text{post}(x_0, x))$

Partial Correctness by computing WLPs

A program P *satisfies* a contract (pre,post) :

- P transforms a variable x from an initial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\mathsf{P}} x_f$
- x_0 satisfies pre : pre(x_0) and x_f satisfies post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow$ post(x_0, x_f)
- $\forall x_0.$pre$(x_0) \Rightarrow WLP(P)($post$(x_0, x))$

Partial Correctness by computing WLPs

A program P *satisfies* a contract (pre,post) :

- P transforms a variable x from an initial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\mathsf{P}} x_f$
- x_0 satisfies pre : pre(x_0) and x_f satisfies post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow$ post(x_0, x_f)
- $\forall x_0.$pre$(x_0) \Rightarrow WLP(P)($post$(x_0, x))$
- WLP is not computable ...
- Using Hoare logic in the WLP computing as suggested by Rustan Leino. de WLP.

Verification of contract (III)

A program P *satisfies* a contract (pre,post) :

- P transforms a variable x from an initial value x_0 and produces a final value x_f : $x_0 \xrightarrow{\mathsf{P}} x_f$
- x_0 satisfies pre : pre(x_0) and x_f satisfait post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow$ post(x_0, x_f)

---

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X

```
begin
/·@assert P0(x0,x)·/
T;
/·@loop invariant I(x0,x)·/
while B(x) do
   S
od
/·@assert Pf(x0,x)·/
end
```

1. $x = x_0 \wedge \mathsf{pre}(x_0) \Rightarrow P_0(x_0, x)$
2. $\mathsf{pre}(x_0) \wedge P_0(x_0, x) \Rightarrow WLP(T)(I(x_0, x))$
3. $I(x_0, x) \wedge B(x) \Rightarrow WLP(S)(I(x_0, x))$
4. $I(x_0, x) \wedge \neg B(x) \Rightarrow P_f(x_0, x)$

Verification of contract (IV)

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X

```
begin
/·@assert P_0(x_0, x)·/
S1;
S2;
/·@assert P_f(x_0, x)·/
end
```

(1)
$x = x_0 \wedge \mathsf{pre}(x_0) \Rightarrow P_0(x_0, x)$
(2) $P_0(x_0, x) \Rightarrow$
$WLP(S1; S2)(P_f(x_0, x))$

Verification of contract (V)

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X

```
begin
/·@assert P0(x0, x)·/
if B(x) do
   S1
else
   S2
elfi
/·@assert Pf(x0, x)·/
end
```

$$x = x_0 \wedge \mathsf{pre}(x_0) \Rightarrow P_0(x_0, x)$$

$$\begin{array}{l} P_0(x_0, x) \Rightarrow \\ \quad B(x) \wedge WLP(S1)(P_f(x_0, x)) \\ \quad \vee \\ \quad \neg B(x) \wedge WLP(S2)(P_f(x_0, x)) \end{array}$$

Example anchina4.c

Listing 30 – anchina4.c

```
/*@ requires a >= 0 ;
  @ assigns \nothing;
  @ ensures \result == 0;
  @*/
int loop(int a)
{
  int x=a, r;
  /*@
    loop invariant (0 <= x <= a);
     loop assigns x;
    loop variant x;
   */
   while (x >0) { --x;};
   r= x;
  return r;
}
```

Example v1 (1/4)

```
[kernel] Parsing anchina4.c (with preprocessing)
[wp] Running WP plugin...
[wp] Warning: Missing RTE guards
[wp] 8 goals scheduled
[wp] Proved goals:    10 / 10
  Terminating:        1
  Unreachable:        1
  Qed:                6
  Alt-Ergo 2.6.0:     2 (12ms-12ms)
```

Example v1 (2/4)

```
Function loop

Goal Post-condition (file anchina4.c, line 3) in 'loop':
Assume {
  Type: is_sint32(a) /\ is_sint32(loop_0) /\ is_sint32(r
      is_sint32(x_1) /\ is_sint32(x_2).
  (* Pre-condition *)
  Have: 0 <= a.
  (* Pre-condition *)
  Have: 0 <= a.
  (* Block In *)
  Have: (ta_r_0=false) /\ (ta_x_0=true) /\ (ta_x_1=false
  (* Initializer *)
```

Example v1 (3/4)

```
(* Initializer *)
Init: (Init_x_0=true).
(* Initializer *)
Init: x_2 = a.
Have: (ta_x_2=true) <-> (ta_x_0=true).
(* Invariant *)
Have: (x_2 <= a) /\ (0 <= x_2).
(* Loop assigns ... *)
Have: ((Init_x_0=true) -> (Init_x_1=true)).
(* Invariant *)
Have: (x_1 <= a) /\ (0 <= x_1).
(* Else *)
Have: x_1 <= 0.
Have: x_1 = x_3.
Have: x_3 = r.
(* Return *)
Have: r = loop_0.
```

Example v1 (4/4)

```
  Have: r = loop_0.
  (* Block Out *)
  Have: (ta_x_2=true).
}
```

Computing WLP

- Frama-c provides a plugin -wp for generating these proof obligations.
- The WLP formulas are not always computable as for instance the case of the xhile loop.
- The method of Rustan Leino is simple and clever.

Current Summary

1. Introduction of notations
   - First example on proving versus testing
   - Second example on proving versus testing
   - Partial correctness but also runtime error
2. Correctness by contract
   - WP calculus
   - Simple example 1
3. Writing a contract
   - " A la Floyd" Verification Conditions for Contract
   - "A la Hoare" Verification Conditions for Contract
4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c

Summary

4 Examples
- Simple annotation (I)
- Simple annotation (II)
- Simple annotation(III)
- Simple annotation(IV)

5 Mechanizing the contract checking

6 Transforming predicates
- Hoare Logic for PC
- Examples in ACSL
- Définition et propriétés du calcul wp

7 Using predicate transformers for checking contracts (next episod)

8 Memory Models in Frama-c

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

10 Generation of Verification Conditions
- WP calculus in Frama-c
- First annotation
- Second annotation

11 Memory Models in Frama-c

12 Logic Specification

13 Organisation of the verification process

14 Conclusion

Two Memory Models

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

Hoare 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.

Typed Model

- 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.

Example swap.c

Listing 31 – anchina4.c

```
#include <limits.h>

/*@
    requires \valid(a) && \valid(b) ;
    assigns *a;
    assigns *b;
    ensures A: *a == \old(*b);
     ensures B: *b == \old(*a);
  @*/
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Example swap.c (1/4)

```
[kernel] Parsing swap.c (with preprocessing)
[wp] Running WP plugin...
[wp] Warning: Missing RTE guards
[wp] 4 goals scheduled
[wp] Proved goals:    6 / 6
  Terminating:        1
  Unreachable:        1
  Qed:                3
  Alt-Ergo 2.6.0:     1 (13ms)
```

Example swap.c (1/4)

```
[kernel] Parsing swap.c (with preprocessing)
[wp] Running WP plugin...
[wp] Warning: Missing RTE guards
[wp] 4 goals scheduled
[wp] Proved goals:    6 / 6
  Terminating:       1
  Unreachable:       1
  Qed:               3
  Alt-Ergo 2.6.0:    1 (13ms)
```

Example swap.c (1/4)

```
Function swap

Goal Post-condition 'A' in 'swap':
Let x = Mint_2[a_1].
Let x_1 = Mint_1[b].
Let x_2 = Mint_0[a].
Assume {
  Type: is_sint32(Mint_1[a]) /\ is_sint32(x_1) /\ is_sint32(x_2) /\
      is_sint32(Mint_0[b]) /\ is_sint32(x).
  (* Heap *)
  Type: (region(a.base) <= 0) /\ (region(b.base) <= 0) /\ linked(Malloc_0) /\
      cinits(Init_0).
  Have: (Malloc_1 = Malloc_0) /\ (Mint_2 = Mint_1) /\ (a_1 = a) /\ (b_1 = b).
  (* Pre-condition *)
  Have: valid_rw(Malloc_1, a_1, 1) /\ valid_rw(Malloc_1, b_1, 1).
  (* Pre-condition *)
  Have: valid_rw(Malloc_1, a_1, 1) /\ valid_rw(Malloc_1, b_1, 1).
  (* Initializer *)
  Init: x = tmp_0.
  Have: (Init_0[a_1 <- true] = Init_1) /\
      (Mint_2[a_1 <- Mint_2[b_1]] = Mint_3).
  Have: (Init_1[b_1 <- true] = Init_2) /\ (Mint_3[b_1 <- tmp_0] = Mint_0).
}
Prove: x_2 = x_1.
Prover Alt-Ergo 2.6.0 returns Valid (13ms) (32)
```

Example swap.c (1/4)

```
Goal Post-condition 'B' in 'swap':
Let x = Mint_2[a_1].
Let x_1 = Mint_1[a].
Let x_2 = Mint_0[b].
Assume {
  Type: is_sint32(x_1) /\ is_sint32(Mint_1[b]) /\ is_sint32(Mint_0[a]) /\
      is_sint32(x_2) /\ is_sint32(x).
  (* Heap *)
  Type: (region(a.base) <= 0) /\ (region(b.base) <= 0) /\ linked(Malloc_0) /\
      cinits(Init_0).
  Have: (Malloc_1 = Malloc_0) /\ (Mint_2 = Mint_1) /\ (a_1 = a) /\ (b_1 = b).
  (* Pre-condition *)
  Have: valid_rw(Malloc_1, a_1, 1) /\ valid_rw(Malloc_1, b_1, 1).
  (* Pre-condition *)
  Have: valid_rw(Malloc_1, a_1, 1) /\ valid_rw(Malloc_1, b_1, 1).
  (* Initializer *)
  Init: x = tmp_0.
  Have: (Init_0[a_1 <- true] = Init_1) /\
      (Mint_2[a_1 <- Mint_2[b_1]] = Mint_3).
  Have: (Init_1[b_1 <- true] = Init_2) /\ (Mint_3[b_1 <- tmp_0] = Mint_0).
}
Prove: x_2 = x_1.
Prover Qed returns Valid
```

Example swap.c (1/4)

```
Goal Assigns (file swap.c, line 5) in 'swap' (1/2):
Effect at line 12
Let x = Mint_2[a].
Assume {
  Type: is_sint32(Mint_0[a_1]) /\ is_sint32(Mint_0[b]) /\
      is_sint32(Mint_1[a_1]) /\ is_sint32(Mint_1[b]) /\ is_sint32(x).
  (* Heap *)
  Type: (region(a_1.base) <= 0) /\ (region(b.base) <= 0) /\
      linked(Malloc_0) /\ cinits(Init_0).
  (* Goal *)
  When: !invalid(Malloc_0, a, 1).
  Have: (Malloc_1 = Malloc_0) /\ (Mint_2 = Mint_0) /\ (a = a_1) /\ (b_1 = b).
  (* Pre-condition *)
  Have: valid_rw(Malloc_1, a, 1) /\ valid_rw(Malloc_1, b_1, 1).
  (* Pre-condition *)
  Have: valid_rw(Malloc_1, a, 1) /\ valid_rw(Malloc_1, b_1, 1).
  (* Initializer *)
  Init: x = tmp_0.
}
Prove: (a = a_1) \/ (b = a).
Prover Qed returns Valid
```

Example swap.c (1/4)

```
Goal Assigns (file swap.c, line 5) in 'swap' (2/2):
Effect at line 13
Assume {
  Type: is_sint32(Mint_0[a]) /\ is_sint32(Mint_0[b_1]) /\
      is_sint32(Mint_1[a]) /\ is_sint32(Mint_1[b_1]) /\
      is_sint32(Mint_2[a_1]).
  (* Heap *)
  Type: (region(a.base) <= 0) /\ (region(b_1.base) <= 0) /\
      linked(Malloc_0) /\ cinits(Init_0).
  (* Goal *)
  When: !invalid(Malloc_0, b, 1).
  Have: (Malloc_1 = Malloc_0) /\ (a_1 = a) /\ (b = b_1).
  (* Pre-condition *)
  Have: valid_rw(Malloc_1, a_1, 1) /\ valid_rw(Malloc_1, b, 1).
  (* Pre-condition *)
  Have: valid_rw(Malloc_1, a_1, 1) /\ valid_rw(Malloc_1, b, 1).
}
Prove: (b = a) \/ (b = b_1).
Prover Qed returns Valid
```

Summary on annotations and assertions

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

Current Summary

1. Introduction of notations
   - First example on proving versus testing
   - Second example on proving versus testing
   - Partial correctness but also runtime error
2. Correctness by contract
   - WP calculus
   - Simple example 1
3. Writing a contract
   - " A la Floyd" Verification Conditions for Contract
   - "A la Hoare" Verification Conditions for Contract
4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c

Summary

4 Examples
- Simple annotation (I)
- Simple annotation (II)
- Simple annotation(III)
- Simple annotation(IV)

5 Mechanizing the contract checking

6 Transforming predicates
- Hoare Logic for PC
- Examples in ACSL
- Définition et propriétés du calcul wp

7 Using predicate transformers for checking contracts (next episod)

8 Memory Models in Frama-c

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

10 Generation of Verification Conditions
- WP calculus in Frama-c
- First annotation
- Second annotation

11 Memory Models in Frama-c

12 Logic Specification

13 Organisation of the verification process

14 Conclusion

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 32 – contrat

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

Examples of contract (1)

(Division)

Listing 33 – 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 f(int x)
{
int y;
y = x/2 ;
return(y);
}
```

Examples of contract (1)

(Division)

Listing 34 – 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 f(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 is the value of the variable x when calling the function $x \geq 0 \wedge x < 10; \Rightarrow \left(\begin{array}{lcl} x \,\%\, 2 & = & 0 \Rightarrow 2 \cdot (x/2) = x \\ x \,\%\, 2 & \neq & 0 \Rightarrow 2 \cdot (x/2) = x - 1 \end{array} \right)$

Proof Obligations from Frama-c

```
  Goal Post-condition (file annotation.c, line 3) in 'annotation':
Let x_1 = x / 2.
Assume {
  Type: is_sint32(x) /\ is_sint32(x_1).
  (* Goal *)
  When: (x % 2) = 0.
  (* Pre-condition *)
  Have: (0 <= x) /\ (x <= 10).
}
Prove: (2 * x_1) = x.
Prover Alt-Ergo 2.6.0 returns Valid (Qed:0.71ms) (13ms) (29)


Goal Post-condition (file annotation.c, line 4) in 'annotation':
Let x_1 = x / 2.
Assume {
  Type: is_sint32(x) /\ is_sint32(x_1).
  (* Goal *)
  When: (x % 2) != 0.
  (* Pre-condition *)
  Have: (0 <= x) /\ (x <= 10).
}
Prove: (1 + (2 * x_1)) = x.
Prover Alt-Ergo 2.6.0 returns Valid (Qed:0.55ms) (14ms) (37)
}
```

Verification of a contract

A programme P *fulfils* a contract (pre,post) :

- P transforms a variable v à partir d'une valeur initiale v_0 and producing a final value v_f : $x_0 \xrightarrow{P} v_f$
- x_0 satisfait pre : pre(v_0) and v_f satisfait post : post(v_0, v_f)
- pre$(v_0) \wedge v_0 \xrightarrow{P} v_f \Rightarrow$ post(v_0, v_f)

requires $pre(in_0)$
ensures $post(in_0, out_f)$
variables in, out
$type1 \quad f(type2\ in_0)$
local variables x
  begin
    $S(f)$
  $return(out)$;
  end

- $v = (in, out, x)$: in (input parameters), out (results), x (local variables)
- pre$(in_0) \wedge (in_0, out_0, v_0) \xrightarrow{P} (in_0, out_f, v_f) \Rightarrow$ post(in_0, out_f)
- pre$(in_0) \wedge in = in_0 \wedge out = out_0 \Rightarrow [S(f)](post(in_0, out))$

Verification Condition for Assignment

Listing 35 – an1.c

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

Verification Condition for Assignment

Listing 36 – an1.c

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

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

Verification Condition for Assignment

Listing 37 – an1.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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 38 – an1.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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 39 – an1.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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 40 – an1.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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 41 – an1.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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 42 – an1.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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 43 – an1.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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:$Q(x)$
-

Verification conditions

Verification conditions

```
  Goal Assertion
  (file annotations14.c, line 5):
Assume {
  Type: is_sint32(x) /\ is_sint32(x_1).
  Have: x_1 = 1.
  Have: (1 + x_1) = x.
}
Prove: x = 2.
Prover Qed returns Valid
```

Verification conditions

```
  Goal Assertion
  (file annotations14.c, line 5):
Assume {
  Type: is_sint32(x) /\ is_sint32(x_1).
  Have: x_1 = 1.
  Have: (1 + x_1) = x.
}
Prove: x = 2.
Prover Qed returns Valid
```

```
void main(void){
  int x;
  x = 1;
  x = x + 1;
  /*@  assert x == 2; */
}
```

Verification conditions

```
  Goal Assertion
  (file annotations14.c, line 5):
Assume {
  Type: is_sint32(x) /\ is_sint32(x_1).
  Have: x_1 = 1.
  Have: (1 + x_1) = x.
}
Prove: x = 2.
Prover Qed returns Valid
```

```
void main(void){
  int x;
  x = 1;
  x = x + 1;
  /*@  assert x == 2; */

  }
```

```
        Goal Assertion
        (file annotations15.c, line 7):
  Assume {
    Type: is_sint32(x) /\ is_sint32(x_1)
         /\ is_sint32(x_2) /\ is_sint32(y).
    Have: x_2 = 1.
    Have: (1 + x_2) = x_1.
    Have: (2 * x_1) = y.
    Have: (x_1 + y) = x.
  }
  Prove: x = 6.
  Prover Qed returns Valid
```

Verification conditions

```
  Goal Assertion
  (file annotations14.c, line 5):
Assume {
  Type: is_sint32(x) /\ is_sint32(x_1).
  Have: x_1 = 1.
  Have: (1 + x_1) = x.
}
Prove: x = 2.
Prover Qed returns Valid
```

```
void main(void){
  int x;
  x = 1;
  x = x + 1;
  /*@  assert x == 2; */

  }
```

```
        Goal Assertion
        (file annotations15.c, line 7):
  Assume {
    Type: is_sint32(x) /\ is_sint32(x_1)
          /\ is_sint32(x_2) /\ is_sint32(y).
    Have: x_2 = 1.
    Have: (1 + x_2) = x_1.
    Have: (2 * x_1) = y.
    Have: (x_1 + y) = x.
  }
  Prove: x = 6.
  Prover Qed returns Valid
```

```
void main(void){
  int x,y;
  x = 1;
  x = x + 1;
  y = 2*x;
  x = x+y;
  /*@  assert x == 6; */

  }
```

Examples of contract (2)

(Precondition)

Listing 44 – 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 45 – 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 fucntion to compute (what to compute ?)
- Define an inductive method for computing the mathematical function and using axioms.

(facctorial what)

Listing 46 – 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

(facctorial how)

Listing 47 – 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;

}
```

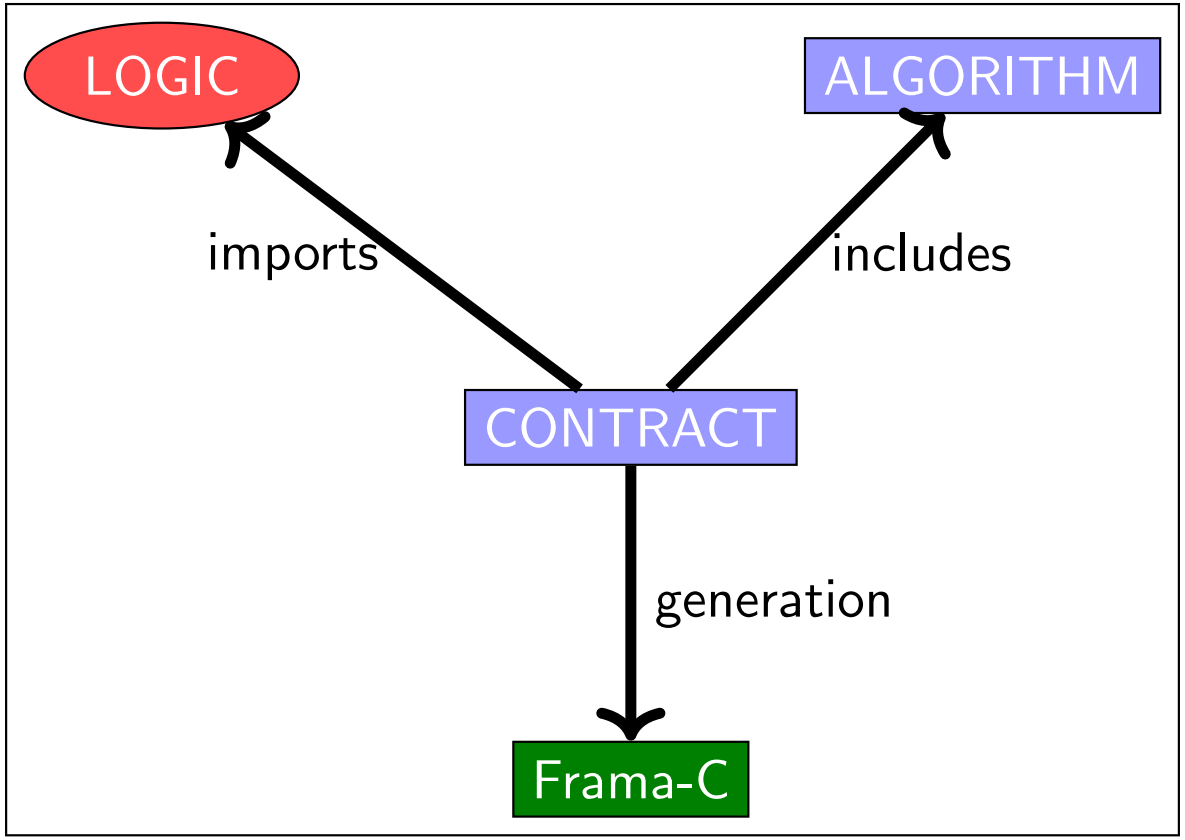
Definition of a contract (approach)

- The specification of a function (mathfact) to compute requires to define it mathematically.
- The definition is stated in an 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 48 – contrat

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

Division should not return silly expressions !

(Pairs of integers)

Listing 49 – project-divers/structures.h

```
#ifndef _STRUCTURE_H


struct s {
    int q;
    int r;
};

#endif
```

Division should not return silly expressions !

(Specification)

Listing 50 – project-divers/division.h

```
#ifndef _A_H
#define _A_H
#include "structures.h"
/*@ requires a >= 0 && b >= 0;
  @ requires b != 0;
  @ ensures 0 <= \result.r;
  @ ensures \result.r < b;
  @ ensures a == b * \result.q + \result.r;
*/
struct s division(int a, int b);
#endif
```

Division should not return silly expressions !

(Algorithm)

Listing 51 – 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 $\{P \wedge B\}\mathbf{S}\{P\}$, 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 52 – loop.c

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

(Invariant de boucle)

Listing 53 – 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;
}
```

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 54 – 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 ;
}
```

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 $\backslash at(e, Old)$

Exemple pour $\backslash at(e, id)$

(label Pre)

Listing 55 – 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 $\backslash at(e, id)$

(autre label)

Listing 56 – 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; */
} }
}
```

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

(otherlabel)

Listing 57 – 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;
}
```

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 58 – 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 59 – 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;
}
```

Labels in ACSL

In **Frama-C** and the **ACSL** language, the expression :

```
\at(e, L)
```

evaluates an expression `e` in the memory state corresponding to a label `L`.

Main labels

- `Pre` : state at the beginning of the function
- `Post` : state at the end of the function
- `Here` : current state
- `Old` : initial value (often in `ensures`)
- `LoopEntry` : state before the loop
- `LoopCurrent` : start of the current iteration
- `Init` : initial state of the programme

Example with Old

```
void swap(int *a, int *b){

  int tmp;

  /*@
    requires a != \null && b != \null;
    assigns *a, *b;
    ensures *a == \at(*b,Old);
    ensures *b == \at(*a,Old);
  */

  tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Why use \at

In proofs with the **WP** plugin, we must compare :

- the state **before** execution
- the state **after** execution

Example :

$$\cdot a_{final} = b_{initial}$$

We therefore write :

```
*a == \at(*b,Pre)
```

This allows the prover to reason about the **initial values of the programme**.

Example

Listing 60 – contrat

```
/*@ requires x >= 0;
@ ensures \result == \old(x) ;
@*/
int f(int x)
{
  int y;
  y=x;
  x=x+1;
  return(y);
 }
```

Example

Listing 61 – contrat

```
/*@ requires x >= 0;
@ ensures \result == \old(x) && \old(x) == x;
@*/
int f(int x)
{int y;
  y=x;
  x=x+1;
  /*@ assert x == \at(x,Pre)+1; */
  return(y);
}
```

Example

Listing 62 – contrat

```
/*@ requires x >= 0;
@ assigns \nothing;
@ ensures \result == \old(x);
@*/
int f(int x)
{int y;
  y=x;
  x=x+1;
  return(y);
}
```

Current Summary

1. Introduction of notations
   - First example on proving versus testing
   - Second example on proving versus testing
   - Partial correctness but also runtime error
2. Correctness by contract
   - WP calculus
   - Simple example 1
3. Writing a contract
   - " A la Floyd" Verification Conditions for Contract
   - "A la Hoare" Verification Conditions for Contract
4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c

Summary

4 Examples
- Simple annotation (I)
- Simple annotation (II)
- Simple annotation(III)
- Simple annotation(IV)

5 Mechanizing the contract checking

6 Transforming predicates
- Hoare Logic for PC
- Examples in ACSL
- Définition et propriétés du calcul wp

7 Using predicate transformers for checking contracts (next episod)

8 Memory Models in Frama-c

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

10 Generation of Verification Conditions
- WP calculus in Frama-c
- First annotation
- Second annotation

11 Memory Models in Frama-c

12 Logic Specification

13 Organisation of the verification process

14 Conclusion

Verification Condition for Assignment

Listing 63 – an1.c

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

Verification Condition for Assignment

Listing 64 – an1.c

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

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

Verification Condition for Assignment

Listing 65 – an1.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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 66 – an1.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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 67 – an1.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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 68 – an1.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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 69 – an1.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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 70 – an1.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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 71 – an1.c

```
//@ assert l1: P(x);
  x = e(x);
//@ 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:$Q(x)$
-

Annotation simple (I)

Listing 72 – an1.c

```
int main(void){
  signed long int x,y,z; //     int x,y,z;
  x = 1;
  /*@  assert x == 1; */
  y = 2;
  /*@ assert x == 1 && y == 2; */
  z = x *y;
  /*@  assert x == 1 && y == 1 && z == 2; */
  return 0;
  }
```

Annotation simple(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
```

Annotation simple (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
```

Annotation simple (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
```

Annotation simple(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
```

Annotation simple (II)

Listing 73 – 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;
}
```

Annotation simple (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
```

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 74 – 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;
}
```

Current Summary

1. Introduction of notations
   - First example on proving versus testing
   - Second example on proving versus testing
   - Partial correctness but also runtime error
2. Correctness by contract
   - WP calculus
   - Simple example 1
3. Writing a contract
   - " A la Floyd" Verification Conditions for Contract
   - "A la Hoare" Verification Conditions for Contract
4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c

Summary

4 Examples
- Simple annotation (I)
- Simple annotation (II)
- Simple annotation(III)
- Simple annotation(IV)

5 Mechanizing the contract checking

6 Transforming predicates
- Hoare Logic for PC
- Examples in ACSL
- Définition et propriétés du calcul wp

7 Using predicate transformers for checking contracts (next episod)

8 Memory Models in Frama-c

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

10 Generation of Verification Conditions
- WP calculus in Frama-c
- First annotation
- Second annotation

11 Memory Models in Frama-c

12 Logic Specification

13 Organisation of the verification process

14 Conclusion

Two Memory Models

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

Hoare 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.

Typed Model

- 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.

Example swap.c

Listing 75 – anchina4.c

```
#include <limits.h>

/*@
    requires \valid(a) && \valid(b) ;
    assigns *a;
    assigns *b;
    ensures A: *a == \old(*b);
     ensures B: *b == \old(*a);
  @*/
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Example swap.c (1/4)

```
[kernel] Parsing swap.c (with preprocessing)
[wp] Running WP plugin...
[wp] Warning: Missing RTE guards
[wp] 4 goals scheduled
[wp] Proved goals:    6 / 6
  Terminating:        1
  Unreachable:        1
  Qed:                3
  Alt-Ergo 2.6.0:     1 (13ms)
```

Example swap.c (1/4)

```
[kernel] Parsing swap.c (with preprocessing)
[wp] Running WP plugin...
[wp] Warning: Missing RTE guards
[wp] 4 goals scheduled
[wp] Proved goals:    6 / 6
  Terminating:       1
  Unreachable:       1
  Qed:               3
  Alt-Ergo 2.6.0:    1 (13ms)
```

Example swap.c (1/4)

```
Function swap

Goal Post-condition 'A' in 'swap':
Let x = Mint_2[a_1].
Let x_1 = Mint_1[b].
Let x_2 = Mint_0[a].
Assume {
  Type: is_sint32(Mint_1[a]) /\ is_sint32(x_1) /\ is_sint32(x_2) /\
      is_sint32(Mint_0[b]) /\ is_sint32(x).
  (* Heap *)
  Type: (region(a.base) <= 0) /\ (region(b.base) <= 0) /\ linked(Malloc_0) /\
      cinits(Init_0).
  Have: (Malloc_1 = Malloc_0) /\ (Mint_2 = Mint_1) /\ (a_1 = a) /\ (b_1 = b).
  (* Pre-condition *)
  Have: valid_rw(Malloc_1, a_1, 1) /\ valid_rw(Malloc_1, b_1, 1).
  (* Pre-condition *)
  Have: valid_rw(Malloc_1, a_1, 1) /\ valid_rw(Malloc_1, b_1, 1).
  (* Initializer *)
  Init: x = tmp_0.
  Have: (Init_0[a_1 <- true] = Init_1) /\
      (Mint_2[a_1 <- Mint_2[b_1]] = Mint_3).
  Have: (Init_1[b_1 <- true] = Init_2) /\ (Mint_3[b_1 <- tmp_0] = Mint_0).
}
Prove: x_2 = x_1.
Prover Alt-Ergo 2.6.0 returns Valid (13ms) (32)
```

Example swap.c (1/4)

```
Goal Post-condition 'B' in 'swap':
Let x = Mint_2[a_1].
Let x_1 = Mint_1[a].
Let x_2 = Mint_0[b].
Assume {
  Type: is_sint32(x_1) /\ is_sint32(Mint_1[b]) /\ is_sint32(Mint_0[a]) /\
      is_sint32(x_2) /\ is_sint32(x).
  (* Heap *)
  Type: (region(a.base) <= 0) /\ (region(b.base) <= 0) /\ linked(Malloc_0) /\
      cinits(Init_0).
  Have: (Malloc_1 = Malloc_0) /\ (Mint_2 = Mint_1) /\ (a_1 = a) /\ (b_1 = b).
  (* Pre-condition *)
  Have: valid_rw(Malloc_1, a_1, 1) /\ valid_rw(Malloc_1, b_1, 1).
  (* Pre-condition *)
  Have: valid_rw(Malloc_1, a_1, 1) /\ valid_rw(Malloc_1, b_1, 1).
  (* Initializer *)
  Init: x = tmp_0.
  Have: (Init_0[a_1 <- true] = Init_1) /\
      (Mint_2[a_1 <- Mint_2[b_1]] = Mint_3).
  Have: (Init_1[b_1 <- true] = Init_2) /\ (Mint_3[b_1 <- tmp_0] = Mint_0).
}
Prove: x_2 = x_1.
Prover Qed returns Valid
```

Example swap.c (1/4)

```
Goal Assigns (file swap.c, line 5) in 'swap' (1/2):
Effect at line 12
Let x = Mint_2[a].
Assume {
  Type: is_sint32(Mint_0[a_1]) /\ is_sint32(Mint_0[b]) /\
      is_sint32(Mint_1[a_1]) /\ is_sint32(Mint_1[b]) /\ is_sint32(x).
  (* Heap *)
  Type: (region(a_1.base) <= 0) /\ (region(b.base) <= 0) /\
      linked(Malloc_0) /\ cinits(Init_0).
  (* Goal *)
  When: !invalid(Malloc_0, a, 1).
  Have: (Malloc_1 = Malloc_0) /\ (Mint_2 = Mint_0) /\ (a = a_1) /\ (b_1 = b).
  (* Pre-condition *)
  Have: valid_rw(Malloc_1, a, 1) /\ valid_rw(Malloc_1, b_1, 1).
  (* Pre-condition *)
  Have: valid_rw(Malloc_1, a, 1) /\ valid_rw(Malloc_1, b_1, 1).
  (* Initializer *)
  Init: x = tmp_0.
}
Prove: (a = a_1) \/ (b = a).
Prover Qed returns Valid
```

Example swap.c (1/4)

```
Goal Assigns (file swap.c, line 5) in 'swap' (2/2):
Effect at line 13
Assume {
  Type: is_sint32(Mint_0[a]) /\ is_sint32(Mint_0[b_1]) /\
      is_sint32(Mint_1[a]) /\ is_sint32(Mint_1[b_1]) /\
      is_sint32(Mint_2[a_1]).
  (* Heap *)
  Type: (region(a.base) <= 0) /\ (region(b_1.base) <= 0) /\
      linked(Malloc_0) /\ cinits(Init_0).
  (* Goal *)
  When: !invalid(Malloc_0, b, 1).
  Have: (Malloc_1 = Malloc_0) /\ (a_1 = a) /\ (b = b_1).
  (* Pre-condition *)
  Have: valid_rw(Malloc_1, a_1, 1) /\ valid_rw(Malloc_1, b, 1).
  (* Pre-condition *)
  Have: valid_rw(Malloc_1, a_1, 1) /\ valid_rw(Malloc_1, b, 1).
}
Prove: (b = a) \/ (b = b_1).
Prover Qed returns Valid
```

Current Summary

1. Introduction of notations
   - First example on proving versus testing
   - Second example on proving versus testing
   - Partial correctness but also runtime error
2. Correctness by contract
   - WP calculus
   - Simple example 1
3. Writing a contract
   - " A la Floyd" Verification Conditions for Contract
   - "A la Hoare" Verification Conditions for Contract
4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c

Summary

4 Examples
- Simple annotation (I)
- Simple annotation (II)
- Simple annotation(III)
- Simple annotation(IV)

5 Mechanizing the contract checking

6 Transforming predicates
- Hoare Logic for PC
- Examples in ACSL
- Définition et propriétés du calcul wp

7 Using predicate transformers for checking contracts (next episod)

8 Memory Models in Frama-c

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

10 Generation of Verification Conditions
- WP calculus in Frama-c
- First annotation
- Second annotation

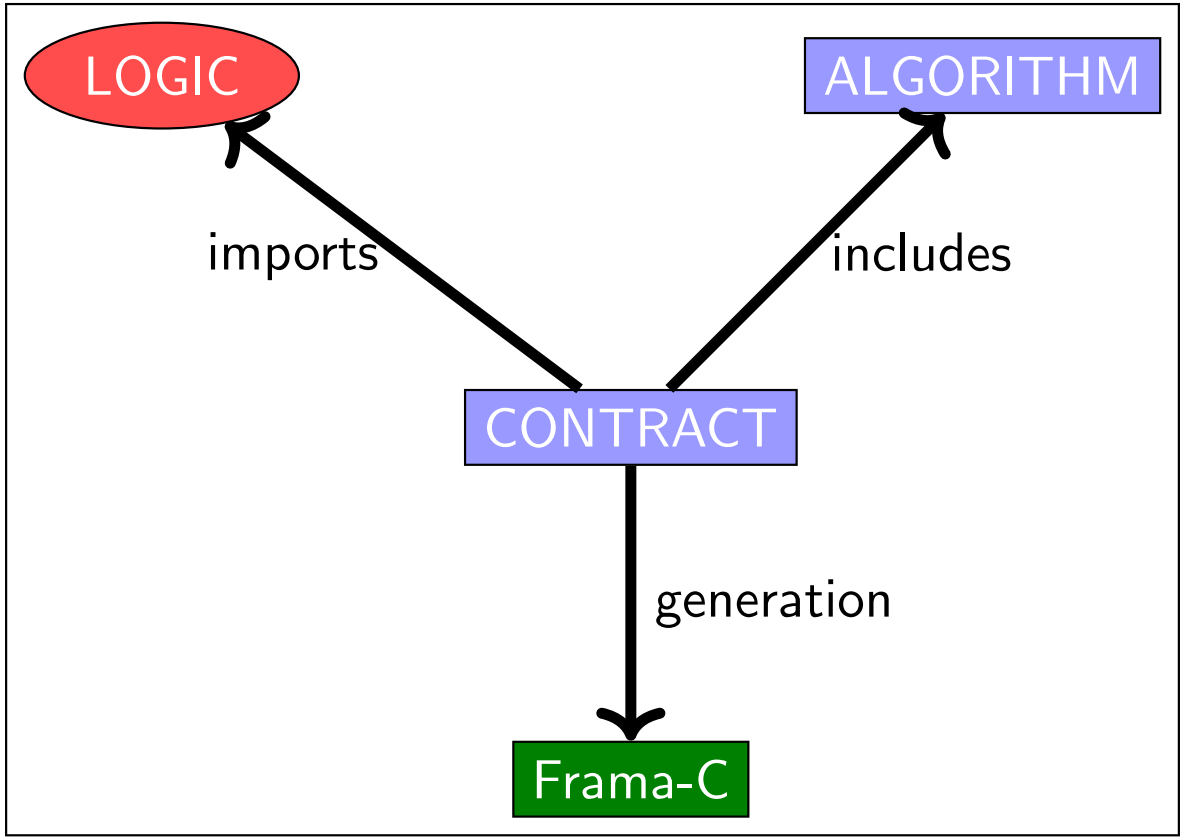
11 Memory Models in Frama-c

12 Logic Specification

13 Organisation of the verification process

14 Conclusion

Tryptic Pattern



Defining domain properties in logical theory

- predicate

Predicates - Logic - Lemma (1)

(Predicate)

Listing 76 – 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 77 – 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 78 – 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
  ==> mathfib(n) ==  mathfib(n-1)+mathfib(n-2);
  @ } */
```

(Definition of gcd)

Listing 79 – 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) ==> is_gcd(a,b,d); @}
@*/
```

Predicates - Logic - Lemma (3)

(Definition of function odd/even)

Listing 80 – 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);
   }
*/
```

wp,wlp,loop . . .

STATES l'ensemble des états sur l'ensemble X des variables, S une instruction de programme sur X, Q une propriété d'éttats, $s \in STATES$.

$$wp(S)(Q)(s) \stackrel{def}{=} \left(\begin{array}{l} \forall t \in STATES : \mathcal{D}(S)(s) = t \Rightarrow tQ(t) \\ \wedge \\ \exists t \in STATES : \mathcal{D}(S)(s) = t \end{array} \right) \ldots$$

wp,wlp,loop . . .

STATES l'ensemble des états sur l'ensemble X des variables, S une instruction de programme sur X, Q une propriété d'éttats, $s \in STATES$.

$$wp(S)(Q)(s) \stackrel{def}{=} \left(\begin{array}{l} \forall t \in STATES : \mathcal{D}(S)(s) = t \Rightarrow tQ(t) \\ \wedge \\ \exists t \in STATES : \mathcal{D}(S)(s) = t \end{array} \right) \quad \ldots \quad \ldots$$

$$wlp(S)(Q)(s) \stackrel{def}{=} (\forall t \in STATES : \mathcal{D}(S)(s) = t \Rightarrow Q(t))$$

. . . . . .

$$wlp(S)(Q)(s) \equiv \overline{(\exists t \in STATES : \mathcal{D}(S)(s) = t \wedge \overline{Q}(t))} \quad \ldots\ldots$$

wp,wlp,loop ...

STATES l'ensemble des états sur l'ensemble X des variables, S une instruction de programme sur X, Q une propriété d'éttats, $s \in STATES$.

$$wp(S)(Q)(s) \stackrel{def}{=} \left(\begin{array}{l} \forall t \in STATES : \mathcal{D}(S)(s) = t \Rightarrow tQ(t) \\ \wedge \\ \exists t \in STATES : \mathcal{D}(S)(s) = t \end{array} \right) \quad \ldots \quad \ldots$$

$$wlp(S)(Q)(s) \stackrel{def}{=} (\forall t \in STATES : \mathcal{D}(S)(s) = t \Rightarrow Q(t))$$

$$wlp(S)(Q)(s) \equiv \overline{(\exists t \in STATES : \mathcal{D}(S)(s) = t \wedge \overline{Q}(t))} \quad \ldots\ldots \quad \ldots\ldots$$

- $loop(S) \equiv \overline{(\exists t \in STATES : \mathcal{D}(S)(s) = t}$ (ensemble des états qui ne permettent pas à S de terminer)
- $wp(S)(Q)(s) \equiv wlp(S)(Q)(s) \wedge \overline{loop(S)}(s)$

Weakest Liberal Precondition of S for P

$$\{S\}P(x) \stackrel{def}{=} \forall x_f . x \xrightarrow{\mathsf{S}} x_f \Rightarrow \mathsf{P}(x_f)$$

Logique de Hoare Correction Totale

⊠ Definitiontriplets de Hoare Correction Totale

$[P]\mathbf{S}[Q] \stackrel{def}{=} P \Rightarrow wp(S)(Q)$

Logique de Hoare Correction Totale

⊠ Definitiontriplets de Hoare Correction Totale

$[P]\mathbf{S}[Q] \stackrel{def}{=} P \Rightarrow wp(S)(Q)$

⊠ Definition(Axiomes et règles d'inférence)

- Axiome d'affectation : $[P(e/x)]$**X :=E(X)**$[P]$.
- Axiome du saut : $[P]$**skip**$[P]$.
- Règle de composition : Si $[P]\mathbf{S}_1[R]$ et $[R]\mathbf{S}_2[Q]$, alors $[P]\mathbf{S}_1$ **;**$\mathbf{S}_2[Q]$.
- Si $[P \wedge B]\mathbf{S}_1[Q]$ et $[P \wedge \neg B]\mathbf{S}_2[Q]$, alors $[P]$**if B then S**$_1$ **then S**$_2$ **fi**$[Q]$.
- Si $[P(n{+}1)]\mathbf{S}[P(n)]$, $P(n{+}1) \Rightarrow b$, $P(0) \Rightarrow \neg b$, alors $[\exists n \in \mathbb{N}.P(n)]$**while B do S od**$[P(0)]$.
- Règle de renforcement/affaiblissement : Si $P' \Rightarrow P$, $[P]\mathbf{S}[Q]$, $Q \Rightarrow Q'$, alors $[P']\mathbf{S}[Q']$.

Correction

:
Si $[P]$**S**$[Q]$ est dérivé selon les règles ci-dessus, alors $P \Rightarrow wp(S)Q)$.

- $[P(e/x)]$**X :=E(X)**$[P]$ est valide : $wp(X := E)(P)/x = P(e/x)$.
- $[\exists n \in \mathbb{N}.P(n)]$**while B do S od**$[P(0)]$: si s est un état de P(n) alors au bout de n boucles on atteint un état s_f tel que P(0) est vrai en s_f.

Complétude

:
Si $P \Rightarrow wp(S)(Q)$, alors il existe une preuve de $[P]\mathbf{S}[Q]$ construites avec les règles ci-dessus,

- $P \Rightarrow wp(X := E(X))(Q)$: $P \Rightarrow Q(e/x)$ et $[Q(e/x)]$**X :=E(X)**$[Q]$ constituent une preuve.
- $P \Rightarrow wp(while)(Q)$:
  - On construit la suite de $P(n)$ en définissant $P(n) = W_n$.
  - On vérifie que cela vérifie la règle du while.

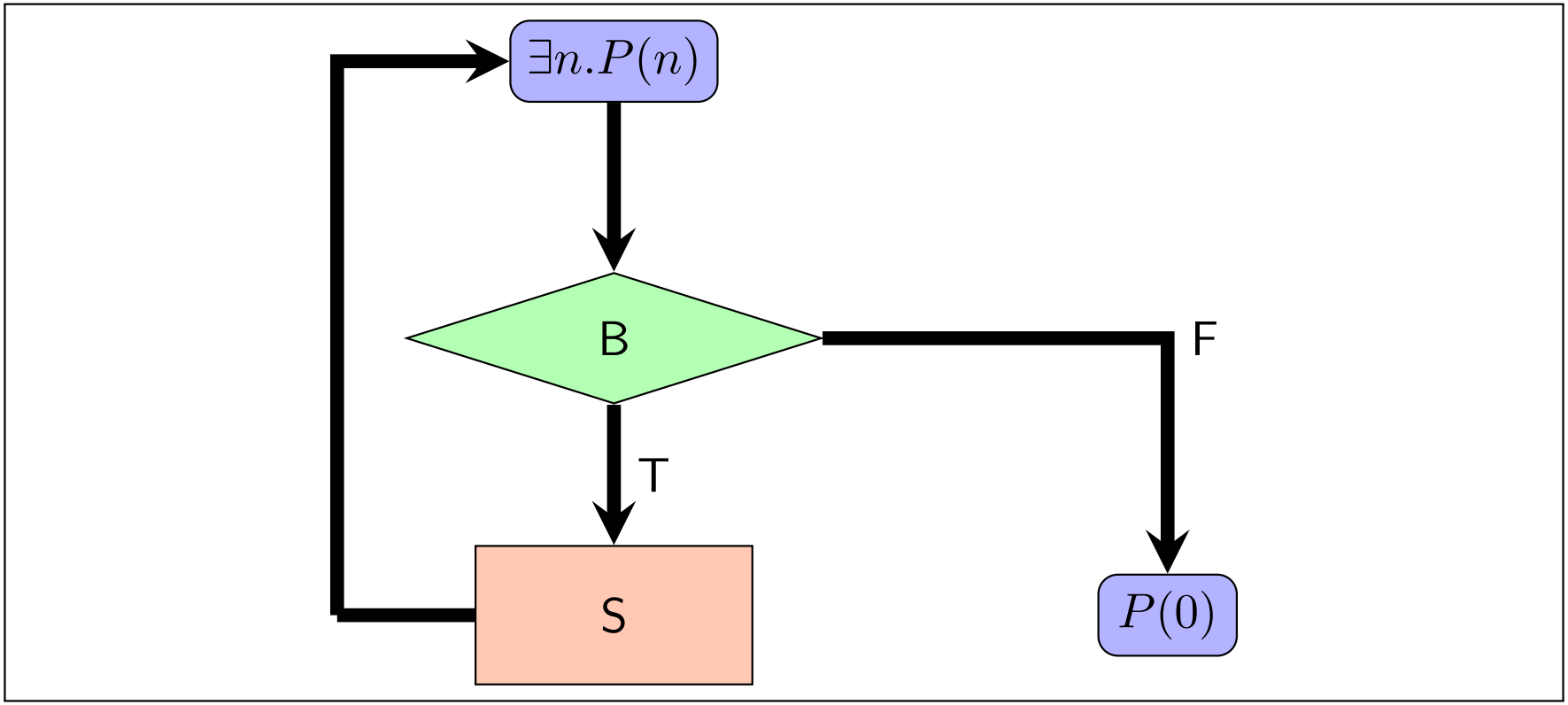
Introducing a variant

Introducing a variant

- assignment and skip are terminating
- If S1 and S2 are terminating, then S1 ;S2 is terminating and if(B,S1,S2) is terminating

Introducing a variant

- assignment and skip are terminating
- If S1 and S2 are terminating, then S1 ;S2 is terminating and if(B,S1,S2) is terminating



Explanation with the least fixed-point

- An assertion Q over $\mathbb{S}$ maps states of $\mathbb{S}$ to booleans of $\mathbb{B}$: $Q \in \mathbb{S} \longrightarrow \mathbb{B}$
- $\mathbb{A} \stackrel{def}{=} \mathbb{S} \longrightarrow \mathbb{B}$
- $\Rightarrow$ defines a partial ordering over $\mathbb{A}$
- $(\mathbb{A}, \Rightarrow)$ is a complete lattice with a smallest element $\perp$, a greatest element $\top$.
- For any Q in $\mathbb{A}$, $set(Q) = \{s \in \mathbb{S} | Q(s) = 1\}$ and $(\mathcal{P}(\mathbb{S}), \subseteq)$ is a complete lattice and $Q1 \Rightarrow Q2 \equiv set(Q1) \subseteq set(Q2)$
- $w(B)(S) \stackrel{def}{=}$ while B do S od
- $F_{B,S}(X) = B \wedge wp(S)(X) \vee \neg B \wedge Q$
- $F_{B,S}(\bigcup_{i \in I} X_i) = \bigcup_{i \in I} F_{B,S}(A_i)$ for any index sequence of assertions A_i $i \in I$.
- $F_{B,S}$ is a continuous map and has a least fixed-point by Kleene's theorem.
- $lfp(F_{B,S}) = \bigcup_{i \in \mathbb{N}} F^i_{B,S}$ with $F^0_{B,S} = \perp$ and $F^{i+1}_{B,S} = F_{B,S}(F^i_{B,S})$ for any $i \in \mathbb{N}$.

Defining approximations of while loop

- $F^0_{B,S} = \bot$: the first approximation means that the loop is diverging
- $F^{i+1}_{B,S} = B \wedge wp(S)(F^i_{B,S}) \vee \neg B \wedge Q$
- $P(0) \stackrel{def}{=} F^1_{B,S} \equiv \neg B \wedge Q$
- $\ldots$
- $P(i{+}1) \stackrel{def}{=} B \wedge wp(S)(P(i))$

The sequence of assertions $P(i)$ satisfies the following properties

- $\forall i \in \mathbb{N}, [P(i{+}1]S[P(i)]$
- $\forall i \in \mathbb{N}, [P(i{+}1] \rightarrow B$
- $P(0) \Rightarrow \neg B$

By the rule of Hoare Logic for Total Correctness, we get the following property $[\exists i \in \mathbb{N}, P(i]w(B)(S)[P(0)]$.

Loop termination

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

(Variant)

Listing 81 – 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 82 – 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;
}
```

Recursive Call

(Variant)

Listing 83 – project-divers/variant4.c

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

Terminates

(Terminates)

Listing 84 – project-divers/terminates1.c

```
/*@
  requires P;
  terminates Q;
*/
void f(...);
```

- Frama-C génère la condition de vérification VC terminates
- $P \Rightarrow Q$

- – 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

Current Summary

1. Introduction of notations
   - First example on proving versus testing
   - Second example on proving versus testing
   - Partial correctness but also runtime error
2. Correctness by contract
   - WP calculus
   - Simple example 1
3. Writing a contract
   - " A la Floyd" Verification Conditions for Contract
   - "A la Hoare" Verification Conditions for Contract
4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c

Summary

4 Examples
- Simple annotation (I)
- Simple annotation (II)
- Simple annotation(III)
- Simple annotation(IV)

5 Mechanizing the contract checking

6 Transforming predicates
- Hoare Logic for PC
- Examples in ACSL
- Définition et propriétés du calcul wp

7 Using predicate transformers for checking contracts (next episod)

8 Memory Models in Frama-c

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

10 Generation of Verification Conditions
- WP calculus in Frama-c
- First annotation
- Second annotation

11 Memory Models in Frama-c

12 Logic Specification

13 Organisation of the verification process

14 Conclusion

Main steps

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

The power two function

(power2.h)

Listing 85 – 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 mathpeven_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;
@ } */
```

The power two function

(power2.h)

Listing 86 – 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
```

Factorial function

(power2.h)

Listing 87 – 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;
        @ loop variant x-k;
*/
  while (k<x)
    {
    ok=k;ocv=cv;ocw=cw;
      k=ok+1;
      cv=ocv+ocw+1;
      cw=ocw+2;

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

Current Summary

1. Introduction of notations
   - First example on proving versus testing
   - Second example on proving versus testing
   - Partial correctness but also runtime error
2. Correctness by contract
   - WP calculus
   - Simple example 1
3. Writing a contract
   - " A la Floyd" Verification Conditions for Contract
   - "A la Hoare" Verification Conditions for Contract
4. Examples
   - Simple annotation (I)
   - Simple annotation (II)
   - Simple annotation(III)
   - Simple annotation(IV)
5. Mechanizing the contract checking
6. Transforming predicates
   - Hoare Logic for PC
   - Examples in ACSL
   - Définition et propriétés du calcul wp
7. Using predicate transformers for checking contracts (next episod)
8. Memory Models in Frama-c

Summary

4 Examples
- Simple annotation (I)
- Simple annotation (II)
- Simple annotation(III)
- Simple annotation(IV)

5 Mechanizing the contract checking

6 Transforming predicates
- Hoare Logic for PC
- Examples in ACSL
- Définition et propriétés du calcul wp

7 Using predicate transformers for checking contracts (next episod)

8 Memory Models in Frama-c

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

10 Generation of Verification Conditions
- WP calculus in Frama-c
- First annotation
- Second annotation

11 Memory Models in Frama-c

12 Logic Specification

13 Organisation of the verification process

14 Conclusion

Principles for RTE

- Plugin -rte adds specifi 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*)