

Frama-C WP

Séminaire CAP'TRONIC

Virgile Prevosto
virgile.prevosto@cea.fr

June 18th, 2015



Basic function contract

A little bit of background

ACSL and WP

Loops

Background

Loop termination

Advanced contracts

Behaviors

User-defined predicates

Example

Conclusion

```
(long n)
{ for (i = 0; i < n; i++)
  C[i] = 0;
  tmp2 = 0;
  // ...
}
```

```
tmp2[i] = (i < (N-1) ? tmp[i] : 0); else if (tmp1[i] >= 0) { i < (N-1) ? tmp2[i] = (i < (N-1) ? 0 : tmp1[i]) : 0; } // Then the second pass looks like the first one:
tmp1[i] = 0; k = 0; k++ } tmp1[i] = mc2[i][k] * tmp2[k]; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is: *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 *MC1
i = 1; tmp1[i] >= 1; /* Final rounding: tmp2[i] is now represented on 9 bits. */ if (tmp1[i] < -255) m2[i] = -255; else if (tmp1[i] > 255) m2[i] = 255; else m2[i] = tmp1[i];
}
```



Basic function contract

A little bit of background

ACSL and WP

Loops

Background

Loop termination

Advanced contracts

Behaviors

User-defined predicates

Example

Conclusion

```
(long n)
{ for (i = 0; i < n; i++)
  C[i] = 0;
  tmp2 = ...
  // ...
}
```

```
tmp2[i] = (i <= (Nb1 - 1)) ? tmp1[i] : 0; else tmp2[i] = (i <= (Nb1 - 1)) ? 0 : tmp1[i]; /* Then the second pass looks like the first one: */
tmp1[i] = 0; k = 0; k++ tmp1[i][k] = mc2[i][k] * tmp2[k]; /* The [i][k] coefficient of the matrix product MC2*TMP2, that is, *MC2*[i][MC1*M1] = MC2*[M1]*[MC1*
i = 1; tmp1[i] >= 1; /* Final rounding: tmp2[i] is now represented on 9 bits. */ if (tmp1[i] < -256) m2[i] = -256; else if (tmp1[i] > 255) m2[i] = 255; else m2[i] = tm
```



Hoare Logic

```
/*@ requires R;  
    ensures E; */  
int f(int* x) {
```

- ▶ Hoare Triples:

$$\{P\}S\{Q\}$$

- ▶ Weakest Preconditions:

$$\forall P, (P \Rightarrow wp(S, Q)) \\ \Rightarrow \{P\}S\{Q\}$$

- ▶ Proof Obligation (PO):

$$R \Rightarrow wp(\text{Body}, E)$$

```
S_1;
```

```
S_2;
```

```
}
```



```
/*@ requires R;
    ensures E; */
int f(int* x) {
```

- ▶ Hoare Triples:

$$\{P\}S\{Q\}$$

- ▶ Weakest Preconditions:

$$\begin{aligned} \forall P, (P \Rightarrow wp(S, Q)) \\ \Rightarrow \{P\}S\{Q\} \end{aligned}$$

```
S_1;
```

```
S_2;
```

```
/*@ assert E; */
}
```

- ▶ Proof Obligation (PO):

$$R \Rightarrow wp(\text{Body}, E)$$



```
/*@ requires R;  
    ensures E; */  
int f(int* x) {
```

- ▶ Hoare Triples:

$$\{P\}S\{Q\}$$

- ▶ Weakest Preconditions:

$$\begin{aligned} \forall P, (P \Rightarrow wp(S, Q)) \\ \Rightarrow \{P\}S\{Q\} \end{aligned}$$

```
S_1;
```

```
/*@ assert wp(S_2,E); */  
S_2;
```

- ▶ Proof Obligation (PO):

$$R \Rightarrow wp(\text{Body}, E)$$

```
/*@ assert E; */  
}
```



```
/*@ requires R;  
    ensures E; */  
int f(int* x) {
```

- ▶ Hoare Triples:

$$\{P\}S\{Q\}$$

- ▶ Weakest Preconditions:

$$\forall P, (P \Rightarrow wp(S, Q)) \\ \Rightarrow \{P\}S\{Q\}$$

- ▶ Proof Obligation (PO):

$$R \Rightarrow wp(\text{Body}, E)$$

```
/*@ assert  
    wp(S_1, wp(S_2, E)); */  
S_1;
```

```
/*@ assert wp(S_2, E); */  
S_2;
```

```
/*@ assert E; */  
}
```

(long no
[for it
C1); if (0
tmp2 =
se of the

tmp2[0] = (t <= (n-1) ? t : 0; else if (tmp1[0] >= (t <= (n-1) ? tmp2[0] : (t <= (n-1) ? 0 : 0; else tmp2[0] = tmp1[0]; /* Then the second part deals like the first one. We have
tmp1[0] = 0; k = 0; k++ tmp1[0] = mc2[0][k] * tmp2[k][0]; /* The [k][0] coefficient of the matrix product MC2*TMP2, that is, *MC2*[TMP1] = MC2*[MC1*M1] = MC2*M1 *MC1
l = 1; tmp1[0] >= 1; /* Final rounding: tmp2[0] is now represented on 9 bits. *if (tmp1[0] < -256) tmp2[0] = -256; else if (tmp1[0] > 255) tmp2[0] = 255; else tmp2[0] = tmp1[0];



Credits

- ▶ Loïc Correnson
- ▶ Zaynah Dargaye
- ▶ Anne Pacalet
- ▶ François Bobot
- ▶ a few others

Basic usage

- ▶ `frama-c-gui -wp [-wp-rte] file.c`
- ▶ WP tab on the GUI
- ▶ Inspect (failed) proof obligation
- ▶ <http://frama-c.com/download/wp-manual.pdf>



Setting values

Example

```
// swap the content of both arguments
void swap(int* p, int* q) {
    int tmp = *q;
    *q = *p;
    *p = tmp;
}
```

Demo

» next



Specification for swap

```
/*@
  requires \valid(p) && \valid(q);
  ensures \old(*p) == *q && \old(*q) == *p;
*/
void swap(int* p, int* q) {
  int tmp = *q;
  *q = *p;
  *p = tmp;
}
```

(long n;
for (i = 0;
i < n; i++)
tmp2 =
... of the

tmp2[i] = (t <= 0 ? (n1 - t)) else if (tmp1[i] >= (t <= (n1 - t) ? (t <= (n1 - t) - 1) : 1) else tmp2[i] = tmp1[i]; /* Then the second pass looks like the first one: "for (i = 0; i < n; i++) tmp1[i] = mc2[i][k] * tmp2[k];" /* The [i][k] coefficient of the matrix product MC2*TMP2, that is, *MC2*[i][TMP1] = MC2*[i][MC1*M1] = MC2*[i][MC1 - 1] = 1. tmp1[0][i] >= 1. /* Final rounding: tmp2[0][i] is now represented on 9 bits. *if (tmp1[0][i] < -256) m2[0][i] = -256; else if (tmp1[0][i] > 255) m2[0][i] = 255; else m2[0][i] = tmp1[0][i];



Specification for swap

```
/*@
  requires \valid(p) && \valid(q);
  ensures \old(*p) == *q && \old(*q) == *p;
*/
void swap(int* p, int* q) {
  int tmp = *q;
  *q = *p;
  *p = tmp;
}
```

This introduces a pre-condition



Specification for swap

```
/*@
  requires \valid(p) && \valid(q);
  ensures \old(*p) == *q && \old(*q) == *p;
*/
void swap(int* p, int* q) {
  int tmp = *q;
  *q = *p;
  *p = tmp;
}
```

This introduces a post-condition



Specification for swap

```
/*@
  requires \valid(p) && \valid(q);
  ensures \old(*p) == *q && \old(*q) == *p;
*/
void swap(int* p, int* q) {
  int tmp = *q;
  *q = *p;
  *p = tmp;
}
```

swap needs valid locations (pointers you can dereference)



Specification for swap

```
/*@
  requires \valid(p) && \valid(q);
  ensures \old(*p) == *q && \old(*q) == *p;
*/
void swap(int* p, int* q) {
  int tmp = *q;
  *q = *p;
  *p = tmp;
}
```

In post-conditions, you can refer to the old state (at the beginning of the function)



Function Calls

```
/*@ requires R_1;  
    ensures E_1;  
    assigns A;
```

```
*/  
void g();
```

```
/*@ requires R_2;  
    ensures E_2;
```

```
*/  
void f() {  
    S_1;  
    g();  
    S_2;  
}
```

- ▶ Contract as a cut

- ▶ First PO: f must call g in a correct context:

$$R_2 \Rightarrow wp(S_1, R_1)$$

- ▶ Second PO: State after g has the desired properties:

$$\forall \text{State}, E_1 \Rightarrow wp(S_2, E_2)$$

- ▶ Must specify effects (Frame rule)

$$\forall x \in \text{State} \setminus A, g \text{ does not change } x$$



Function Calls

```
/*@ requires R_1;  
    ensures E_1;  
    assigns A;
```

```
*/  
void g();
```

```
/*@ requires R_2;  
    ensures E_2;
```

```
*/  
void f() {  
    S_1;  
    g();  
    S_2;  
}
```

- ▶ Contract as a cut

- ▶ First PO: f must call g in a correct context:

$$R_2 \Rightarrow wp(S_1, R_1)$$

- ▶ Second PO: State after g has the desired properties:

$$\forall \text{State}, E_1 \Rightarrow wp(S_2, E_2)$$

- ▶ Must specify effects (Frame rule)

$$\forall x \in \text{State} \setminus A, g \text{ does not change } x$$



Function Calls

```
/*@ requires R_1;  
    ensures E_1;  
    assigns A;
```

```
*/  
void g();
```

```
/*@ requires R_2;  
    ensures E_2;
```

```
*/  
void f() {  
    S_1;  
    g();  
    S_2;  
}
```

- ▶ Contract as a cut

- ▶ First PO: f must call g in a correct context:

$$R_2 \Rightarrow wp(S_1, R_1)$$

- ▶ Second PO: State after g has the desired properties:

$$\forall State, E_1 \Rightarrow wp(S_2, E_2)$$

- ▶ Must specify effects (Frame rule)

$$\forall x \in State \setminus A, g \text{ does not change } x$$



Function Calls

```
/*@ requires R_1;  
    ensures E_1;  
    assigns A;
```

```
*/  
void g();
```

```
/*@ requires R_2;  
    ensures E_2;
```

```
*/  
void f() {  
    S_1;  
    g();  
    S_2;  
}
```

- ▶ Contract as a cut
- ▶ First PO: f must call g in a correct context:

$$R_2 \Rightarrow wp(S_1, R_1)$$

- ▶ Second PO: State after g has the desired properties:

$$\forall \text{State}, E_1 \Rightarrow wp(S_2, E_2)$$

- ▶ Must specify effects (Frame rule)

$$\forall x \in \text{State} \setminus A, g \text{ does not change } x$$



Function Calls

```
/*@ requires R_1;  
    ensures E_1;  
    assigns A;
```

```
*/  
void g();
```

```
/*@ requires R_2;  
    ensures E_2;
```

```
*/  
void f() {  
    S_1;  
    g();  
    S_2;  
}
```

- ▶ Contract as a cut

- ▶ First PO: f must call g in a correct context:

$$R_2 \Rightarrow wp(S_1, R_1)$$

- ▶ Second PO: State after g has the desired properties:

$$\forall State, E_1 \Rightarrow wp(S_2, E_2)$$

- ▶ Must specify effects (Frame rule)

$$\forall x \in State \setminus A, g \text{ does not change } x$$



Function call: example

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

```
// permutation a -> b -> c -> a
```

```
void permut(int* a, int *b, int* c) {
    swap(a,b);
    swap(a,c);
}
```

Demo

▶ next



Function call: contracts

```
/*@ requires \valid(a) && \valid(b);
   assigns *a,*b;
   ensures \old(*a) == *b && \old(*b) == *a;
*/
void swap(int* a, int *b);

void permut(int* a, int *b, int* c) {
    swap(a,b);
    swap(a,c);
}
```



Function call: contracts

```
/*@ requires \valid(a) && \valid(b);
   assigns *a,*b;
   ensures \old(*a) == *b && \old(*b) == *a;
*/
void swap(int* a, int *b);

void permut(int* a, int *b, int* c) {
    swap(a,b);
    swap(a,c);
}
```

Indicates that swap only modifies content of its two arguments



Function call: contracts

```

/*@ requires \valid(a) && \valid(b);
   assigns *a,*b;
   ensures \old(*a) == *b && \old(*b) == *a;
*/
void swap(int* a, int *b);

void permut(int* a, int *b, int* c) {
    swap(a,b);
    swap(a,c);
}

```

swap's contracts indicates that *c is not modified by this call



Contract of permut

```

/*@ requires \valid(a);
   requires \valid(b);
   requires \valid(c);
   requires \separated(a,b,c)};
   assigns *a, *b, *c;
   ensures \at(*a,Pre) == *b;
   ensures \at(*b,Pre) == *c;
   ensures \at(*c,Pre) == *a;
*/
void permut(int* a, int *b, int* c) {
    swap(a,b);
    swap(a,c);
}

```



Contract of permut

```

/*@ requires \valid(a);
    requires \valid(b);
    requires \valid(c);
    requires \separated(a,b,c)};
    assigns *a, *b, *c;
    ensures \at(*a,Pre) == *b;
    ensures \at(*b,Pre) == *c;
    ensures \at(*c,Pre) == *a;
*/
void permut(int* a, int* b, int* c) {
    swap(a,b);
    swap(a,c);
}

```

permutation will work only if the pointers do not point to the same area



tmp2[j][i] = -1 << (NBI - 1); else if (tmp1[j][i]) >= (1 << (NBI - 1)) - tmp2[j][i] = (1 << (NBI - 1)) - 1; else tmp2[j][i] = tmp1[j][i]; /* Then the second pass. Looks like the first one. */
 tmp1[j][i] = 0; k < 8; k++) tmp1[j][i] += mc2[j][k] * tmp2[k][i]; /* The [i,j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*tm1*(MCI, 2)



```
/*@ requires R;  
    ensures E;
```

```
*/  
void f() {  
  S_1;
```

```
while(e) { B }  
S_2;  
}
```

- ▶ Need to capture effects of **all** loop steps
- ▶ Inductive loop invariant:
 - ▶ Holds at the beginning (after 0 step). PO is $R \Rightarrow wp(S_1, I)$
 - ▶ If it holds after n steps, it holds after $n + 1$ steps. PO is $\forall State, I \wedge e \Rightarrow wp(B, I)$
 - ▶ Must imply the post-condition. PO is $\forall State, I \wedge \neg e \Rightarrow wp(S_2, E)$
- ▶ Specify effects of the loop:
 - $\forall x \in$
 - $State \setminus A, B$ does not change x



```
/*@ requires R;
    ensures E;

*/
void f() {
  S_1;

  /*@ loop invariant I;

*/
  while(e) { B }
  S_2;
}
```

- ▶ Need to capture effects of all loop steps
- ▶ Inductive loop invariant:
 - ▶ Holds at the beginning (after 0 step). PO is $R \Rightarrow wp(S_1, I)$
 - ▶ If it holds after n steps, it holds after $n + 1$ steps. PO is $\forall State, I \wedge e \Rightarrow wp(B, I)$
 - ▶ Must imply the post-condition. PO is $\forall State, I \wedge \neg e \Rightarrow wp(S_2, E)$
- ▶ Specify effects of the loop: $\forall x \in State \setminus A, B$ does not change x



```
/*@ requires R;
    ensures E;

*/
void f() {
  S_1;

  /*@ loop invariant I;

*/
  while(e) { B }
  S_2;
}
```

- ▶ Need to capture effects of all loop steps
- ▶ Inductive loop invariant:
 - ▶ Holds at the beginning (after 0 step). PO is $R \Rightarrow wp(S_1, I)$
 - ▶ If it holds after n steps, it holds after $n + 1$ steps. PO is $\forall State, I \wedge e \Rightarrow wp(B, I)$
 - ▶ Must imply the post-condition. PO is $\forall State, I \wedge \neg e \Rightarrow wp(S_2, E)$
- ▶ Specify effects of the loop:
 - $\forall x \in$
 - $State \setminus A, B$ does not change x



```

/*@ requires R;
    ensures E;

*/
void f() {
  S_1;

  /*@ loop invariant I;

*/
  while (e) { B }
  S_2;
}

```

- ▶ Need to capture effects of all loop steps
- ▶ Inductive loop invariant:
 - ▶ Holds at the beginning (after 0 step). PO is $R \Rightarrow wp(S_1, I)$
 - ▶ If it holds after n steps, it holds after $n + 1$ steps. PO is $\forall State, I \wedge e \Rightarrow wp(B, I)$
 - ▶ Must imply the post-condition. PO is $\forall State, I \wedge \neg e \Rightarrow wp(S_2, E)$
- ▶ Specify effects of the loop:
 - $\forall x \in$
 - $State \setminus A, B$ does not change x



```
/*@ requires R;
    ensures E;

*/
void f() {
  S_1;

  /*@ loop invariant I;

*/
  while (e) { B }
  S_2;
}
```

- ▶ Need to capture effects of all loop steps
- ▶ Inductive loop invariant:
 - ▶ Holds at the beginning (after 0 step). PO is $R \Rightarrow wp(S_1, I)$
 - ▶ If it holds after n steps, it holds after $n + 1$ steps. PO is $\forall State, I \wedge e \Rightarrow wp(B, I)$
 - ▶ Must imply the post-condition. PO is $\forall State, I \wedge \neg e \Rightarrow wp(S_2, E)$
- ▶ Specify effects of the loop:
 - $\forall x \in$
 - $State \setminus A, B$ does not change x



```
/*@ requires R;
    ensures E;
*/
void f() {
  S_1;

  /*@ loop invariant I;
  loop assigns A;
  */
  while (e) { B }
  S_2;
}
```

- ▶ Need to capture effects of all loop steps
- ▶ Inductive loop invariant:
 - ▶ Holds at the beginning (after 0 step). PO is $R \Rightarrow wp(S_1, I)$
 - ▶ If it holds after n steps, it holds after $n + 1$ steps. PO is $\forall State, I \wedge e \Rightarrow wp(B, I)$
 - ▶ Must imply the post-condition. PO is $\forall State, I \wedge \neg e \Rightarrow wp(S_2, E)$
- ▶ Specify effects of the loop:
 - $\forall x \in$
 $State \setminus A, B$ does not change x



Loop example

```

/* return the maximal value found in m */
int max_array(int* a, int length) {
    int m = a[0];
    for (int i = 1; i<length; i++) {
        if (a[i] > m) m = a[i];
    }
    return m;
}

```

Demo

» next



Max array contract

```

/*@ requires length > 0;
    requires \valid(a+(0 .. length));
    ensures \forall integer i;
        0<=i<length ==> \bresult >= a[i];
    ensures \exists integer i;
        0<=i<length && \bresult == a[i];
*/
int max_array(int* a, int length) {

```



Max array contract

```

/*@ requires length > 0;
    requires \valid(a+(0 .. length));
    ensures \forall forall integer i;
        0<=i<length ==> \bresult >= a[i];
    ensures \exists exists integer i;
        0<=i<length && \bresult == a[i];
*/
int max_array(int* a, int length) {

```

Impose validity of a whole block of memory



Max array contract

```

/*@ requires length > 0;
    requires \valid(a+(0 .. length));
    ensures \forall forall integer i;
        0<=i<length ==> \bresult >= a[i];
    ensures \exists exists integer i;
        0<=i<length && \bresult == a[i];
*/
int max_array(int* a, int length) {

```

we want all i in the interval to verify the inequality



Max array contract

α α

a i

conversely, we want some i that is in the interval and verify the equality



Loop annotations

```

int max_array(int* a, int length) {
    int m = a[0];
    /*@
        loop invariant 0<=i<=length;
        loop invariant
            \forall integer j; 0<=j<i ==> m >= a[j];
        loop invariant
            \exists integer j; 0<=j<i && m == a[j];

        loop assigns i,m;

    */
    for (int i = 1; i<length; i++) {
        if (a[i] > m) m = a[i];
    }
    return m;
}
  
```



Loop annotations

```
int max_array(int* a, int length) {
    int m = a[0];
    /*@
        loop invariant 0<=i<=length;
        loop invariant
            \forall integer j; 0<=j<i ==> m >= a[j];
        loop invariant
            \exists integer j; 0<=j<i && m == a[j];

        loop assigns i,m;
```

```
*/
```

“structural” invariant giving indications on the control-flow of the program



Loop annotations

```

int max_array(int* a, int length) {
  int m = a[0];
  /*@
    loop invariant 0<=i<=length;
    loop invariant
      \forall integer j; 0<=j<i ==> m >= a[j];
    loop invariant
      \exists integer j 0<=j<i && m == a[j];

    loop assigns i,m;
  */

```

inequality is large, as it must also be preserved by the very last step of the loop



Loop annotations

```

int max_array(int* a, int length) {
  int m = a[0];
  /*@
    loop invariant 0<=i<=length;
    loop invariant
      \forall integer j; 0<=j<i ==> m >= a[j];
    loop invariant
      \exists integer j; 0<=j<i && m == a[j];

    loop assigns i,m;

  */

```

“functional” invariant establishing the property: m is the maximum seen so far



Loop annotations

```

int max_array(int* a, int length) {
  int m = a[0];
  /*@
    loop invariant 0<=i<=length;
    loop invariant
      \forall integer j; 0<=j<i ==> m >= a[j];
    loop invariant
      \exists integer j; 0<=j<i && m == a[j];

    loop assigns i, m;
  */

```

only m and i may change. In particular, content of a stays the same during the loop



Loop termination

Terminology

- ▶ **Partial correctness:** if the function terminates, it respects its specification
- ▶ **Total correctness:** the function terminates, and it respects its specification

Loop Variant

- ▶ Key idea: provide an upper bound for the number of remaining steps:
- ▶ Must stay positive (except possibly when exiting loop)
- ▶ Must decrease strictly at each step

» next



Loop example

```

/* return the maximal value found in m */
int max_array(int* a, int length) {
    int m = a[0];
    for (int i = 1; i < length; i++) {
        if (a[i] > m) m = a[i];
    }
    return m;
}

```

Demo



loop variant

```

int max_array(int* a, int length) {
    int m = a[0];
    /*@

        loop variant length - i;
    */
    for (int i = 1; i<length; i++) {
        if (a[i] > m) m = a[i];
    }
    return m;
}
  
```



loop variant

```
int max_array(int* a, int length) {
    int m = a[0];
    /*@
        loop variant length - i;
    */
    for (int i = 1; i < length; i++) {
        if (a[i] > m) m = a[i];
    }
    return m;
}
```

length-i is positive and strictly decreasing



Basic function contract

A little bit of background

ACSL and WP

Loops

Background

Loop termination

Advanced contracts

Behaviors

User-defined predicates

Example

Conclusion

```
(long n)
{ for (i = 0; i < n; i++)
  C[i] = 0;
  tmp2 = ...
  // ...
}
```

```
tmp2[i] = (i <= (N-1) ? tmp[i] : 0); if (i <= (N-1) ? tmp[i] : 0) tmp2[i] = tmp[i]; /* Then the second pass looks like the first one: */
tmp[i] = 0; k = 0; k++; tmp[i][k] = mc2[i][k] * tmp2[k]; /* The [i][k] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1
i = i+1; tmp[i][0] >= 1; /* Final rounding: tmp2[i][0] is now represented on 9 bits. */ if (tmp[i][0] < -256) m2[i][0] = -256; else if (tmp[i][0] > 255) m2[i][0] = 255; else m2[i][0] = tmp[i][0];
}
```



Specification by cases

- ▶ Global precondition (**requires**) and postcondition (**ensures**, **assigns**) applies to all cases
- ▶ Behaviors refine global contract in particular cases
- ▶ For each case (each **behavior**)
 - ▶ the subdomain is defined by **assumes** clause
 - ▶ can give additional constraints with local **requires** clauses
 - ▶ the behavior's postcondition is defined by **ensures**, **assigns** clauses
 - ▶ it must be ensured whenever **assumes** condition is true
- ▶ **complete behaviors** states that given behaviors cover all cases
- ▶ **disjoint behaviors** states that given behaviors do not overlap



Predicate and logic function definitions

- ▶ factorize often-used concepts into logic functions or predicates
- ▶ can be defined
 - ▶ with a direct definition
 - ▶ with a declaration and a set of axioms
 - ▶ inductively
- ▶ can be used in any annotation once defined

(long no
[for 0 <=
C1); if (0
tmp2 =
st of the

tmp2[0] = (t <= (Nb1 - 1)) else if (tmp1[0]) >= (t <= (Nb1 - 1)) tmp2[0] = (t <= (Nb1 - 1)) - 1; else tmp2[0] = tmp1[0]; /* Then the second pass looks like the first one: */
tmp1[0] = 0; k = 0; k++ tmp1[0] = mc2[0][k] * tmp2[k][0]; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is: *MC2*[i][MC1*M1] = MC2*[M1]*MC1*
l = 1; tmp1[0] >= 1; /* Final rounding: tmp2[0] is now represented on 9 bits. */ if (tmp1[0] < -256) tmp2[0] = -256; else if (tmp1[0] > 255) tmp2[0] = 255; else tmp2[0] = tmp1[0];



MJRTY Algorithm

- ▶ given an array, returns the element found in the majority of the cells if any
- ▶ in constant space
- ▶ proposed by Boyer and Moore in 1980
- ▶ proved in Fortran
- ▶ proved by Jean-Christophe Filliâtre in Why3 in 2013



Two steps process

- ▶ First loop selects a candidate for the majority (pairing)
- ▶ Second loop checks whether the candidate has majority (counting)

Pairing phase

- ▶ if k is 0, select current cell value as candidate, count is 1
- ▶ otherwise, if current cell matches current candidate increase k
- ▶ otherwise decrease k

Proof Sketch

- ▶ a group of k cells contain current candidate
- ▶ another group in which cells can be paired with different values (because we decrease k)
- ▶ Thus, for any other value v , at most $(i-k)/2$ cells contain v

▶ next



mjrtty: logic functions

```

/*@ axiomatic FCount {
  logic integer count{L}(unsigned* a,
    integer length, unsigned elt)
    reads a[0 .. length - 1];
  axiom count_0:
    \forallall unsigned*a, elt, integer length;
      length <=0 ==> count(a,length,elt)==0;
  axiom count_neq:
    \forallall unsigned*a, elt, integer length;
      length>=0 && a[length] != elt ==>
        count(a,length,elt)==
          count(a,length+1,elt);
  axiom count_eq: ...
}
*/

```



mjrtty: general contract

```

requires length > 0;
requires \valid(a+(0 .. length - 1));
requires wf_result(a,length);
assigns \nothing;
behavior has_majority:
assumes \exists unsigned cand;
    majority(a,length,cand);
ensures majority(a,length,\result);
behavior no_majority:
assumes \forall unsigned cand;
    !majority(a,length,cand);
ensures \result == 0;
complete behaviors;
disjoint behaviors;

```



mjrtty: loop annotations

```

loop invariant 1 <= i <= length;
loop invariant cnt <= count(a,i,cand);
loop invariant \forall c unsigned c;
    c != cand ==> 2 * count(a,i,c) <= i - cnt;
loop invariant
    2 * (count(a,i,cand) - cnt) <= i - cnt;
loop assigns i, cnt, cand;
loop variant length - i;
    
```



tmp2[j][i] = -1 << (NBI - 1); else if (tmp1[j][i]) >= (1 << (NBI - 1)) tmp2[j][i] = (1 << (NBI - 1)) - 1; else tmp2[j][i] = tmp1[j][i]; /* Then the second pass. Looks like the first one. */
 tmp1[j][i] = 0; k < k + 1; tmp1[j][i] += mc2[j][k] * tmp2[k][i]; /* The [j,i] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1 = MC1*M1) = MC2*tm1*tmcl, 2



ACSL and WP are powerful tools for specifying and proving functional properties of C programs.

To go further

- ▶ Use several automated provers via Why3.
- ▶ Interactive proof assistant (Coq).
- ▶ Other kinds of specifications (Aoraï)
- ▶ Other uses of ACSL (EACSL and StaDy)

```
(long n)
{ for (i = 0; i < n; i++)
  C[i] = 0;
  tmp2 = ...
  // ...
}
```

tmp2[i] = (i < n) ? tmp[i] : 0; else if (tmp[i] >= 0) tmp2[i] = (i < n) ? tmp[i] : 0; else tmp2[i] = tmp[i]; /* Then the second part looks like the first one: "if (i < n) tmp2[i] = 0; k = 0; k++" tmp2[i] = mc2[i][k] * tmp2[k][i] /* The [i][k] coefficient of the matrix product MC2*TMP2, that is: *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 *MC1

