

Frama-C WP Tutorial

Virgile Prevosto

October 13th, 2014

(long m
(for i < 0
C1); if (b
tmp2 =
at the
tmp2[i][0] = -1, i < (NBi - 1), else if (tmp1[i][0] >= 1, i < (NBi - 1)) tmp2[i][0] = 0, i < (NBi - 1), else tmp2[i][0] = tmp1[i][0] > Than have second pass (code like this first one)
tmp1[i][j][0], 0; k < NBi - i) tmp1[i][0] = mC2[i][j][k]; tmp2[i][j][0]; The [i][j] coefficient of the matrix product MC2^T * MP2, that is, $(MC2^T)^T \cdot MP2 = MC2^T \cdot (MC1^T)^T \cdot MP2 = MC2^T \cdot MC1^T \cdot MP2$
i++ 1; tmp1[i][0] >= 1; /* Final rounding, tmp2[i][0] is now represented on 9 bits. */ if (tmp1[i][0] > 255) m2[i][0] = 255; else if (tmp1[i][0] < -255) m2[i][0] = -255; else m2[i][0] = tmp1[i][0];



Main objective:

Rigorous, mathematical proof of semantic properties of a program

- ▶ functional properties
- ▶ safety:
 - ▶ all memory accesses are valid,
 - ▶ no arithmetic overflow,
 - ▶ no division by zero, ...
- ▶ termination
- ▶ ...

In this tutorial, we will see

- ▶ how to specify a C program with ACSL
- ▶ how to prove it automatically with Frama-C/WP
- ▶ how to understand and fix proof failures

```
(long m[8][8];  
for (i = 0; i < 8; i++) {  
    for (j = 0; j < 8; j++) {  
        m[i][j] = 0;  
    }  
}
```

```
if (m[0][0] == 0) {  
    m[0][0] = 1;  
    for (i = 1; i < 8; i++) {  
        m[0][i] = -1;  
    }  
}  
else if (m[0][0] > 0) {  
    for (i = 0; i < 8; i++) {  
        m[0][i] = 1;  
    }  
}  
else if (m[0][0] < 0) {  
    for (i = 0; i < 8; i++) {  
        m[0][i] = -1;  
    }  
}
```

```
tmp2[0][0] = -1; i = 1; k = 0;  
while (i < 8) {  
    if (tmp2[0][0] == 0) {  
        m[0][0] = 1; m[0][1] = -1; m[0][2] = 1; m[0][3] = -1; m[0][4] = 1; m[0][5] = -1; m[0][6] = 1; m[0][7] = -1;  
        for (j = 1; j < 8; j++) {  
            tmp2[0][j] = -1;  
        }  
    }  
    else if (tmp2[0][0] > 0) {  
        m[0][0] = 1; m[0][1] = 1; m[0][2] = 1; m[0][3] = 1; m[0][4] = 1; m[0][5] = 1; m[0][6] = 1; m[0][7] = 1;  
        for (j = 1; j < 8; j++) {  
            tmp2[0][j] = 1;  
        }  
    }  
    else if (tmp2[0][0] < 0) {  
        m[0][0] = -1; m[0][1] = 1; m[0][2] = -1; m[0][3] = 1; m[0][4] = -1; m[0][5] = 1; m[0][6] = -1; m[0][7] = 1;  
        for (j = 1; j < 8; j++) {  
            tmp2[0][j] = -1;  
        }  
    }  
    i++;  
}
```



Presentation of Frama-C Context

Basic function contract

A little bit of background
ACSL and WP

Loops

Background
Loop termination

Advanced contracts

Behaviors
User-defined predicates

Conclusion

(long m[
for i < 0
C1); if (b
tmp2 =
at the
tmp2[i][0] = -1 <= (NBi - 1); else if (tmp1[i][0] >= (1 - (NBi - 1)) tmp2[i][0] = (1 - (NBi - 1)) else tmp2[i][0] = tmp1[i][0]; tmp1[i][0] = tmp2[i][0]; The [i][j] coefficient of the matrix product MC2^T * MP2, that is, $MC2^T \cdot (MC1 \cdot M1) = MC2^T \cdot MC1 \cdot M1 = MC2^T \cdot M1$ (MC1 is the identity matrix). The value of the [i][j] component of the matrix product is now represented on 9 bits. If (tmp1[i][0] > 255) m2[i][0] = 255; else if (tmp1[i][0] < -255) m2[i][0] = -255; else m2[i][0] = tmp1[i][0].



Presentation of Frama-C Context

```
(long m)
    for (i = 0
        ; i < m
        ; i++)
        C1;
    if (ft
        > 0)
        temp2 =
```

tmp2[0][0] = -(1 << (NB-1)) ; else if (tmp1[0][0]) >= (1 << (NB-1)) tmp2[0][0] = 1 << (NB-1) - 1; else tmp2[0][0] = tmp1[0][0]; // Then the second pass. Looks like the first one. void

Frama-C at a glance

- ▶ A framework for modular analysis of C code.
- ▶ <http://frama-c.com/>
- ▶ Developed at CEA LIST and INRIA Saclay (Proval, now Toccata team).
- ▶ Released under LGPL license (Neon in March 2014)
- ▶ Kernel based on CIL (Necula et al. – Berkeley).
- ▶ ACSL annotation language.
- ▶ Extensible platform
 - ▶ Collaboration of analysis over same code
 - ▶ Inter plug-in communication through ACSL formulas.
 - ▶ Adding specialized plug-in is easy

```
(long m[10][10];  
for (i = 0; i < 10; i++) {  
    for (j = 0; j < 10; j++) {  
        if (i == j) m[i][j] = 1;  
        else if ((i + j) % 2 == 0) m[i][j] = 1;  
        else m[i][j] = 0;  
    }  
}
```

```
m[0][0] = -1; /* i < -(NBi - 1); */ else if ((tmp1[0][0] >= 1) < -(NBi - 1)) tmp2[0][0] = (1 < -(NBi - 1)) ? 1 : 0; else if ((tmp1[0][0] >= 1) < -(NBi - 1)) tmp2[0][0] = 1; else if ((tmp1[0][0] >= 1) < -(NBi - 1)) tmp2[0][0] = 0; k = 0; k < 10; k++) m[0][k] = tmp2[0][k]; /* The [0][j] coefficient of the matrix product MC2^n * MP2, that is, n * MC2^n * (MP2)^n = MC2^n * MC1^n * M = MC2^n * MC1^n * m[0][0]. */  
for (i = 1; i < 10; i++) {  
    for (j = 0; j < 10; j++) {  
        if (i == j) m[i][j] = 1;  
        else if ((i + j) % 2 == 0) m[i][j] = 1;  
        else m[i][j] = 0;  
    }  
}
```

ACSL: ANSI/ISO C Specification Language

Presentation

- ▶ Based on the notion of contract, like in Eiffel
- ▶ Allows users to specify functional properties of their code
- ▶ Allows communication between various plugins
- ▶ Independent from a particular analysis
- ▶ ACSL manual at <http://frama-c.com/acsl>

Basic Components

- ▶ First-order logic
- ▶ Pure C expressions
- ▶ C types + \mathbb{Z} (integer) and \mathbb{R} (real)
- ▶ Built-ins predicates and logic functions, particularly over pointers:
`\valid(p)` `\valid(p+0..2)`,
`\separated(p+0..2,q+0..5)`, `\block_length(p)`



Main plug-ins



External plugins

- ▶ Taster (coding rules, Atos/Airbus, Delmas &al., ERTS 2010)
- ▶ Dassault's internal plug-ins (Pariente & Ledinot, FoVeOOs 2010)
- ▶ Fan-C (flow dependencies, Atos/Airbus, Duprat &al., ERTS 2012)
- ▶ Simple Concurrency plug-in (Adelard, first release in 2013)
- ▶ Various academic experiments (mostly security and/or concurrency related)

```
(long m)
  (for i < 0
    C1); if (b
      tmp2 =
```

```
at the
```

```
tmp2[0][0] = -1; i < (NbI - 1); else if (tmp1[0][0] >= 1 < (NbI - 1)) tmp2[0][0] = 0; i < (NbI - 1); else if (tmp2[0][0] >= 1 < (NbI - 1)) tmp2[0][0] = 0; k < (k + i) tmp1[0][0] = tmp2[0][0]; i < NbI; j < k; The [j][j] coefficient of the matrix product MC2^T / MP2, that is, "MC2^T / TMP2" = MC2^T / (MC1 \ MC1^T) = MC2^T / MC1^T. MC1^T =
```

```
i == 1; tmp1[0][0] >= 1; /* Final rounding, tmp2[0][0] is now represented on 9 bits. */ if (tmp1[0][0] > 255) m2[0][0] = 255; else if (tmp1[0][0] < -255) m2[0][0] = -255; else m2[0][0] = tmp1[0][0];
```

Presentation of Frama-C

Context

Basic function contract

A little bit of background
ACSL and WP

Loops

Background
Loop termination

Advanced contracts

Behaviors
User-defined predicates

Conclusion

(long m[
 for i < 0
 C1;
 if (b
 tmp2 =
 at the
 tmp2[i][0] = -1, i < (NBi - 1); else if (tmp1[i][0] >= (1 - i) * (NBi - 1)) tmp2[i][0] = (1 - i) * (NBi - 1) - 1, else tmp2[i][0] = -tmp1[i][0]; // Then have good pass conditions for the first row of the matrix product
 tmp2[i][j] = 0; k < NBi - i; tmp1[k][0] = mc2[i][jk]; tmp2[i][j] = tmp2[i][j] + mc2[i][jk]; // The [i][j] coefficient of the matrix product MC2^T * MP2, that is, $MC2^T \cap MP2 = MC2^T \cap (MC1 \cap M1) = MC2^T \cap M1 \cap MC1$
 i += 1; tmp1[i][0] >= 1; // Final rounding, tmp2[i][0] is now represented on 9 bits.
 if (tmp1[i][0] > 255) m2[i][0] = -255; else if (tmp1[i][0] > 255) m2[i][0] = 255; else m2[i][0] = tmp1[i][0];
 at the
end of the loop
)



Hoare Logic

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

S_1;

S_2;

}

(long m[
for i < 0;
C1); if (b
tmp2 =

tmp2[i][0] = -1, i < (NB1 - 1), else if (tmp1[i][0] >= 1, i < (NB1 - 1)) tmp2[i][0] = 0, i < (NB1 - 1), else tmp2[i][0] = tmp1[i][0] - 1; then have second pass (code like this first): for (i = 0; i < NB1; i++) {
tmp1[i][0] = 0; k < 6, k > i, tmp1[i][0] <= mC2[i][k]; } tmp2[i][0] = 0; } The [i][j] coefficient of the matrix product MC2^T * MP2, that is, "MC2^T(MC1^W1) = MC2^T(MC1^W1) * MC1^W1". The value of the [i][j] coefficient is now represented on 9 bits. If (tmp1[i][0] < -255, m2[i][0] = -255, else if (tmp1[i][0] > 255, m2[i][0] = 255, else m2[i][0] = tmp1[i][0]);

► Hoare Triples:

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

► Weakest Preconditions:

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

► Proof Obligation (PO):

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



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

```
S_1;
```

```
S_2;
```

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

```
(long m[  
for i < 0  
C1); if (b  
tmp2 =
```

► Hoare Triples:

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

► Weakest Preconditions:

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

► Proof Obligation (PO):

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



Hoare Logic

```
/*@ requires R;
   ensures E; */
int f(int* x) {
    S_1;
    /*@ assert wp(S_2,E); */
    S_2;
    /*@ assert E; */
}
```

```
(long m)
  (for i < 0
  C1); if (b
  tmp2, j
  at the
```

► Hoare Triples:

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

► Weakest Preconditions:

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

► Proof Obligation (PO):

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



```
/*@ requires R;
ensures E; */
int f(int* x) {
/*@ assert
    wp(S_1,wp(S_2,E)); */
S_1;
/*@ assert wp(S_2,E); */
S_2;

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

```
(long m)
  (for i < 0
  C1); if (b
  tmp2, i
  at the
```

► Hoare Triples:

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

► Weakest Preconditions:

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

► Proof Obligation (PO):

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



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>

```
(long m1[100][100];  
for (i = 0; i < 100; i++) {  
    for (j = 0; j < 100; j++) {  
        m1[i][j] = 0;  
    }  
}  
C1: if (0 <= i < 100 && 0 <= j < 100)  
    m1[i][j] = 1;
```

```
m1[m2[0][0]] = -1; /* i <= (NB1-1), else if (tmp1[0][0] >= 1 <= (NB1-1)) tmp2[0][0] = (0 <= (NB2-1) <= 1, else if (tmp2[0][0] >= 1 <= (NB2-1)) */ Then frama-c will pass. Code like this fails:  
m1[m2[0][0]] = -1; /* i <= (NB1-1), else if (tmp1[0][0] >= 1 <= (NB1-1)) tmp2[0][0] = (0 <= (NB2-1) <= 1, else if (tmp2[0][0] >= 1 <= (NB2-1)) */ Then frama-c will fail.  
tmp2[0][0] = 0; k = 0; k < 0; k++ ) tmp1[0][0] = m2[0][0]; tmp2[0][0] = m2[0][0]; */ The [i][j] coefficient of the matrix product MC2^T * MP2, that is,  $(MC2^T)^T \cdot MP2 = MC2^T \cdot (MC2 \cdot MP) = MC2^T \cdot MP$ , is now represented on 9 bits. If (tmp1[0][0] >= 255) m2[0][0] = 255, else m2[0][0] = tmp2[0][0].
```

Dealing with pointers

Example

```
// returns the maximum of *p and *q
int max_ptr ( int *p, int *q ) {
    if ( *p >= *q )
        return *p ;
    return *q ;
}
```

Demo

```
(long m1[10][10], long m2[10][10], long m3[10][10])
{ for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
        for (k = 0; k < 10; k++)
            m3[i][j] = m1[i][k] * m2[k][j];
}
```

```
if (tmp2[i][j] < -256) tmp2[i][j] = -256; else if (tmp2[i][j] > 255) tmp2[i][j] = 255; else m2[i][j] = tmp2[i][j]; }
```

Setting values

Example

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

Demo

```
(long m)
  (for 0 < i < N1)
    C1; if (b[i])
      m[m2[i]] = 0;
    else if (tmp1[i] >= -1 && i < (N2-1)) m[m2[i]] = 0 < (N2-1); else if (tmp1[i] >= -1 && i < (N2-1)) m[m2[i]] = -1; else if (tmp1[i] >= 0 && i < (N2-1)) m[m2[i]] = tmp1[i]; else if (tmp1[i] >= 0 && i < (N2-1)) m[m2[i]] = -tmp1[i];
```

```
  m[m2[i]] = 0; if (tmp1[i] >= -255 && m2[i] <= -255) m[m2[i]] = -255; else if (tmp1[i] >= 255 && m2[i] <= 255) m[m2[i]] = 255; else m[m2[i]] = tmp1[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;
}
```



(long m[
(for i < 0 <
C1); if (b
tmp2, i
at the
1) { if (tmp2[i][0] == -1 << (NB1-1)) else if (tmp1[i][0] >= 1 << (NB1-1)) tmp2[i][0] = 0 << (NB1-1)-1 else if (tmp2[i][0] >= 1 << (NB1-1)-2) then have to add one like this first: m2[i][0] = (m2[i][0] << 1) + 1; if (tmp1[i][0] << (NB1-1)-1 == 0) m2[i][0] = 0 << (NB1-1)-1 else if (tmp1[i][0] >= 1 << (NB1-1)-1) m2[i][0] = 1 << (NB1-1)-1; if (tmp1[i][0] >= 1 << (NB1-1)-2) m2[i][0] = 2 << (NB1-1)-2; if (tmp1[i][0] >= 1 << (NB1-1)-3) m2[i][0] = 3 << (NB1-1)-3; if (tmp1[i][0] >= 1 << (NB1-1)-4) m2[i][0] = 4 << (NB1-1)-4; if (tmp1[i][0] >= 1 << (NB1-1)-5) m2[i][0] = 5 << (NB1-1)-5; if (tmp1[i][0] >= 1 << (NB1-1)-6) m2[i][0] = 6 << (NB1-1)-6; if (tmp1[i][0] >= 1 << (NB1-1)-7) m2[i][0] = 7 << (NB1-1)-7; if (tmp1[i][0] >= 1 << (NB1-1)-8) m2[i][0] = 8 << (NB1-1)-8; if (tmp1[i][0] >= 1 << (NB1-1)-9) m2[i][0] = 9 << (NB1-1)-9; if (tmp1[i][0] >= 1 << (NB1-1)-10) m2[i][0] = 10 << (NB1-1)-10; if (tmp1[i][0] >= 1 << (NB1-1)-11) m2[i][0] = 11 << (NB1-1)-11; if (tmp1[i][0] >= 1 << (NB1-1)-12) m2[i][0] = 12 << (NB1-1)-12; if (tmp1[i][0] >= 1 << (NB1-1)-13) m2[i][0] = 13 << (NB1-1)-13; if (tmp1[i][0] >= 1 << (NB1-1)-14) m2[i][0] = 14 << (NB1-1)-14; if (tmp1[i][0] >= 1 << (NB1-1)-15) m2[i][0] = 15 << (NB1-1)-15; if (tmp1[i][0] >= 1 << (NB1-1)-16) m2[i][0] = 16 << (NB1-1)-16; if (tmp1[i][0] >= 1 << (NB1-1)-17) m2[i][0] = 17 << (NB1-1)-17; if (tmp1[i][0] >= 1 << (NB1-1)-18) m2[i][0] = 18 << (NB1-1)-18; if (tmp1[i][0] >= 1 << (NB1-1)-19) m2[i][0] = 19 << (NB1-1)-19; if (tmp1[i][0] >= 1 << (NB1-1)-20) m2[i][0] = 20 << (NB1-1)-20; if (tmp1[i][0] >= 1 << (NB1-1)-21) m2[i][0] = 21 << (NB1-1)-21; if (tmp1[i][0] >= 1 << (NB1-1)-22) m2[i][0] = 22 << (NB1-1)-22; if (tmp1[i][0] >= 1 << (NB1-1)-23) m2[i][0] = 23 << (NB1-1)-23; if (tmp1[i][0] >= 1 << (NB1-1)-24) m2[i][0] = 24 << (NB1-1)-24; if (tmp1[i][0] >= 1 << (NB1-1)-25) m2[i][0] = 25 << (NB1-1)-25; else if (tmp1[i][0] > 255 << (NB1-1)) m2[i][0] = 255 << (NB1-1); if (tmp1[i][0] < -256 << (NB1-1)) m2[i][0] = -256 << (NB1-1); else if (tmp1[i][0] > 255 << (NB1-1)) m2[i][0] = -255 << (NB1-1); else m2[i][0] = tmp1[i][0]; } }

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:

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

► Must specify effects (Frame rule)

$\forall x \in State \setminus A, g$ 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;
}

(long m)
{for (j = 0; j < n; j++)
C1; if (the
tmp2 :=
```

- ▶ 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)

$\forall x \in State \setminus A, g$ 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)

$\forall x \in State \setminus A, g$ 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;
}

(long m)
(for i < n)
(C1); if (b)
    tmp2 := ...;
at the end of the loop:
tmp2 := ...;
```

- ▶ 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)

$\forall x \in State \setminus A, g$ 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

```
(long m  
(for i < 0  
C1); if (b  
tmp2 =  
at the  
tmp2)) = -1 <= (NBi - 1); else if (tmp1[i][0] >= (1 <= (NBi - 1)) tmp2[i][0] = (1 <= (NBi - 1)) else if (tmp2[i][0] >= (1 <= (NBi - 1)) tmp2[i][0] = 0 <= (NBi - 1)) then have good pass (code like this first)  
tmp2[i][0] = 0; k = 0; k < i; k < i) tmp1[i][k] = mC2[i][k]; tmp2[i][k] = mC2[i][k]; // The [i][j] coefficient of the matrix product MC2^T * MP2, that is,  $(MC2^T)^T \cdot MP2 = MC2^T \cdot (MC2 \cdot MC1^T) = MC2^T \cdot MC1^T \cdot MC1 = MC2^T \cdot MC1^T$   
i += 1; tmp1[i][0] >= 1; // Final rounding, tmp2[i][0] is now represented on 9 bits.  
if (tmp1[i][0] < -255) m2[i][0] = -255; else if (tmp1[i][0] > 255) m2[i][0] = 255; else m2[i][0] = tmp1[i][0];  
}
```



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);
}
```

(long m
(for i < 0
C1); if (b
tmp2, i
at the

tmp2[0][0] = -1, i < (NBi - 1); else if (tmp1[0][0] >= 1, i < (NBi - 1)) tmp2[0][0] = 0, i < (NBi - 1), else if (tmp1[0][0] >= 1, i < (NBi - 1)) tmp2[0][0] = 0, i < (NBi - 1); tmp2[1][0] = 0, i < (NBi - 1); tmp2[2][0] = 0, i < (NBi - 1); The [i][j] coefficient of the matrix product MC2^TTMPC, that is, $(MC2^T)^T(MC1^W) = MC2^T(MC1^W) = MC2^T(MC1^W)$, where $W = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$. Final rounding: tmp2[0][0] is now represented on 9 bits. If (tmp1[0][0] >= 255, m2[0][0] >= 255, else if (tmp1[0][0] > 255, m2[0][0] > 255, else m2[0][0] = tmp1[0][0], m2[1][0] = tmp1[1][0], m2[2][0] = tmp1[2][0].

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 permute(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 permute(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 permute

```
/*@ 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 permute(int* a, int *b, int* c) {
    swap(a,b);
    swap(a,c);
}
```

(long m
for i < 0
C1); if (b
tmp2, i
at the
tmp2[i][0] = -1, i < (NBi - 1); else if (tmp1[i][0] >= 1, i < (NBi - 1)) tmp2[i][0] = 0, i < (NBi - 1), else if (tmp2[i][0] >= 1, i < (NBi - 1)) tmp2[i][0] = 1, i < (NBi - 1), else if (tmp2[i][0] >= 0, i < (NBi - 1)) tmp2[i][0] = 0, i < (NBi - 1); The [i][j] coefficient of the matrix product MC2^TTMPC, that is, $(MC2^T)^T M P C = M C 2^T (M C_1^{-1} M_1) = M C 2^T M_1^{-1} M C_1$, where $M C_1^{-1} M_1 = M C_2^T$. Final rounding: tmp2[0][0] is now represented on 9 bits. If (tmp1[0][0] > -255 & m2[0][0] < -255) else if (tmp1[0][0] > 255 & m2[0][0] < 255) else m2[0][0] = tmp1[0][0].



Contract of permute

```
/*@ 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 permute(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



Presentation of Frama-C Context

Basic function contract

A little bit of background
ACSL and WP

Loops

Background
Loop termination

Advanced contracts

Behaviors
User-defined predicates

Conclusion

(long m[
(for i < 0
C1); if (b[i]
tmp2 =
at the
tmp2[i][0] = -1, i < (NBi - 1), else if (tmp1[i][0] >= (1 - (NBi - 1)) tmp2[i][0] = (1 - (NBi - 1)) else tmp2[i][0] = tmp1[i][0] - 1); then b[i], cond pass (only like this first); then
tmp1[i][j][0] >= 0, k < i) tmp1[i][0] = mcp2[i][j][k]; tmp2[i][j][0] = The [i][j] coefficient of the matrix product MC2^T * MP2, that is, $MC2^T \cap MP2 = MC2^T \cap (MC1 \cap M1) = MC2^T \cap M1 \cap MC1$; if (tmp1[i][0] >= 1) tmp1[i][0] >= 1; /* Final rounding, tmp2[i][0] is now represented on 9 bits. */ if (tmp1[i][0] > -255 m2[i][0] = -255, else if (tmp1[i][0] > 255) m2[i][0] = 255, else m2[i][0] = tmp1[i][0] + 128);



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

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

(long m
(for i < 0
C1); if (b
tmp2 =

- ▶ 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;
}
```

(long m
(for i < 0
C1); if (B
tmp2 =

- ▶ 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

tmp2[i][0] = -1 <= (NBi - 1); else if (tmp1[i][0] >= 1 <= (NBi - 1)) tmp2[i][0] = 0 <= (NBi - 1); else tmp2[i][0] = tmp1[i][0] / tmp2[i][0]; tmp2[i][1] = tmp2[i][0] * k; if (tmp1[i][0] >= 1) tmp2[i][1] = 0; k <= 0; k >= 1; tmp1[i][0] >= 1; tmp2[i][1] >= 1; The [i][j] coefficient of the matrix product MC2^T * MP2, that is, $(MC2^T * MP2)_{ij} = MC2_{ik} * (MC1^T)_{kj} = MC2_{ik} * (MC1^T)_{kj}$. Final rounding: tmp2[i][1] is now represented on 9 bits. if (tmp1[i][0] > 255) m2[i][0] = -255; else if (tmp1[i][0] < -255) m2[i][0] = 255; else m2[i][0] = tmp2[i][1].



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

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

(long m
(for i < 0
C1); if (B
tmp2 =

- ▶ 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;
}
```

(long m
(for i < 0
C1); if (B
tmp2 =

- ▶ 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;
}
```

(long m
(for i < 0
C1); if (B
tmp2 =

- ▶ 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;
}
```

(long m[
(for i < 0;
C1); if (B1
tmp2 =

- ▶ 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

```
(long m)
{ for (i = 0
C1); if (b
tmp2 =
```

$m2[0][0] = -1 \leq -(N\!B\!-1)$, else if ($m2[0][0] > -(1 \leq -(N\!B\!-1))$) $m2[0][0] = -(1 \leq -(N\!B\!-1))$, else $m2[0][0] > -(1 \leq -(N\!B\!-1))$ than have second pass. Code like this first:

```
tmp2[0][0] = 0; k = 0; k < n tmp1[0][0] >= m2[0][0]; tmp2[0][0] = tmp1[0][0];
```

The [0][0] coefficient of the matrix product $MC_2^T \cdot MP_2$, that is, $^T MC_2 \cdot (^T MP_2) = MC_2^T \cdot (MC_1 \cdot M_1) = MC_2^T \cdot M_1 \cdot MC_1$.

$m2[0][0] = -1 \leq -(N\!B\!-1)$, else if ($m2[0][0] > -(1 \leq -(N\!B\!-1))$) $m2[0][0] = -(1 \leq -(N\!B\!-1))$, else $m2[0][0] > -(1 \leq -(N\!B\!-1))$ than have second pass. Code like this first:

```
tmp2[0][0] = 0; k = 0; k < n tmp1[0][0] >= m2[0][0]; tmp2[0][0] = tmp1[0][0];
```

Final rounding, $tmp2[0][0]$ is now represented on 9 bits. If $(tmp2[0][0] > 255) m2[0][0] = 255$, else if $(tmp2[0][0] > 255) m2[0][0] = -256$, else if $(tmp2[0][0] > 255) m2[0][0] = 255$, else $m2[0][0] = tmp2[0][0]$.

Max array contract

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

(long m
(for i < 0
C1); if (b
tmp2, /

tmp2)[0][0] = -1, i < (NBi - 1); else if (tmp1[0][0] >= (1 - i) * (NBi - 1)) tmp2[0][0] = (1 - i) * (NBi - 1) - 1, else tmp2[0][0] = -tmp1[0][0]; Then the second pass (code like the first one) is done.
tmp2[i][j] = 0; k < NBi - i; tmp1[i][k] >= tmp2[i][j]; tmp2[i][j] = 1; The [i][j] coefficient of the matrix product MC2^TTM P2, that is, $^T(MC2)(MC1)(M1) = MC2^T(MC1)(M1) = MC2^T(M1)(MC1)$.
i += 1; tmp1[0][0] >= 1; /* Final rounding, tmp2[0][0] is now represented on 9 bits. */ if (tmp1[0][0] > 255) m2[0][0] = 255; else if (tmp1[0][0] < -255) m2[0][0] = -255; else m2[0][0] = tmp1[0][0];

Max array contract

```

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

```

Impose validity of a whole block of memory

```

(long m)
  for i = 0 to N-1 do
    C1; if (b[i] < 0) then
      tmp2 := -tmp1[i];
    else if (tmp1[i] < -(N-1)) then tmp2[i] := 0 <= (N-1) else tmp2[i] := 0 <= (N-1)-1 else tmp2[i] := 0 <= (N-1)-2; then have cond pass. Code like this first:
    long
      tmp2[i][j][k] = 0; k < 0; k++(tmp1[i][j]) * tmp2[i][j][k]; // The [j,k] coefficient of the matrix product MC2^T / MP2, that is, (MC2^T)(MP2) = MC2^T(MC1^W) = MC2^T(MC1^W)(MC1^W)^T = MC2^T(MC1^W)^T. Final rounding: tmp2[i][j][k] is now represented on 9 bits. //if (tmp1[i][j] < -255) m2[i][j] = -255; else if ((tmp1[i][j] > 255) m2[i][j] = 255; else m2[i][j] = tmp1[i][j];

```

Max array contract

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

we want all i in the interval to verify the inequality

```
(long m1[100][100];  
for (i = 0; i < 100; i++)  
  for (j = 0; j < 100; j++)  
    m1[i][j] = 0;  
int main()  
{  
  long m2[100][100];  
  for (i = 0; i < 100; i++)  
    for (j = 0; j < 100; j++)  
      m2[i][j] = 0;  
  for (i = 0; i < 100; i++)  
    for (j = 0; j < 100; j++)  
      for (k = 0; k < 100; k++)  
        m1[i][j] += m1[i][k] * m2[k][j];  
  for (i = 0; i < 100; i++)  
    for (j = 0; j < 100; j++)  
      cout << m1[i][j] << endl;  
}
```



Max array contract

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

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

(long m
for i < 0
C1); if (b
tmp2 =

tmp2[0][0] = -1 <= (NB-1); else if (tmp1[0][0] >= 1 <= (NB-1)) tmp2[0][0] = 0 <= (NB-1) <= 1 else tmp2[0][0] = tmp1[0][0]; // Then have second pass code like this first one. Then
tmp1[0][0], 0 <= k <= n) tmp1[0][k] = tmp2[0][k]; // The [0][j] coefficient of the matrix product MC2^T / MP2, that is, $(MC2^T)^T(MP2) = MC2^T(MC1^T) = MC2^T(MC1^T)^T(MC2)$
if (tmp1[0][0] >= 1, // Final rounding, tmp2[0][0] is now represented on 9 bits. // If (tmp1[0][0] > 255) m2[0][0] = 255; else if ((tmp1[0][0] > 255) m2[0][0] = 255; else m2[0][0] = tmp1[0][0];

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  
}  
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;

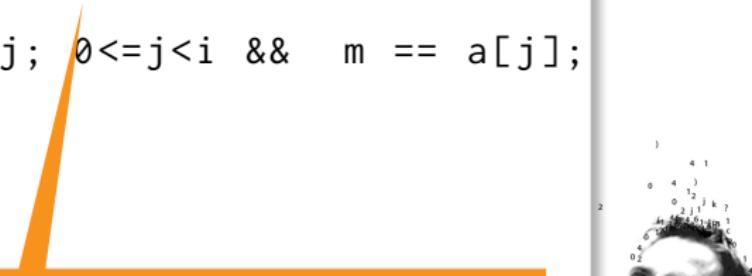
*/

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

}

return m;

(long m)
(for i = 0 to length - 1)
(C1); if (the condition)
(tmp2 := tmp1[i]); if (the condition)
(tmp2 := tmp2[i]); if (the condition)



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

- ▶ Program termination is undecidable
- ▶ A tool cannot deduce neither the exact number of iterations, nor even an upper bound
- ▶ If an upper bound is given, a tool can **check it by induction**
- ▶ An upper bound on the number of remaining loop iterations is the key idea behind the **loop variant**

Terminology

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

```
(long m)
  for i = 0 to N-1
    C[i] := 0
    for j = 0 to N-1
      tmp2[i][j] = -1 <= (NBB-1) * i + j <= (NBB-1) * (tmp2[0][0] - 1 <= (NBB-1) * (-1) else tmp2[1][0] - tmp2[1][0] > -1) then have second pass code like this first pass
      tmp2[i][j] = 0; k = 0; k < N; tmp2[i][j] := tmp2[i][j] * tmp2[k][j]; k = k + 1; if (tmp2[i][j] > 255 or tmp2[i][j] < -256) then
      i++ else if (tmp2[i][j] > 255) then m2[i][j] = 255 else m2[i][j] = tmp2[i][j]
```



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

```
(long m)
{ for (i = 0; i < N; i++)
  C1; if (b
  tmp2 =
```

```
tmp2[i][0] = -1 <= (NBi - 1); else if (tmp1[i][0] >= (1 - (NBi - 1)) tmp2[i][0] = (1 - (NBi - 1)) else tmp2[i][0] = tmp1[i][0]; tmp2[i][1] = tmp2[i][0] * m2[i][1]; } The [i][j] coefficient of the matrix product MC2^T * MP2, that is,  $MC2^T \cap MP2 = MC2^T \cap (MC1 \cap M1) = MC2^T \cap M1$  (MC1 \cap M1) = MC2^T \cap M1
```

```
i++= 1; tmp1[i][0] >= 1; } Final rounding, tmp2[i][0] is now represented on 9 bits. If (tmp1[i][0] > 255 m2[i][0] = -255; else if (tmp1[i][0] > 255 m2[i][0] = 255; else m2[i][0] = tmp1[i][0]; tmp2[i][1] = tmp2[i][0] * m2[i][1]; }
```

loop variant

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

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

(long m;
(for i < 0;
C1); if (b
tmp2, i
at the
tmp2)) = -1, i < (NB1 - 1); else if (tmp1[i][0] >= (1 < (NB1 - 1)) tmp2[i][0] = (1 < (NB1 - 1)) else tmp2[i][0] = tmp1[i][0]; if (tmp1[i][0] >= (1 < (NB1 - 1)) tmp2[i][0] = 0; k < (k < i) tmp1[i][0] >= mc2[i][j][k]; tmp2[i][0] = 0; The [i][j] coefficient of the matrix product MC2^T * MP2, that is, "MC2^T * TMP1" = MC2^T * (MC1 * M1) = MC2^T * MP2. MC1
tmp1[i][0] >= 1; tmp1[i][0] >= 1; /* Final rounding, tmp2[i][0] is now represented on 9 bits. */ if (tmp1[i][0] >= 255) m2[i][0] = -255; else if (tmp1[i][0] > 255) m2[i][0] = 255; else m2[i][0] = tmp1[i][0];



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



Presentation of Frama-C Context

Basic function contract

A little bit of background
ACSL and WP

Loops

Background
Loop termination

Advanced contracts

Behaviors
User-defined predicates

Conclusion

```
(long m)
  (for i = 0
    C1); if (b)
      tmp2 = i;
```

```
tmp2[j][0] = -1, i < (NBi - 1); else if (tmp1[i][0] >= (1 - i) * (NBi - 1)) tmp2[i][0] = (1 - i) * (NBi - 1) - 1, else tmp2[i][0] = -tmp1[i][0]; // Then have good pass conditions for the first row of the matrix product. The [i][j] coefficient of the matrix product MC2^t * MP2, that is, "MC2^t * (MC1^t * M1)" = MC2^t * (MC1^t * M1) = MC2^t * M1 * MC1^t * M1 = MC2^t * M1 = MC2^t.
```



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

directly

```
predicate is_sorted(int* a, l) =  
  \forall i; 0 <= i < l - 1 ==> a[i] <= a[i + 1];
```

with axioms

```
axiomatic Sorted {  
  predicate is_sorted{L}(int* a, l);  
  axiom def: \forall int*a, l,i; ...}
```

inductively

```
inductive is_sorted{L}(int* a, l) {  
  case is_sorted_nil: \forall int* a,  
    is_sorted(a, 0);  
  case is_sorted_cons: ... }
```



Example

```
/* returns index of a cell containing key,
   returns -1 iff key is not present
   in the array */
int binary_search(int* a, int length, int key) {
    int low = 0, high = length - 1;
    while (low<=high) {
        int mid = (low+high)/2;
        if (a[mid] == key) return mid;
        if (a[mid] < key) { low = mid+1; }
        else { high = mid - 1; }
    }
    return -1;
}
```

Demo

(long m
(for i < d
C1); if
tmp2[i]
at the

tmp2[0][0] = -1 <= (NBi - 1) % else if (tmp1[0][0] >= 1 <= (NBi - 1)) tmp2[0][0] = 0 <= (NBi - 1) - 1 else tmp2[0][0] = tmp1[0][0] >= Than have second pass like this first one
tmp2[1][0] = 0 <= (NBi - 1) % else if (tmp1[1][0] >= 1 <= (NBi - 1)) tmp2[1][0] = 0 <= (NBi - 1) - 1 else tmp2[1][0] = tmp1[1][0] >= The [i][j] coefficient of the matrix product MC2^T * MP2, that is, $(MC2^T)^T \cdot MP2 = MC2^T \cdot (MC1^T \cdot MP1) = MC2^T \cdot MC1^T \cdot MP1 = MC2^T \cdot MP1$.
tmp2[1][1] = 0 <= (NBi - 1) % else if (tmp1[1][1] >= 1 <= (NBi - 1)) tmp2[1][1] = 0 <= (NBi - 1) - 1 else tmp2[1][1] = tmp1[1][1] >= Final rounding, tmp2[0][0] is now represented on 9 bits, 7 if (tmp1[0][0] >= -256 else if (tmp1[0][0] > 255) m2[0][0] = 255, else m2[0][0] = tmp1[0][0]

Binary search: general contract

```
/*@  
  requires \valid(a+(0..length-1));  
  requires is_sorted(a,length);  
  requires length >=0;  
  
  assigns \nothing;  
  
 */  
int binary_search(int* a, int length, int key) {
```

(long m
(for i < 0
C1); if (b
tmp2 =

if (tmp2[0][0] == -1 && i < (NBi - 1)) else if (tmp1[0][0] >= (1 < (NBi - 1))) tmp2[0][0] = (1 < (NBi - 1)) else if (tmp2[0][0] >= (tmp1[0][0] >= (1 < (NBi - 1)))) tmp2[0][0] = tmp1[0][0]; // Then have good pass. Code like this first. if (tmp1[0][0] >= 0 && i < (NBi - 1)) tmp2[0][0] = 0; k = (key - i) * tmp2[0][0]; if (tmp2[0][0] >= 1) tmp2[0][0] = 1; // The [i][j] coefficient of the matrix product MC2^T * MP2, that is, $(MC2^T)^T \cdot MP2 = MC2^T \cdot (MC1^T \cdot MP1) = MC2^T \cdot MC1^T \cdot MP1$. if (tmp1[0][0] >= -255 & m2[0][0] < -255) if (tmp1[0][0] > 255) m2[0][0] = 255; else if (tmp1[0][0] < -255) m2[0][0] = -255; else m2[0][0] = tmp1[0][0];



Binary search: general contract

```
/*@  
    requires \valid(a+(0..length-1));  
    requires is_sorted(a,length);  
    requires length >=0;  
  
    assigns \nothing;  
  
*/  
int binary_search(in * a, int length, int key) {
```

we use our predicate



Binary search: behavior 1

```
/*@  
behavior exists:  
assumes  
    \exists integer i;  
        0 <= i < length && a[i] == key;  
ensures  
    0 <= \result < length && a[\result] == key;  
*/  
int binary_search(int* a, int length, int key) {
```



Binary search: behavior 1

```
/*@  
  
behavior exists:  
    assumes  
        \exists integer i;  
        0 <= i < length && a[i] == key;  
    ensures  
        0 <= \result < length && a[\result] == key;  
*/  
int binary_search(int* a, int length, int key) {
```

We are in this behavior when key is present in the array



Binary search: behavior 1

```
/*@  
  
behavior exists:  
    assumes  
        \exists integer i;  
        0 <= i < length && a[i] == key;  
    ensures  
        0 <= \result < length && a[\result] == key;  
*/  
int binary_search(int* a, int length, int key) {
```

If we are in exists, we must return an appropriate index



Binary search: behavior 2

```

/*@ 
  behavior not_exists:
  assumes
    \forall integer i;
    0 <= i < length ==> a[i] != key;
  ensures \result == -1;

*/
int binary_search(int* a, int length, int key) {

```

```
(long m1[8][8];
for (j = 0; j < 8; j++)
  C1[j][j] = 1;
  for (i = 0; i < 8; i++)
    for (j = 0; j < 8; j++)
      if (i == j)
        m1[i][j] = 1;
      else
        m1[i][j] = 0;
      else if ((i + 1) == (j + 1))
        m1[i][j] = 1;
      else
        m1[i][j] = 0;
  )
}

void f()
{
  long m2[8][8];
  for (j = 0; j < 8; j++)
    for (i = 0; i < 8; i++)
      if (i == j)
        m2[i][j] = 1;
      else
        m2[i][j] = 0;
}

```



Binary search: relations between behaviors

```
/*@  
  
    complete behaviors;  
    disjoint behaviors;  
*/  
int binary_search(int* a, int length, int key) {
```

(long m
(for i < 0
C1); if
tmp2 >

tmp2[0][0] = -1 <= (NBl - 1); else if (tmp1[0][0] >= (1 <= (NBl - 1)) tmp2[0][0] = (1 <= (NBl - 1) <= TmP1[0][0]) & (tmp1[0][0] >= TmP1[0][0]); then have cond pass. Code like this follows: if (tmp1[0][0] <= 0 & k < 0 & k >= n - 1) m2[0][0] = tmp2[0][0]; /* The [0][0] coefficient of the matrix product MC2^T * MP2, that is, "MC2^T(MC1^W1)=MC2^T(MC1^W1)=MC2^T(MC1^W1)=MC2^T(MC1^W1)" */ i++ == 1; /* Final rounding, tmp2[0][0] is now represented on 9 bits. */ if (tmp1[0][0] <= -255 & m2[0][0] <= -255 else if (tmp1[0][0] >= 255 & m2[0][0] <= 255 else m2[0][0] = tmp2[0][0]; /*

Binary search: relations between behaviors

```
/*@  
  
    complete behaviors;  
    disjoint behaviors;  
*/  
int binary_search(int* a, int length, int key) {
```

The two behaviors cover all possible contexts in which
binary_search might be called

```
(long m  
(for i < 0  
C1); if (b  
tmp2, i  
at the  
tmp2[i][0] = -1, i < (NB1 - 1); else if (tmp1[0][0] > -(1 < (NB1 - 1)) tmp2[0][0] = (1 < (NB1 - 1)) else tmp2[0][0] = -1; else tmp2[0][0] = tmp1[0][0] - 1; then have good pass. Code like this first:  
tmp1[0][0]; Q2: k < 0; k < n) tmp1[0][0] <> tmp2[0][0]; tmp2[0][0];
```



Binary search: relations between behaviors

```
/*@  
  
    complete behaviors;  
    disjoint behaviors;  
*/  
int binary_search(int* a, int length, int key) {
```

We can't be in both behaviors at the same time

```
(long m  
(for i < 0  
C1); if  
tmp2 >  
at the
```

```
(tmp2[j][0] = -1 <= j < (Nb-1); else if (tmp1[i][0] >= 1 <= (Nb-1)) tmp2[i][0] = (i < -(Nb-1)) else tmp2[i][0] = tmp1[i][0] > Than have second pass like this first one. Then  
tmp1[i][j]] = 0; k < Nb-k) tmp1[i][k]] >> tmp2[i][k]]; The [i][j] coefficient of the matrix product MC2^T * MP2, that is, MC2^T * TMP2 = MC2^T * (MC1 * M1) = MC2^T * M1 * MC1.  
i+++= 1; tmp1[i][0] >= 1; /* Final rounding. tmp2[i][0] is now represented on 9 bits. */ if (tmp1[i][0] > 255) m2[i][0] = 255; else if (tmp1[i][0] < -255) m2[i][0] = -255; else m2[i][0] = tmp1[i][0];
```

Binary search: loop annotations

```
/*@ loop invariant 0<=low<=high+1;
   loop invariant high<length;
   loop assigns low,high;
   loop invariant
       \forall integer k;
       0<=k<low ==> a[k] < key;
   loop invariant
       \forall integer k;
       high<k<length ==> a[k] > key;
   loop variant high-low;
*/
while (low<=high) {
```

(long m
for i < 0
C1); if (t
tmp2 =

at the

tmp2[0][0] = -1, i < -(NB-1), else if (tmp1[0][0] > -(1 < -(NB-1))) tmp2[0][0] = -(1 < -(NB-1))-1, else tmp2[0][0] = -tmp1[0][0]; then have good passes like this first one: tmp2[0][0] = -1, i < -(NB-1), else if (tmp1[0][0] > -(1 < -(NB-1))) tmp2[0][0] = -(1 < -(NB-1))-1, else tmp2[0][0] = -tmp1[0][0]; 0 < k < i, tmp1[0][0] > tmp2[0][0]; "tmp2[0][0]" is the [0,0] coefficient of the matrix product MC2^T * MP2, that is, "MC2^T * MP2" = MC2^T * (MC1 * M1) = MC2^T * M1 * MC1. Then we have tmp2[0][0] = -1, i < -(NB-1), else if (tmp1[0][0] > -(1 < -(NB-1))) tmp2[0][0] = -(1 < -(NB-1))-1, else tmp2[0][0] = -tmp1[0][0]; 0 < k < i, tmp1[0][0] > tmp2[0][0]; "tmp2[0][0]" is now represented on 9 bits. If (tmp1[0][0] > -255) m2[0][0] = -255, else if (tmp1[0][0] > 255) m2[0][0] = 255, else m2[0][0] = tmp1[0][0].

Exercise

```
struct tree { int data;
              struct tree* left;
              struct tree* right; };

struct tree* search(int key, struct tree* t) {
    struct tree* current = t;
    while (current) {
        if (current->data == key) return current;
        if (current->data < key)
            current = current->left;
        else current = current -> right;
    }
    return current; }
```

Demo

(long m
(for 0 <= i
C); if the
tmp2[i] =
at thetmp2[0][0] = -1 <= (NBi - 1)%, else if (tmp1[0][0] >= 1 <= (NBi - 1)) tmp2[0][0] = 0 <= (NBi - 1) - 1 else tmp2[0][0] = tmp1[0][0] / NBi have second pass code like this first one. Then
tmp2[i][j] = 0; k < 0; k < i) tmp2[i][k] = (mc2[i][k]) * tmp2[k][j]; The [i][j] coefficient of the matrix product MC2^T * MP2, that is, $(MC2^T)^T \cdot MP2 = MC2^T \cdot (MC1 \cdot MC1^T) = MC2^T \cdot MC1^T$.
tmp2[i][j] = 0; k <= 1; tmp1[0][k] >= 1;) Final rounding, tmp2[0][0] is now represented on 9 bits. If (tmp1[0][0] > 255) m2[0][0] = -255; else if ((tmp1[0][0] > 255) m2[0][0] = 255; else m2[0][0] = tmp1[0][0].

Presentation of Frama-C Context

Basic function contract

A little bit of background
ACSL and WP

Loops

Background
Loop termination

Advanced contracts

Behaviors
User-defined predicates

Conclusion

(long m[
for i = 0
C1); if (b[i]
tmp2 = i
at the
tmp2[i][0] = -1, i < (Nb-1), else if (tmp1[i][0] >= (1, i < (Nb-1)) tmp2[i][0] = (1, i < (Nb-1)) else tmp2[i][0] = tmp1[i][0]; then b[i] and pass b[i] like this for i = 0 to Nb-1
tmp2[i][j][0] = 0, k < i, k < i) tmp2[i][j][0] = m2[i][j][k] * tmp2[k][j][0]; The [i][j] coefficient of the matrix product MC2 * TMP2, that is, "MC2[i][TMP2[i][j]] = MC2[i][MC1[i][j]] = MC2[i][m1[i][j]]";
i++ == 1; tmp1[i][0] >= 1;" Final rounding, tmp2[i][0] is now represented on 9 bits. If (tmp1[i][0] > 255) m2[i][0] = 255, else if (tmp1[i][0] > 255) m2[i][0] = tmp1[i][0] - 255, else m2[i][0] = tmp1[i][0]



Summary

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 m
(for i < 0
C1); if
tmp2 =
at the
tmp2[i][0] = -1 <= (NBi - 1); else if (tmp1[i][0] >= 1 <= (NBi - 1)) tmp2[i][0] = 0 <= (NBi - 1) <= tmp1[i][0]; /> Then here second pass (code like the first one)
tmp1[i][0] <= 0 & k < i) tmp1[i][0] => tmp2[i][0]; /> The [i][j] coefficient of the matrix product MC2^T * MP2, that is, $(MC2^T)^T \cdot MP2 = MC2 \cdot (MC1^T \cdot MP1) = MC2 \cdot MC1^T \cdot MP1$

i++ 1; tmp1[i][0] >= 1; /> Final rounding, tmp2[i][0] is now represented on 9 bits. /> if (tmp1[i][0] > 255) m2[i][0] = 255; else if (tmp1[i][0] < -255) m2[i][0] = -255; else m2[i][0] = tmp1[i][0]; />