Frama-C WP

Séminaire CAP'TRONIC

Virgile Prevosto
virgile.prevosto@cea.fr

June 18th, 2015

frama C
Software Analyzers

- Basic function contract
  - A little bit of background
  - ACSL and WP
- Loops
  - Background
  - Loop termination
- Advanced contracts
  - Behaviors
  - User-defined predicates
  - Example
- Conclusion

frama C
Software Analyzers

Hoare Logic

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


S_1;


S_2;


}
```

- Hoare Triples:

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

- Weakest Preconditions:

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

- Proof Obligation (PO):

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

Hoare Logic

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


S_1;


S_2;

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

- Hoare Triples:

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

- Weakest Preconditions:

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

- Proof Obligation (PO):

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

Hoare Logic

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


S_1;

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

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

- Hoare Triples:

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

- Weakest Preconditions:

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

- Proof Obligation (PO):

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

Hoare Logic

```
/*@ 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; */
}
```

- Hoare Triples:

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

- Weakest Preconditions:

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

- Proof Obligation (PO):

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

WP plug-in

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;
}
```

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)

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

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)

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

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)

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

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)

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

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:

$$\mathrm{R\_2} \Rightarrow wp(\mathrm{S\_1}, \mathrm{R\_1})$$

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

$$\forall State, \mathrm{E\_1} \Rightarrow wp(\mathrm{S\_2}, \mathrm{E\_2})$$

- Must specify effects (Frame rule)

$$\forall x \in State \backslash A, \mathrm{g\ does\ not\ change}\ x$$

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

frama C
Software Analyzers

Loops

```
/*@ 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: $\forall x \in State \backslash A$, B does not change x

Loops

```
/*@ 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 $R \Rightarrow wp(\mathtt{S\_1}, \mathtt{I})$
  - If it holds after n steps, it holds after $n+1$ steps. PO is $\forall State, I \wedge \mathtt{e} \Rightarrow wp(\mathtt{B}, \mathtt{I})$
  - Must imply the post-condition. PO is $\forall State, I \wedge \neg \mathtt{e} \Rightarrow wp(\mathtt{S\_2}, \mathtt{E})$
- Specify effects of the loop: $\forall x \in State \backslash A, \mathtt{B}$ does not change x

Loops

```
/*@ 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 $R \Rightarrow wp(\mathtt{S\_1}, \mathtt{I})$
  - If it holds after n steps, it holds after $n+1$ steps. PO is $\forall State, I \wedge \mathtt{e} \Rightarrow wp(\mathtt{B}, \mathtt{I})$
  - Must imply the post-condition. PO is $\forall State, I \wedge \neg \mathtt{e} \Rightarrow wp(\mathtt{S\_2}, \mathtt{E})$
- Specify effects of the loop: $\forall x \in State \backslash A, \mathtt{B}$ does not change x

Loops

```
/*@ 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 $R \Rightarrow wp(\mathtt{S\_1}, \mathtt{I})$
  - If it holds after n steps, it holds after $n+1$ steps. PO is $\forall State, I \wedge \mathtt{e} \Rightarrow wp(\mathtt{B}, \mathtt{I})$
  - Must imply the post-condition. PO is $\forall State, I \wedge \neg \mathtt{e} \Rightarrow wp(\mathtt{S\_2}, \mathtt{E})$
- Specify effects of the loop: $\forall x \in State \backslash A, \mathtt{B}$ does not change x

Loops

```
/*@ 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 $R \Rightarrow wp(\mathtt{S\_1}, \mathtt{I})$
  - If it holds after n steps, it holds after $n+1$ steps. PO is $\forall State, I \wedge \mathtt{e} \Rightarrow wp(\mathtt{B}, \mathtt{I})$
  - Must imply the post-condition. PO is $\forall State, I \wedge \neg \mathtt{e} \Rightarrow wp(\mathtt{S\_2}, \mathtt{E})$
- Specify effects of the loop: $\forall x \in State \backslash A, \mathtt{B}$ does not change x

Loops

```
/*@ 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 $R \Rightarrow wp(\mathtt{S\_1}, \mathtt{I})$
  - If it holds after n steps, it holds after $n+1$ steps. PO is $\forall State, I \wedge \mathtt{e} \Rightarrow wp(\mathtt{B}, \mathtt{I})$
  - Must imply the post-condition. PO is $\forall State, I \wedge \neg \mathtt{e} \Rightarrow wp(\mathtt{S\_2}, \mathtt{E})$
- Specify effects of the loop: $\forall x \in State \backslash A, \mathtt{B}$ 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 ==> \result >= a[i];
    ensures \exists integer i;
     0<=i<length &&  \result == a[i];
*/
int max_array(int* a, int length) {
```

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

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

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

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;

  */
  }
  return 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;

  */
  }
  return 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;

  */
  }
  return 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;

  */
  }
  return 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

Behaviors

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

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

Algorithm

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

mjrty: logic functions

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

mjrty: 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;
```

mjrty: loop annotations

```
loop invariant 1<= i <= length;
loop invariant cnt <= count(a,i,cand);
loop invariant \forall 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;
```

frama C
Software Analyzers

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)