Frama-C WP Tutorial

Virgile Prevosto

October 13th, 2014

Motivation

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

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

ACSL: ANSI/ISO C Specification Language

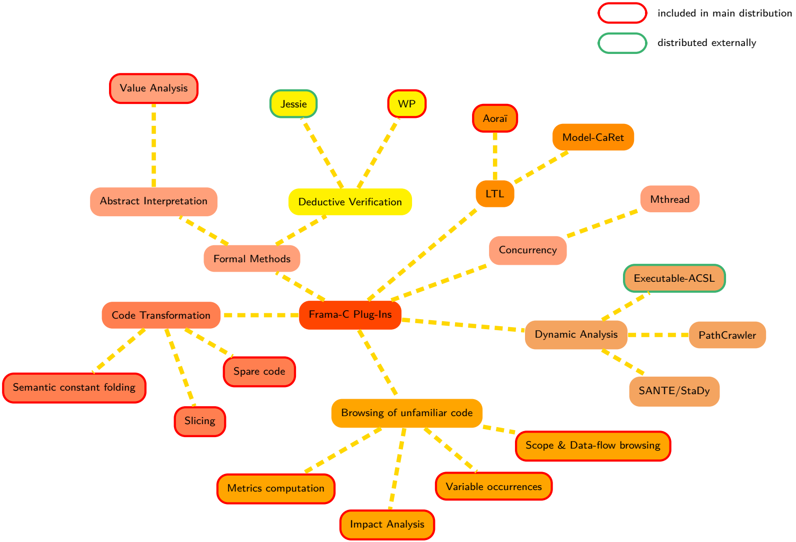
Presentation

- 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)

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):

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

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):

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

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):

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

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):

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

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`

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

Setting values

Example

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

Demo

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:

$$R\_2 \Rightarrow wp(S\_1, R\_1)$$

- 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 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

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

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 $\forall State, I \land \mathtt{e} \Rightarrow wp(\mathtt{B}, \mathtt{I})$
  - Must imply the post-condition. PO is $\forall State, I \land \neg \mathtt{e} \Rightarrow wp(\mathtt{S\_2}, \mathtt{E})$
- 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: $\forall x \in State \backslash A$, 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 $R \Rightarrow wp(\texttt{S\_1}, \texttt{I})$
  - If it holds after n steps, it holds after $n+1$ steps. PO is $\forall State, I \wedge \texttt{e} \Rightarrow wp(\texttt{B}, \texttt{I})$
  - Must imply the post-condition. PO is $\forall State, I \wedge \neg\texttt{e} \Rightarrow wp(\texttt{S\_2}, \texttt{E})$
- Specify effects of the loop: $\forall x \in State \setminus A$, 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 $R \Rightarrow wp(\texttt{S\_1}, \texttt{I})$
  - If it holds after n steps, it holds after $n+1$ steps. PO is $\forall State, I \wedge \texttt{e} \Rightarrow wp(\texttt{B}, \texttt{I})$
  - Must imply the post-condition. PO is $\forall State, I \wedge \neg \texttt{e} \Rightarrow wp(\texttt{S\_2}, \texttt{E})$
- Specify effects of the loop: $\forall x \in State \setminus A$, 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 $R \Rightarrow wp(\texttt{S\_1}, \texttt{I})$
  - If it holds after n steps, it holds after $n+1$ steps. PO is $\forall State, I \wedge \texttt{e} \Rightarrow wp(\texttt{B}, \texttt{I})$
  - Must imply the post-condition. PO is $\forall State, I \wedge \neg\texttt{e} \Rightarrow wp(\texttt{S\_2}, \texttt{E})$
- Specify effects of the loop: $\forall x \in State \backslash A$, $\texttt{B}$ 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

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

- 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

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

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

```c
/* 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

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) {
```

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) {
```

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) {
```

Binary search: relations between behaviors

```
/*@

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

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

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

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) {
```

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

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)