

# Frama-C and ACSL

(Based on [The Frama-C Website, V. Prevosto 2013,  
A. Blanchard 2020])

Yih-Kuen Tsay  
(with help from Ming-Hsien Tsai and Yi-Fan Tsai)

Dept. of Information Management  
National Taiwan University

- 🌐 Frama-C is an open-source extensible and collaborative platform dedicated to source-code analysis of C programs.
  - 🌐 With suitable plug-ins, it is a tool for formal verification (among several source-code analysis features) of C code with specification written in ACSL (ANSI/ISO C Specification Language).
  - 🌐 Two most relevant plug-ins for (Hoare Logic-based) formal verification are:
    - ☀ WP
    - ☀ Jessie
- Both in turn rely on external automatic or interactive provers such as Alt-Ergo and Coq to complete the verification tasks.
- 🌐 Given an ACSL-annotated program, both WP and Jessie can generate verification conditions using weakest-precondition calculus and send them to designated provers.




- 🌐 The Frama-C kernel is based on a modified version of CIL, which is a front-end for C that parses ISO C99 programs into a normalized representation.
- 🌐 Frama-C extends CIL to support ACSL.
- 🌐 This modified CIL front-end produces the C + ACSL abstract syntax tree (AST), an abstract view of the program shared among all analyzers.
- 🌐 Analyzers are developed as separate plug-ins on top of the kernel.
- 🌐 Plug-ins are dynamically linked against the kernel to offer new analyses, or to modify existing ones.
- 🌐 Any plug-in can register new services in a plug-in database stored in the kernel, thereby making these services available to all plug-ins.

- 🌐 ACSL (ANSI/ISO C Specification Language) is a behavioral interface specification language implemented in the Frama-C framework.
- 🌐 It allows for formally specifying the behavioral properties of a C program, in order to be able to formally verify the implementation with respect to these properties.
- 🌐 ACSL was inspired by and resembles JML.

## A First Example

```
1 /*@ ensures \result >= x && \result >= y;  
2     ensures \result == x || \result == y;  
3 */  
4 int max (int x, int y) { return (x > y) ? x : y; }
```

- So, ACSL annotations are written in special C comments, the difference with plain comments being that annotations begin with `/*@`.
- It is also possible to write one-line annotations introduced by `//@`.
- The “function contract” is written immediately above the function declaration.
- In this example, the contract contains only post-conditions (`ensures` clauses), as `max` does not have any particular requirement.
- The keyword `\result` denotes the returned value of the function.

-  Complete specification  
Any function that satisfies the specification should be deemed a satisfactory implementation.
-  Partial specification  
Partial specifications express some of the properties that are expected to hold for any implementation.
-  Generally speaking, partial formal specifications are the most likely to be encountered in practice for real-life examples.

# An Example of Partial Specification

```
1 /*@ requires \valid(p) && \valid(q);
2     ensures *p <= *q;
3 */
4 void max_ptr(int *p, int *q)
5 {
6     *p = *q = 0;
7 }
```

- 🌐 The `requires` clause specifies the pre-condition.
- 🌐 The predicate `\valid(p)` indicates that the address `p` is included in an allocated block which is large enough to store an `int` starting at `p`.
- 🌐 The partial specification may be met by an unwanted implementation.



```
1 /*@ requires n > 0;
2 requires \valid(p+ (0..n-1));
3 ensures \forallall int i; 0 <= i <= n - 1 ==> \result >= p[i];
4 ensures \exists int e; 0 <= e <= n - 1 && \result == p[e];
5 */
6 int max_seq(int * p, int n) {
7     int res = *p;
8     for (int i = 0; i < n; i++) {
9         if (res < *p) { res = *p; }
10        p++;
11    }
12    return res;
13 }
```

Problem: the specification does not prevent the implementation from altering the integer array.



## Function Contract (cont.)

```
1  /*@ requires n > 0;
2     requires \valid(p+ (0..n-1));
3     ensures \forallall int i; 0 <= i <= n - 1 ==> p[i] == \old(p[i]);
4     ensures \forallall int i; 0 <= i <= n - 1 ==> \result >= p[i];
5     ensures \exists int e; 0 <= e <= n - 1 && \result == p[e];
6  */
7  int max_seq(int * p, int n) {
8     int res = *p;
9     for (int i = 0; i < n; i++) {
10        if (res < *p) { res = *p; }
11        p++;
12    }
13    return res;
14 }
```

-  The red line ensures that the content of the integer array is not modified.
-  The built-in logic function `\old` gets the old (i.e., before the call) value of a given element.

```
1 /*@ requires \valid(p) && \valid(q);
2     assigns *p, *q;
3 */
4 void swap (int *p, int *q) {
5     int tmp = *p;
6     *p = *q;
7     *q = tmp;
8     return ;
9 }
```

- When no **assigns** clauses are specified, the function is allowed to modify any visible variable.
- If there are **assigns** clauses specified, the function can only modify the content of the locations that are explicitly mentioned in the clauses.

## Assigns Clauses (cont.)

```
1 /*@ requires n > 0;
2    requires \valid(p+ (0..n-1));
3    assigns \nothing;
4    ensures \forall int i; 0 <= i <= n - 1 ==> \result >= p[i];
5    ensures \exists int e; 0 <= e <= n - 1 && \result == p[e];
6 */
7 int max_seq(int * p, int n) {
8     int res = *p;
9     for (int i = 0; i < n; i++) {
10        if (res < *p) { res = *p; }
11        p++;
12    }
13    return res;
14 }
```

The red line ensures that the integer array  $p$  and the integer  $n$  are not modified.

```
1 int max_seq (int* p, int n) {
2   int res = *p;
3   /*@ loop invariant
4     \forall integer j; 0 <= j <= i ==> res >= *(p + j);
5   */
6   for (int i = 0; i < n; i++) {
7     if (res < *p) { res = *p; }
8     p++;
9   }
10  return res;
11 }
```



- Termination is guaranteed by attaching a measure function to each loop and each recursive function.
- Loops are annotated by “//@ loop variant e;”  
For each iteration, the value of e at the end of the iteration must be smaller than its value at the beginning. Its value at the beginning must be nonnegative.
- Functions are annotated by “//@ decreases e;”  
Each function must be annotated with the same decreases clause. Any recursive call must occur in a state where the measure is smaller than the measure in the pre-state.
- Functions can also be annotated by “//@ terminates p;”  
If p holds, the function is guaranteed to terminate.

## Termination (cont.)

```
1 /*@ assigns \nothing
2     terminates c > 0;
3 */
4 void f (int c) {
5     while (!c)
6         ;
7     return ;
8 }
```

The function  $f$  can be called with any argument  $c$ , but the function is not guaranteed to terminate if  $c \leq 0$ .

```
1 /*@ requires n >= 0 && n < 100;
2 */
3 int f (int n) {
4     int tmp = 100 - n;
5     //@ assert tmp > 0;
6     //@ assert tmp < 100;
7     return tmp;
8 }
```

-  An annotation “assert  $p$ ;” means that  $p$  must hold in the current state.
-  Assertions are useful to provide hints for the generation of verification conditions.

```
1  typedef enum { Max, Min } kind;  
2  
3  /*@ requires k == Max || k == Min;  
4     assigns \nothing;  
5     ensures \result == x || \result == y;  
6     ensures k == Max ==> \result >= x && \result >= y;  
7     ensures k == Min ==> \result <= x && \result <= y;  
8  */  
9  int extremum (kind k, int x, int y) {  
10     return ((k == Max ? x > y : x < y) ? x : y);  
11 }
```

This specification may not be clear to say that `extremum` has two distinct modes of operation.



## Behaviors (cont.)

Behaviors allow us to specify the different cases for postconditions.

```
1 /*@ requires k == Max || k == Min;
2   assigns \nothing;
3   ensures \result == x || \result == y;
4   behavior is_max:
5     assumes k == Max;
6     ensures \result >= x && \result >= y;
7   behavior is_min:
8     assumes k == Min;
9     ensures \result <= x && \result <= y;
10  complete behaviors is_max, is_min;
11  disjoint behaviors is_max, is_min;
12 */
13 int extremum (kind k, int x, int y);
```

# Constructs for Specifying Properties

- 🌐 Predicates and functions provided by ACSL: `\valid`, `\separated`, `\old`, `\at`, etc.
- 🌐 Logical types provided by ACSL: `integer`, `real`, etc. Mathematical integers and reals are more general than `ints` and `floats` in C.
- 🌐 User-defined predicates and functions

```
1 /*@
2   predicate unchanged{L0, L1}(int* i) =
3     \at(*i, L0) == \at(*i, L1);
4 */
```

```
1 /*@
2   logic integer ax_b(integer a, integer x, integer b) =
3     a * x + b;
4 */
```

- 🌐 Lemmas: user-specified properties about predicates and functions

- 🌐 Lemmas can be proved in isolation of the rest of the proof of a program by automatic or (more often) interactive provers.
- 🌐 Once the proof is done, the information that it states can be safely used to simplify the reasoning in other, more complex proofs, without having to prove it again.

```
1 /*@
2 lemma lt_plus_lt:
3   \forall integer i, j ; i < j ==> i+1 < j+1;
4 */
```

# Inductive Definitions

Inductive predicates give a way to state properties whose verification requires reasoning by induction.

```
1 /*@
2   inductive zeroed{L}(int* a, integer b, integer e){
3     case zeroed_empty{L}:
4       \forall int* a, integer b, e; b >= e ==> zeroed{L}(a,b,e);
5     case zeroed_range{L}:
6       \forall int* a, integer b, e; b < e ==>
7         zeroed{L}(a,b,e-1) && a[e-1] == 0 ==> zeroed{L}(a,b,e);
8   }
9 */
```

```
1 /*@
2   requires \valid(array + (0 .. length - 1));
3   assigns array[0 .. length - 1];
4   ensures zeroed(array, 0, length);
5 */
6 void reset(int* array, size_t length){
7   /*@
8     loop invariant 0 <= i <= length;
9     loop invariant zeroed(array, 0, i);
10    loop assigns i, array[0 .. length - 1];
11    loop variant length - i;
12   */
13   for(size_t i = 0; i < length; ++i)
14     array[i] = 0;
15 }
```

# Axiomatic Definitions




Instead of defining directly a function or predicate, an axiomatic definition declares it and then defines axioms that specify its behavior.

```
1  /*@
2   axiom A_all_zeros{
3     predicate zeroed{L}(int* a, integer b, integer e)
4       reads a[b .. e-1];
5
6     axiom zeroed_empty{L}:
7       \forall int* a, integer b, e; b >= e ==> zeroed{L}(a,b,e);
8
9     axiom zeroed_range{L}:
10      \forall int* a, integer b, e; b < e ==>
11        zeroed{L}(a,b,e-1) && a[e-1] == 0 ==> zeroed{L}(a,b,e);
12  }
13  */
```

## Ghost Variables

Ghost variables and statements are like C variables and statements, but visible only in the specification.

```
1  int max_seq (int* p, int n) {
2      int res = *p;
3      /*@ ghost int e = 0;
4          loop invariant \forall int j;
5                          0 <= j <= i ==> res >= p[j];
6          loop invariant \forall \valid(p+e) && p[e] == res;
7      */
8      for (int i = 0; i < n; i++) {
9          if (res < *p) {
10             res = *p;
11             //@ ghost e = i;
12         }
13         p++;
14     }
15     return res;
16 }
```

-  The Frama-C Website: <https://www.frama-c.com/>
-  V. Prevosto. *ACSL Mini-Tutorial*, CEA LIST and INRIA, 2013.
-  A. Blanchard. *Introduction to C program proof with Frama-C and its WP plugin*, Creative Commons, 2020.