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: Introduction

- 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).
- The most relevant plug-in of Frama-C for (Hoare Logic-based) formal verification in this course is WP.
- WP in turn relies on external automatic or interactive provers such as Alt-Ergo and Coq to complete the verification tasks, with Why3 as the bridge.
- Given an ACSL-annotated program, WP generates verification conditions using weakest-precondition calculus and sends them to designated provers.

Software Architecture

- 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: Introduction

- 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

```
/*@ ensures \result >= x && \result >= y;
    ensures \result == x || \result == y;
*/
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.

Regarding Formal Specifications

- 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

```
/*@ requires \valid(p) && \valid(q);
    ensures *p <= *q;
*/
void max_ptr(int *p, int *q)
{
    *p = *q = 0;
}
```

- 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.

Function Contract

```
/*@ requires n > 0;
    requires \valid(p+ (0..n-1));
    ensures \forall int i; 0 <= i <= n - 1 ==> \result >= p[i];
    ensures \exists int e; 0 <= e <= n - 1 && \result == p[e];
*/
int max_seq(int * p, int n) {
  int res = *p;
  for (int i = 0; i < n; i++) {
    if (res < *p) { res = *p; }
    p++;
  }
  return res;
}
```

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

Function Contract (cont.)

```
/*@ requires n > 0;
    requires \valid(p+ (0..n-1));
    ensures \forall int i; 0 <= i <= n - 1 ==> p[i] == \old(p[i]);
    ensures \forall int i; 0 <= i <= n - 1 ==> \result >= p[i];
    ensures \exists int e; 0 <= e <= n - 1 && \result == p[e];
*/
int max_seq(int * p, int n) {
  int res = *p;
  for (int i = 0; i < n; i++) {
    if (res < *p) { res = *p; }
    p++;
  }
  return res;
}
```

- 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.

Assigns Clauses

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

- 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.)

```
/*@ requires n > 0;
    requires \valid(p+ (0..n-1));
    assigns \nothing;
    ensures \forall int i; 0 <= i <= n - 1 ==> \result >= p[i];
    ensures \exists int e; 0 <= e <= n - 1 && \result == p[e];
*/
int max_seq(int * p, int n) {
  int res = *p;
  for (int i = 0; i < n; i++) {
    if (res < *p) { res = *p; }
    p++;
  }
  return res;
}
```

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

Loop Invariants

```
int max_seq (int* p, int n) {
  int res = *p;
  /*@ loop invariant
      \forall integer j; 0 <= j <= i ==> res >= *(p + j);
  */
  for (int i = 0; i < n; i++) {
    if (res < *p) { res = *p; }
    p++;
  }
  return res;
}
```

- Recall (as a special case of a theorem we studied earlier) that, if $P \wedge B \Rightarrow wlp(S, P)$, then $P \Rightarrow wlp(\text{do } B \rightarrow S \text{ od}, P \wedge \neg B)$.
- P here is usually called an *inductive* loop invariant.

Termination

- 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.)

```
/*@ assigns \nothing
    terminates c > 0;
*/
void f (int c) {
  while(!c)
    ;
  return;
}
```

The function `f` can be called with any argument `c`, but the function is not guaranteed to terminate if `c <= 0`.

Assertions

```
/*@ requires n >= 0 && n < 100;
*/
int f (int n) {
   int tmp = 100 - n;
   //@ assert tmp > 0;
   //@ assert tmp < 100;
   return tmp;
}
```

- 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.

Behaviors

```
typedef enum { Max, Min } kind;

/*@ requires k == Max || k == Min;
    assigns \nothing;
    ensures \result == x || \result == y;
    ensures k == Max ==> \result >= x && \result >= y;
    ensures k == Min ==> \result <= x && \result <= y;
*/
int extremum (kind k, int x, int y) {
  return ((k == Max ? x > y : x < y) ? x : y);
}
```

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.

```
/*@ requires k == Max || k == Min;
    assigns \nothing;
    ensures \result == x || \result == y;
    behavior is_max:
      assumes k == Max;
      ensures \result >= x && \result >= y;
    behavior is_min:
      assumes k == Min;
      ensures \result <= x && \result <= y;
    complete behaviors is_max, is_min;
    disjoint behaviors is_max, is_min;
*/
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 `int`s and `float`s in C.
- User-defined predicates and functions

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

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

- Lemmas: user-specified properties about predicates and functions

Lemmas

- 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.

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

Inductive Definitions

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

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

So, for instance, zeroed$(A, 0, 6)$ asserts that array A contains only 0's in $A[0..5]$ (where 5 comes from $6 - 1$).

Inductive Definitions (cont.)

```
/*@
  requires n >= 1 && \valid(array + (0 .. n-1));
  assigns array[0 .. n-1];
  ensures zeroed(array,0,n);
*/
void reset(int* array, int n){
  /*@
    loop invariant 0 <= i <= n;
    loop invariant zeroed(array,0,i);
    loop assigns i, array[0 .. n-1];
    loop variant n-i;
  */
  for(int i = 0; i < n; ++i)
    array[i] = 0;
}
```

Axiomatic Definitions

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

```
/*@
  axiomatic A_all_zeros{
    predicate zeroed{L}(int* a, integer b, integer e)
      reads a[b .. e-1];

    axiom zeroed_empty{L}:
      \forall int* a, integer b, e; b >= e ==> zeroed{L}(a,b,e);

    axiom zeroed_range{L}:
      \forall int* a, integer b, e; b < e ==>
        zeroed{L}(a,b,e-1) && a[e-1] == 0 ==> zeroed{L}(a,b,e);
  }
*/
```

Ghost Variables

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

```
int max_seq (int* p, int n) {
  int res = *p;
  /*@ ghost int e = 0;
      loop invariant \forall int j;
                0 <= j <= i ==> res >= p[j];
      loop invariant
                \valid(\at(p,Pre)+e) && \at(p,Pre)[e] == res;
      ....
  */
  for (int i = 0; i < n; i++) {
    if (res < *p) {
      res = *p;
      //@ ghost e = i;
    }
    p++;
  }
  return res;
}
```

References

- 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.