Frama-C WP Tutorial

Virgile Prevosto, Nikolay Kosmatov and Julien Signoles

June 11th, 2013

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
- ...

Our goal

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

- Presentation of Frama-C
  - Context
  - First steps
  - Frama-C plugins
- Basic function contract
  - A little bit of background
  - ACSL and WP
  - Specifying side-effects
- Loops
  - Background
  - Loop invariants in ACSL
  - Loop termination
- Advanced contracts
  - Behaviors
  - User-defined predicates

- Presentation of Frama-C
  - Context
  - First steps
  - Frama-C plugins
- Basic function contract
  - A little bit of background
  - ACSL and WP
  - Specifying side-effects
- Loops
  - Background
  - Loop invariants in ACSL
  - Loop termination
- Advanced contracts
  - Behaviors
  - User-defined predicates

A brief history

- 90's: CAVEAT, an Hoare logic-based tool for C programs at CEA
- 2000's: CAVEAT used by Airbus during certification process of the A380 (DO-178 level A qualification)
- 2002: Why and its C front-end Caduceus (at INRIA)
- 2006: Joint project to write a successor to CAVEAT and Caduceus
- 2008: First public release of Frama-C (Hydrogen)
- 2010: start of Device-Soft project between Fraunhofer FIRST (now FOKUS) and CEA LIST
- today:
  - Frama-C Fluorine (v9)
  - Multiple projects around the platform
  - A growing community of users
  - and of plug-ins developers

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 (Fluorine v1/v2 in April-May)
- 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 the users to specify functional properties of their programs
- 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)`

Installation

On Linux

- On Debian, Ubuntu, Fedora, Gentoo, OpenSuse, Linux Mint, ...
- Compile from sources using OCaml package managers:
  - Godi (`http://godi.camlcity.org/godi/index.html`)
  - Opam (`http://opam.ocamlpro.com/`)

On Windows

- Godi
- Wodi (`http://wodi.forge.ocamlcore.org/`)

On Mac OS X

- Binary package available
- Source compilation through homebrew.

Installed files

Executables

- `frama-c`: Console-based interface
- `frama-c-gui`: Graphical User Interface

Others

- `FRAMAC_PLUGINS`: location of plug-ins
- `FRAMAC_SHARE`: various configuration files
- `FRAMAC_SHARE/libc`: standard headers

Documentation

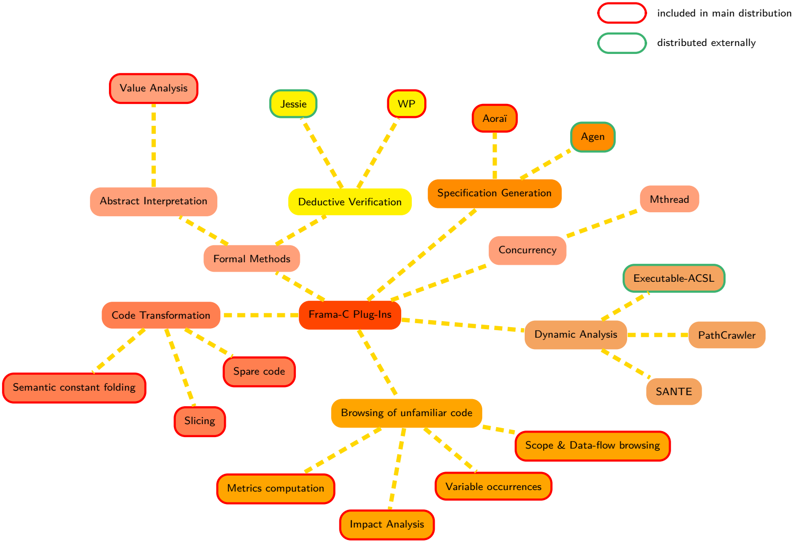
Manuals

- `http://frama-c.com/support.html`
- In directory `$(frama-c -print-share-path)/manuals`
- inline help (`frama-c -kernel-help`, `frama-c -`*`plugin`*`-help`)

Support

- `frama-c-discuss@gforge.inria.fr`
- tag `frama-c` on `http://stackoverflow.com`

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)

- Presentation of Frama-C
  - Context
  - First steps
  - Frama-C plugins
- Basic function contract
  - A little bit of background
  - ACSL and WP
  - Specifying side-effects
- Loops
  - Background
  - Loop invariants in ACSL
  - Loop termination
- Advanced contracts
  - Behaviors
  - User-defined predicates

Summary

Contracts

Goal: specification of imperative functions

Approach: give assertions (i.e. properties) about the functions

- **Precondition** is supposed to be true on entry (ensured by callers of the function)
- **Postcondition** must be true on exit (ensured by the function if it terminates)

Nothing is guaranteed when the precondition is not satisfied

Termination may or may not be guaranteed (total or partial correctness)

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):

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

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):

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

A first example

```c
#include "limits.h"
// returns the maximum of x and y
int max ( int x, int y ) {
  if ( x >=y )
    return x ;
  return y ;
}
```

WP plug-in

Credits

- Loïc Correnson
- Zaynah Dargaye
- Anne Pacalet
- François Bobot
- a few others

Basic usage

- `frama-c-gui -wp file.c`
- WP tab on the GUI
- Inspect (failed) proof obligation
- `http://frama-c.com/download/wp-manual.pdf`

Avoiding run-time errors

Example

```c
// returns the absolute value of x
int abs ( int x ) {
  if ( x >=0 )
    return x ;
  return -x ;
}
```

Command

- `frama-c-gui -pp-annot -wp -wp-rte abs.c`
- or use switch directly in GUI

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 ;
}
```

Main ingredients

- built-in predicate `\valid(...)`
- **assigns** clause

Setting values

Example

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

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)

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

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:

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

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

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

- Must specify effects (Frame rule)

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

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:

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

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

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

- Must specify effects (Frame rule)

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

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:

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

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

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

- Must specify effects (Frame rule)

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

Function call: example

```
#include "limits.h"
/*@
  requires \valid(p) && \valid(q);
  ensures \result >= *p && \result >= *q;
  ensures \result == *p || \result == *q;
  assigns \nothing;
*/
int max ( int* p, int* q );
```

Function call: example (cont'd)

```
/*@
  requires \valid(p) && \valid(q);
  assigns *x, *y;
  ensures *x == \at(*y,Pre);
  ensures *y == \at(*x,Pre);
*/
void swap(int* x, int* y);

// ensures that *high contains
// the maximum of the two values.
int max_swap( int* low, int* high ) {
  if (*high != max(low,high)) swap(low,high);
}
```

Presentation of Frama-C

- Context
- First steps
- Frama-C plugins

Basic function contract

- A little bit of background
- ACSL and WP
- Specifying side-effects

Loops

- Background
- Loop invariants in ACSL
- Loop termination

Advanced contracts

- Behaviors
- User-defined predicates

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 $R \Rightarrow wp(\mathtt{S\_1}, \mathtt{I})$
  - 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 $R \Rightarrow wp(\texttt{S\_1}, \texttt{I})$
  - 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: $\forall x \in State \backslash A$, 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 $R \Rightarrow wp(\texttt{S\_1}, \texttt{I})$
  - 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: $\forall x \in State \setminus A$, $\texttt{B}$ 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: $\forall x \in State \setminus A$, $\mathtt{B}$ 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 $R \Rightarrow wp(\texttt{S\_1}, \texttt{I})$
  - 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

Loops: example

```
// returns a non-zero value iff all elements
// in a given array t of n integers are zeros
int all_zeros(int t[], int n) {
  int k;
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

Loop invariants - some hints

How to find a suitable loop invariant? Consider two aspects:

- identify locations modified in the loop
  - define their possible value intervals (relationships) after *k* iterations
  - use **`loop assigns`** clause to list variables that (might) have been assigned so far after *k* iterations
- identify realized actions, or properties already ensured by the loop
  - what part of the job already realized after *k* iterations?
  - what part of the expected loop results already ensured after *k* iterations?
  - why the next iteration can proceed as it does? ...

A stronger property on each iteration may be required to prove the final result of the loop.

Loop invariants - more hints

Remember: a loop invariant must be true

- before (the first iteration of) the loop, even if no iteration is possible
- after any complete iteration even if no more iterations are possible
- in other words, any time right before the loop condition check

In particular, a **for** loop

```
for(i=0; i<n; i++) { /* body */ }
```

should be seen as

```
i=0;            // action before the first iteration
while( i<n )    // an iteration starts by the condition check
  {
    /* body */
    i++;        // last action in an iteration
  }
```

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 variants - some hints

- Unlike an invariant, a loop variant is an integer expression, not a predicate
- Loop variant is not unique: if V works, $V+1$ works as well
- No need to find a precise bound, any working loop variant is OK
- To find a variant, look at the loop condition
  - For the loop `while(exp1 > exp2 )`, try `loop variant exp1-exp2;`
- In more complex cases: ask yourself why the loop terminates, and try to give an integer upper bound on the number of remaining loop iterations

Presentation of Frama-C

- Context
- First steps
- Frama-C plugins

Basic function contract

- A little bit of background
- ACSL and WP
- Specifying side-effects

Loops

- Background
- Loop invariants in ACSL
- Loop termination

Advanced contracts

- Behaviors
- User-defined predicates

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

Using behaviors: example

```
/* input: a sorted array a, its length,
   and a value key to search.
   output: index of a cell which contains key,
   or -1 if 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;
}
```

A look a C strings

From C std library

```
#include "limits.h"

typedef unsigned int size_t;

void* memcpy(void* dest, void* src, size_t length);

size_t strlen(char* s);

char* strcpy(char *s1, const char* s2);

char *strncpy(char* s1, const char *s2, size_t n);
```