Tutorial on Frama-C WP

Sylvain Boulmé
mailto:Sylvain.Boulme@univ-grenoble-alpes.fr

MOSIG 2023-2024

FRAMA-C is a static analysis tool for C language developed by CEA-List and, historically, with Toccata team of Inria-Saclay & Université Paris-Saclay. It allows to verify that the source code complies with a provided formal functional specification. These specifications are written in a dedicated language, ACSL. The specifications can be partial, concentrating on one aspect of the analyzed program at a time.

FRAMA-C features several plugins. In this tutorial, we mainly focus on WP plugin, that implements a *weakest liberal precondition calculus* for ACSL annotations over C programs. For each code annotation, this technique generates a bundle of "*WP goals*" (also called "*Verifying Conditions*" or "*Proof Obligations*"). These goals express that the annotations are satisfied by the code as mathematical first-order logic formula that can be verified by a SMT-solver like ALT-ERGO (other provers are also supported through the Why platform).

We will also use RTE plugin, that inserts annotations in the C sources, in order to ensure absence of runtime errors (e.g. *undefined behaviors* like invalid memory accesses or arithmetic overflows). Hence, typically WP will check assertions generated by RTE thanks to goals discharged by ALT-ERGO.

1 A brief introduction to ACSL and WP

On the contrary to some other plugins of FRAMA-C, such as EVA, WP performs a modular check of the code : each function is checked independently of its clients, and inside each function body, each loop is itself verified as a kind of independent local function (the `main` function is thus not mandatory). Modularity makes verification much easier, but contradicts a bit the claim that "*specifications can be partial*". Indeed, WP considers that all functions and loops have their own specifications which must be satisfied independently of the remaining code. When these specifications are not given explicitly in ACSL annotations of the source, WP uses an implicit specification meaning that "*anything can happen*". Thus – in practice – in order to check a particular property (e.g. absence of runtime errors) on a function g that invokes an other function f, the user has to provide an independent specification of f that will allow to prove the particular property on g. As a consequence, mastering ACSL annotations is mandatory to use WP plugin (on the contrary to other plugins of FRAMA-C).

ACSL annotations are given directly in C source as comments starting with symbol `@`. These annotations often start with a keyword like `requires` or `ensures` followed by a logical predicate on the memory locations (i.e. program variables and the implicit heap). The syntax of such predicates is inspired from "Boolean expressions" of C language. However, all these expressions must be *pure* (i.e. without side-effects).

2 Verifying functions with ACSL and WP

In ACSL, a function specification describes how the memory is transformed by any call to this function. If all goals associated to the function implementation have been proved, then each function call satisfies the specification (otherwise there is a bug somewhere in the environment : FRAMA-C, ALT-ERGO, the C compiler, the OS or the computer). Let us detail this idea on the following specification of `div` function that computes the euclidean division of `x` by `y`, and stores the quotient in `*q` and the remainder in `*r`.

```
/*@ requires 0 <= x && 0 < y;
  @ assigns *q, *r;
  @ ensures x == *q * y + *r && 0 <= *r < y; */
void div(int x,int y, int* q, int* r) {
  ...
}
```

Such a specification can be interpreted as a pair of *precondition/postcondition.* Keyword `requires` introduces the *precondition* : a condition on the memory state at the initial state of the call. The *postcondition* – the condition on the final state – is given by keywords `assigns` (which is followed by a set of memory locations) and `ensures`. This specification expresses that each call to `div` satisfies the following property

Given the respective actual values x_a, y_a, q_a, r_a of variables `x`, `y`, `q`, `r` *at runtime,*

- *if* `div` *terminates without running into an undefined behavior (e.g. an invalid memory access) or an integer overflow,*
- *and, if* $0 \leq x_a$ *and* $0 < y_a$ **(clause `requires`)**

then, ***after the call****,*

(clause `assigns`) *the value of any memory location distinct of q_a and r_a pointers is* unchanged *w.r.t to its value before the call;*

(clause `ensures`) *the respective values q_v and r_v at locations q_a and r_a satisfy*

$$x_a = q_v \times y_a + r_v \quad \wedge \quad 0 \leq r_v < y_a$$

Note that locations q_a and r_a may have been assigned by the call, but they also may not. All we know on the values stored at q_a and r_a is expressed by clause `ensures`. Actually, `assigns` clause expresses a postcondition (on the global memory state) about memory locations that may be unknown at this point of the source code. For instance, let us consider the following code (inside an other function) :

```
int a = 10, q, r;
div(2*a, a-7, &q, &r);
```

If `div` *safely* terminates (i.e. without running into an undefined behavior or an integer overflow), we know that, since $2 \times 10 = 20 \geq 0$ and $10 - 7 = 3 > 0$, the values of variables after the call satisfy :

- `a == 10` (it is unchanged) ;
- `20 == q * 3 + r` and `0 <= r < 3`. In other words, `q == 6` and `r == 2`.

Let us now consider this weird alternative code :

```
int a = 10;
div(2*a, a-7, &a, &a);
```

Here, the value of `a` after the call satisfies `20 == a * 3 + a` with `0 <= a < 3`. This is unsatisfiable (because the equality implies `a == 5`). Hence, there is *no safely terminating* implementation of `div` that satisfies the above specification !

This last example illustrates that aliasing of memory locations (here between parameter `q` and `r`) makes reasoning on programs tricky ! Fixing by yourself this specification (for a provided safely terminating implementation) is one of your goal during this tutorial. Let us thus consider a simpler example that illustrates other important concepts of ACSL :

```
/*@ assigns *x;
  @ ensures \result == \old(*x) && *x == \result+1; */
int incr(int* x) {
  return (*x)++;
}
```

Here, absence of `requires` means that precondition is trivially true. In `ensures` clause, `\result` is a special variable representing any result returned by the function (through **`return`** keyword of C). And, an expression like `\old(`e`)` evaluates e in the *initial* state of the call. Hence, `\old` is a way to express a relation between the initial state and the final state in `ensures` clause. In other words, if `*x` is initially an integer n then `incr(x)` returns n and `*x` is finally $n+1$.

WP plugin checks that the provided implementation of `incr` satisfies the specification. Let us consider the following `main` function :

```
1 #define MAXINT 2147483647
2
3 int main() {
4   int x = MAXINT;
5   int r = incr(&x);
6   /*@ assert x > r; */
7   if (x <= r) return 1/(r+1-x);
8   return x;
9 }
```

The `assert` clause expresses a condition that must be satisfied at this control point (in any safely terminating execution). Below, we run WP with the following command on the whole program (containing `incr` & `main`) :

```
frama-c -wp incr0.c
```

we get the following answer :

```
[wp] 4 goals scheduled
[wp] Proved goals:    4 / 4
     Qed:             4  (4ms)
```

This means that WP has generated 4 goals to check assertions of this program which have all been proved by `Qed` solver (an intern solver of WP dedicated to trivial goals). However, at runtime, we observe that a division-by-zero happens at line 7! Actually, assertion at line 6 is also not satisfied. This is because `incr` runs into an overflow of a signed integer. We are not in a safely terminating execution : it “terminates” but not “safely”.

The purpose of RTE is precisely to avoid such issues. RTE generates `assert` annotations inside the source ensuring that any *terminating* execution is *safe*. Typically, we make WP works with RTE using `-wp-rte` option.

```
frama-c-gui -wp -wp-rte incr0.c &
```

In this case, this ensures that for proved programs and for all terminating executions, all assertions are satisfied! Command `frama-c-gui` opens a window where we can inspect RTE annotations and status of ACSL annotations (“proved” in case of green bullet, “unproved” in case of orange bullet, “proved with assuming unproved goals” in case of bi-color bullet). A frame on the left of the window allows to navigate inside the sources. If we click on “`incr`” function, we get the transformed ACSL/C code on which WP has computed goals :

```
/*@ ensures \result == \old(*x) && *\old(x) == \result+1;
    assigns *x; */
int incr(int *x)
{
  int tmp;
  { /* sequence */
    /*@ assert rte: mem_access: \valid_read(x); */
    tmp = *x;
    /*@ assert rte: mem_access: \valid(x); */
```

```
10      /*@ assert rte: signed_overflow: *x+1 <= 2147483647; */
11      /*@ assert rte: mem_access: \valid_read(x); */
12      (*x) ++;
13      ;
14    }
15    return tmp;
16  }
```

Let us remark Frama-C has "simplified" the C code in order to remove the side-effects from the `return` instruction. It has also inserted RTE annotations checking that memory accesses (read with `\valid_read` or read+write with `\valid`) to pointer `x` are valid and that addition of operation `++` does not overflow. Assertions at lines 7, 9, 10 are in orange : they are unproved. Assertion at line 11 is bi-color because it is a consequence of assertion at line 7.

In order to have a "`incr`" function that satisfies its specification together with the RTE assertions, we restrict the specification as follows :

```
/*@ requires \valid(x) && *x < MAXINT;
  @ assigns *x;
  @ ensures \result == \old(*x) && *x == \result+1; */
```

Now, WP (with RTE) proves this, but in the `main` function, line 5 has now an orange bullet, because the precondition of `incr` is unproved. This is expected, because this precondition is not satisfied (and the whole execution is unsafe).
At last, let us consider an alternative `main` function, relying on an external `any` function :

```
1   /* returns an integer from standard input */
2   extern int any();
3
4   int main() {
5     int x = any();
6     int r;
7     if (x == MAXINT) return -1;
8     r=incr(&x);
9     /*@ assert x > r; */
10    if (x <= r) {
11      /*@ assert 0==1; */
12      return 1/(r+1-x);
13    }
14    return x;
15  }
```

Here, running `frama-c -wp -wp-rte incr1.c` outputs :

```
[wp] 14 goals scheduled
[wp] Proved goals:   14 / 14
     Qed:            12  (4ms-8ms)
     Alt-Ergo:        2  (16ms-20ms) (12)
```

In particular, unsatisfiable assertion at line 11 is proved because it is on a *unreachable* control point (i.e. in dead code).

Summary of function verification by WP**.** The specification of a given function f in ACSL defines some precondition P (clause `requires`) and some postcondition Q (clauses `assigns` & `ensures`) on the memory state. When using WP together with RTE, the meaning of such a specification is "*if the initial state satisfies P and if the function terminates then Q is satisfied on the final state and the function has no runtime errors.*" After running RTE to insert runtime error annotations, WP performs the following computations :

postcondition propagation for each call to f, propagate the instance of Q on the final state of the call as an assumption for checking the code following this call ;

postcondition checking propagate P as an assumption for checking the body of f ; check assertions in the body of f (under propagated assumptions) ; check that any final state of f satisfies Q (under propagated assumptions) ;

precondition checking for each call to f, check that the initial state at the call satisfies P (under propagated assumptions).

Let us remark that postcondition propagations and precondition checks also apply to recursive calls (even in mutually recursive functions). We illustrate this point in the next section.

3 Specifying loops for WP with ACSL

Verification of loops with WP is very similar to verification of (recursive) functions. Actually, any loop

```
while (E) { S }
```

is verified like if the following local function (accessing to all variables inside E and S) were defined

```
void while_simu() {
  if (E) {
    S
    while_simu();
  }
}
```

and like if the loop itself were replaced by the call

```
while_simu();
```

Hence, the user is assumed to provide a specification of the loop like

```
/*@ loop invariant I;
  @ loop assigns A; */
while (E) { S }
```

which can be understood using the code transformation below (in this explanation, E is assumed to be without side-effects for the sake of simplicity) :

```
/*@ requires I
  @ assigns A;
  @ ensures I && !E; */
void while_simu() {
  if (E) {
    S
    while_simu();
  }
}
```

In other words, invariant I is a predicate on memory states and computations made by WP with this predicate are the following :

final invariant propagation the code following the loop is checked for any final state satisfying I and leading E to be false (propagation of `while_simu` postcondition after the initial call) ;

invariant preservation for any “intermediate” state satisfying I and leading E to be true, then

- check assertions of S ;
- check I after execution of S (instance of `while_simu` precondition in the recursive call) ;

— check that any location not in A is unchanged after execution of S (partial verification of `while_simu` postcondition) ;

invariant initialization check I is in the initial state of the loop (instance of `while_simu` precondition at the initial call).

Here, let us remark that the above verifications suffices to establish the postcondition of `while_simu` for both branches of the `if`. When E is false, this is immediate from the precondition. When E is true, it trivially derives from the postcondition propagation of the inner recursive call. This illustrates that WP verifications on a recursive function correspond to an inductive proof.

For example, WP (with RTE) succeeds to prove all goals for the code below. In particular, the invariant is sufficient to prove the final assertion.

```
int x = any();
if (x >= 20) { x=0; }
/*@ loop invariant x <= 20;
  @ loop assigns x; */
while (x <= 19) {
  x++;
}
/*@ assert x==20; */
```

Let us remark that `\old` is syntactically forbidden inside loop invariants. Actually, ACSL introduces a more general construct "`\at(`e`,`L`)`" where L is a C label or a predefined label of ACSL like `Pre` (a label corresponding to the initial state of the surrounding function). Hence, in `ensures` clause, `\old(`e`)` is only a macro for `\at(`e`,Pre)`. At this point, we reach the limit of the "`while_simu`" explanation, since "`\at(`e`,Pre)`" is authorized in clause `loop invariant` and it would not have the same meaning in `ensures` clause of `while_simu` function. This construct allows the following simple proof for WP (with RTE)

```
/*@ assigns \nothing;
  @ ensures \result >= x; */
int max20(int x) {

  /*@ loop invariant \at(x,Pre) <= x;
    @ loop assigns x; */
  while (x <= 19) {
    x++;
  }

  return x;
}
```

Here, note that this function does actually no observable assignment, because assignments on parameter `x` are not observable after the call (C semantics of parameter passing).

4 Links to get more details

Weakest-precondition calculus has been invented by Dijkstra in the sixties (it is worth reading its seminal paper "Guarded commands, nondeterminacy and formal derivation of programs" of 1975). Actually, the WP plugin implements a *weakest-liberal-precondition* calculus that does not ensure termination of programs, on the contrary to the original calculus of Dijkstra (but C is a much more complex programming language than the one originally addressed by Dijkstra). A naive algorithm computing weakest-preconditions is given in

http://en.wikipedia.org/wiki/Predicate_transformer_semantics

FRAMA-C's official webpage is :

https://frama-c.com

Some help about ACSL can be found on the official tutorial

https://allan-blanchard.fr/frama-c-wp-tutorial.html

or in WP's official user manual :

https://frama-c.com/download/frama-c-wp-manual.pdf

The ACSL specification language is very rich ; you may see its definition at

http://frama-c.com/download/acsl.pdf