Deductive Verification of C Programs with Frama-C

Frédéric Loulergue

Update of a tutorial designed by

Allan Blanchard, Nikolai Kosmatov, Frédéric Loulergue

with some slides authored by Julien Signoles

Email: frederic.loulergue@univ-orleans.fr

iFM 2024 — Manchester, UK, November 14, 2024

Université d'ORLÉANS

Outline

An overview of Frama-C

Frama-C Open-Source Distribution

Framework for Analysis of source code written in ISO 99 C

[Kirchner et al, FAC'15 and CACM Aug. 2021]

- analysis of C code extended with ACSL annotations
- ACSL Specification Language
  - *langua franca* of Frama-C analyzers
- mostly open-source (LGPL 2.1)

http://frama-c.com

- also proprietary extensions and distributions
- targets both academic and industrial usage

Frama-C Historical Context

- 90's: CAVEAT, Hoare logic-based tool for C code at CEA
- 2000's: CAVEAT used by Airbus during certification process of the A380
  (DO-178 level A qualification)

Frama-C Historical Context

- 90's: CAVEAT, Hoare logic-based tool for C code 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)

Frama-C Historical Context

- 90's: CAVEAT, Hoare logic-based tool for C code 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)
- 2004: start of Frama-C project as a successor to CAVEAT and Caduceus
- 2008: First public release of Frama-C (Hydrogen)

Frama-C Historical Context

- 90's: CAVEAT, Hoare logic-based tool for C code 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)
- 2004: start of Frama-C project as a successor to CAVEAT and Caduceus
- 2008: First public release of Frama-C (Hydrogen)
- 2012: WP: Weakest-precondition based plugin
- 2012: E-ACSL: Runtime Verification plugin
- 2013: CEA Spin-off TrustInSoft
- 2016: Eva: Evolved Value Analysis
- Today: Frama-C version 29.0 (copper)

Example: a C Program Annotated in ACSL

```
/*@ requires n>=0 && \valid(t+(0..n-1));
    assigns \nothing;
    ensures \result != 0 <==> (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros (int t[], int n) {
  int k;
  /*@ loop invariant 0 <= k <= n;
      loop invariant \forall integer j; 0<=j<k ==> t[j]==0;
      loop assigns k;
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

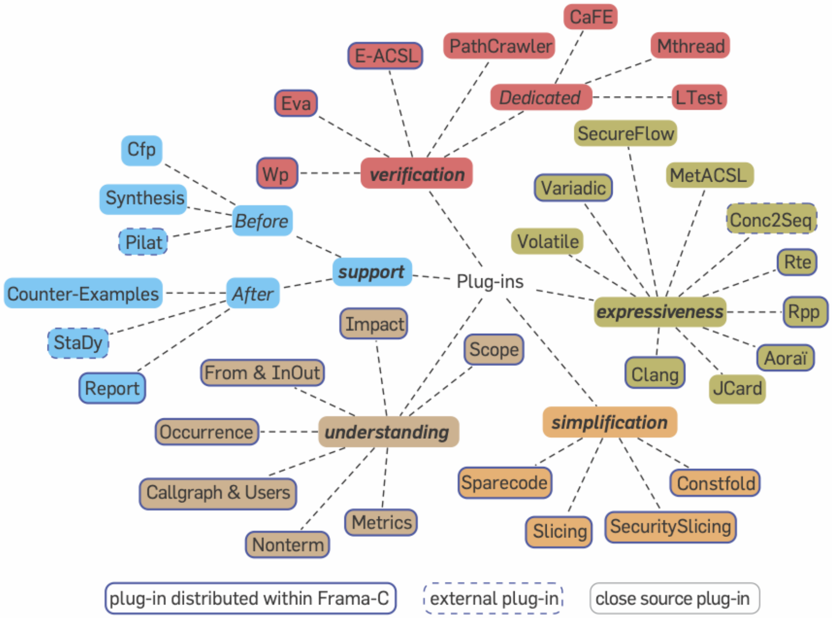
Can be proven
with Frama-C/WP

Frama-C, a Collection of Tools

Several tools inside a single platform

- plugin architecture like in Eclipse
- tools provided as plugins
  - over 20 plugins in the open-source distribution
  - close-source plugins, either at CEA (about 20) or outside
- a common kernel
  - provides a uniform setting
  - provides general services
  - synthesizes useful information

Plugin Gallery (CACM 2021)



Frama-C, a Development Platform

- mostly developed in OCaml
  (≈ 330 kloc in the open-source distribution)
- initially based on Cil [Necula et al, CC'02]
- library dedicated to analysis of C code

development of plugins by third party

- dedicated plugins for specific task (verifying your coding rules)
- dedicated plugins for fine-grained parameterization
- extensions of existing analysers

Focus of this Tutorial: Deductive Verification

Objectives of deductive verification

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

Plugin for deductive verification

- WP
- Related documentation:
  - WP User manual
  - ACSL language reference
  - ACSL language implementation

Overview of ACSL and WP

ACSL: ANSI/ISO C Specification Language

Presentation

- Based on the notion of contract, like in Eiffel, JML
- Allows users to specify functional properties of programs
- Allows communication between various plugins
- Independent from a particular analysis
- Manual at `http://frama-c.com/acsl`

Basic Components

- Typed 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**, **\separated**, **\block_length**, ...

WP plugin

- Hoare-logic based plugin, developed at CEA List
- Proof of semantic properties of the program
- Modular verification (function by function)
- Input: a program and its specification in ACSL
- WP generates verification conditions (VCs)
- Relies on Automatic Theorem Provers to discharge the VCs
  - Alt-Ergo, Z3, CVC3, CVC4, Yices, Simplify . . .
- If all VCs are proved, the program respects the given specification
  - Does it mean that the program is correct?

WP plugin

- Hoare-logic based plugin, developed at CEA List
- Proof of semantic properties of the program
- Modular verification (function by function)
- Input: a program and its specification in ACSL
- WP generates verification conditions (VCs)
- Relies on Automatic Theorem Provers to discharge the VCs
    - Alt-Ergo, Z3, CVC3, CVC4, Yices, Simplify ...
- If all VCs are proved, the program respects the given specification
    - Does it mean that the program is correct?
    - NO! If the specification is wrong, the program can be wrong!

Function contracts

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 the caller)
  - Postcondition must be true on exit (ensured by the function)
- Nothing is guaranteed when the precondition is not satisfied
- Termination may be guaranteed or not (total or partial correctness)

Primary role of contracts

- Must reflect the informal specification
- Should not be modified just to suit the verification tasks

Example 1

Specify and prove the following program:

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

Try to prove with Frama-C/WP using one of the basic command:

- CLI: `frama-c -wp file.c`
- GUI: `frama-c-gui -wp file.c`
  - does not work on MacOS, Windows
  - bugs on Linux
- New GUI: `ivette -wp file.c`
  - support for WP is not as complete as the former GUI

Example 1 (Continued)

The basic proof succeeds for the following program:

```
/*@ ensures (x >= 0 ==> \result == x) &&
        (x < 0 ==> \result == -x);
*/
int abs ( int x ) {
   if ( x >=0 )
     return x ;
   return -x ;
}
```

Demo

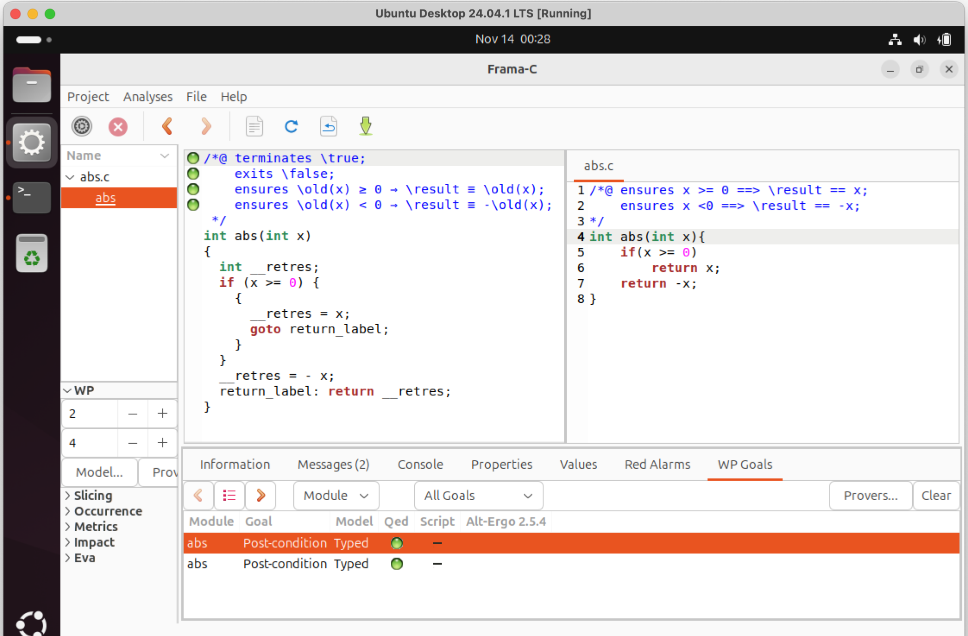
Example 1: CLI

examples_and_exercices — -bash — 70×16

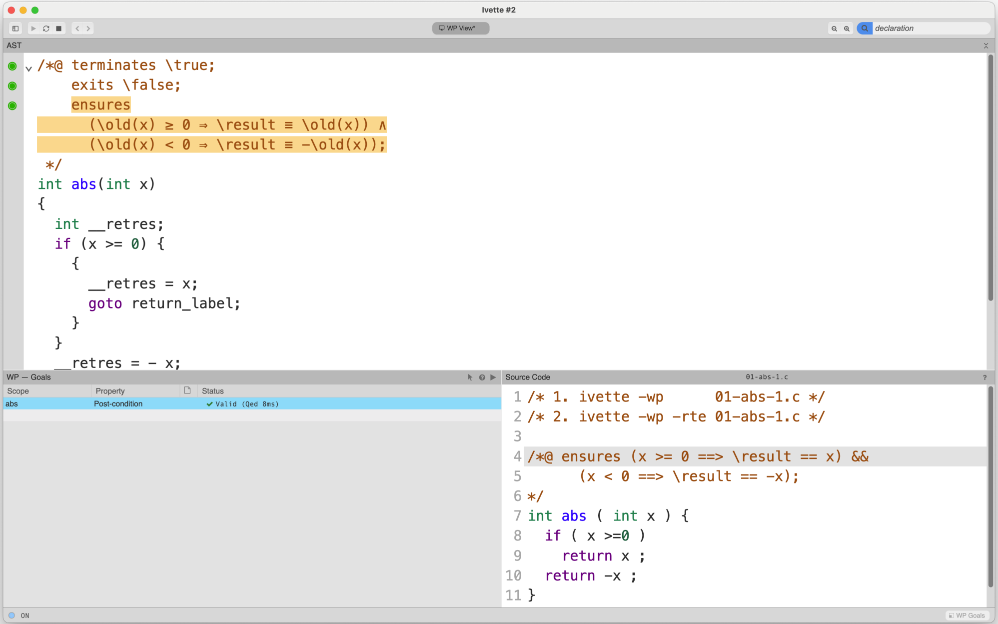
.../2024_ifm/examples_and_exercices — -bash | ...— docker pull fredblgr/framac-novnc:2022 | ...wnloads/frama-c-29.0-Copper — -bash

```
[MacBookPro13-FL|examples_and_exercices]frama-c -wp 01-abs-1.c
[kernel] Parsing 01-abs-1.c (with preprocessing)
[wp] Warning: Missing RTE guards
[wp] 1 goal scheduled
[wp] Proved goals:    3 / 3
  Terminating:     1
  Unreachable:     1
  Qed:             1
[wp:pedantic-assigns] 01-abs-1.c:7: Warning:
  No 'assigns' specification for function 'abs'.
  Callers assumptions might be imprecise.
[MacBookPro13-FL|examples_and_exercices]
```

Example 1: GUI



Example 1: New GUI



Example 1 (Continued)

The basic proof succeeds for the following program:

```
/*@ ensures (x >= 0 ==> \result == x) &&
      (x < 0 ==> \result == -x);
*/
int abs ( int x ) {
  if ( x >=0 )
    return x ;
  return -x ;
}
```

There is a warning: Missing RTE guards

Safety warnings: arithmetic overflows

Absence of arithmetic overflows can be important to check

- A sad example: crash of Ariane 5 in 1996

WP can automatically check the absence of runtime errors

- To add RTE guards:
  - right-click on the function name $\hookrightarrow$ “Populate WP RTE guards”
  - option `-rte`
- It generates VCs to ensure that runtime errors do not occur
  - in particular, arithmetic operations do not overflow
- If not proved, an error may occur.

In Example 1: `/*@ assert rte: signed_overflow: -2147483647 <= x; */` before the expression $-x$

Example 1 (Continued): Solution

Run WP: `ivette -wp -rte 01-abs-2.c`

This completely specified program is proved:

```
#include<limits.h>
/*@ requires x > INT_MIN;
    ensures x >= 0 ==> \result == x;
    ensures x < 0  ==> \result == -x;
    assigns \nothing;
*/
int abs ( int x ) {
  if ( x >=0 )
    return x ;
  return -x ;
}
```

Example 1 (Continued): Problem

Run WP: `ivette -wp -rte 01-abs-3.c`. All VC are proved! What's the problem?

```
#include "limits.h"
/*@ requires x < INT_MIN;
    ensures (x >= 0 ==> \result == -x);
    ensures (x < 0 ==> \result == x);
    assigns \nothing;
*/
int abs ( int x ) {
  if ( x >=0 )
    return x ;
  return -x ;
}
```

- `frama-c -wp -rte -wp-smoke-tests 01-abs-3.c`
- Smoke tests fail

Example 2

Specify and prove the following program:

```
// returns the maximum of a and b
int max ( int a, int b ) {
  if ( a > b )
    return a ;
  return b ;
}
```

Example 2 (Continued) - Find the error

Run WP: `ivette -wp -rte 02-max-1.c`

The following program is proved. Do you see any error?

```
/*@ ensures \result >= a && \result >= b;
*/
int max ( int a, int b ) {
  if ( a >= b )
    return a ;
  return b ;
}
```

Example 2 (Continued) - A wrong version

Run WP: `ivette -wp -rte 02-max-2.c`

This is a wrong implementation that is also proved. Why?

```
#include<limits.h>
/*@ ensures \result >= a && \result >= b; */
int max ( int a, int b ) {
  return INT_MAX ;
}
```

Example 2 (Continued) - A wrong version

Run WP: `ivette -wp -rte 02-max-2.c`

This is a wrong implementation that is also proved. Why?

```
#include<limits.h>
/*@ ensures \result >= a && \result >= b; */
int max ( int a, int b ) {
  return INT_MAX ;
}
```

- Our specification is incomplete
- Should say that the returned value is one of the arguments

Example 2 (Continued) - Another issue

The following program is proved. Do you see any issue?

```
/*@ ensures \result >= a && \result >= b;
    ensures \result == a || \result == b ;
*/
int max ( int a, int b ) {
  if ( a >= b )
    return a ;
  return b ;
}
```

Example 2 (Continued) - Another issue

With this specification, we cannot prove the following program. Why?

```
/*@ ensures \result >= a && \result >= b ;
    ensures \result == a || \result == b ; */
int max(int a, int b);

extern int x ;

int main(){
  x = 3;
  int r = max(4,2);
  //@ assert x == 3 ;
}
```

Example 2 (Continued) - Another issue

With this specification, we cannot prove the following program. Why?

```
/*@ ensures \result >= a && \result >= b ;
    ensures \result  == a || \result  == b ; */
int max(int a, int b);

extern int x ;

int main(){
  x = 3;
  int r = max(4,2);
  //@ assert x == 3 ;
}
```

- Again, our specification is incomplete
- Should say that max does not modify any memory location

Assigns clause

The clause **assigns** v1, v2, ... , vN;

- Part of the postcondition
- Specifies which (non local) variables can be modified by the function
- No need to specify local variable modifications in the postcondition
  - a function is allowed to change local variables
  - a postcondition cannot talk about them anyway, they do not exist after the function call
- If nothing can be modified, specify **assigns \nothing**

Assigns clause

The clause **assigns** v1, v2, ... , vN;

- Part of the postcondition
- Specifies which (non local) variables can be modified by the function
- No need to specify local variable modifications in the postcondition
  - a function is allowed to change local variables
  - a postcondition cannot talk about them anyway, they do not exist after the function call
- If nothing can be modified, specify **assigns \nothing**
- Avoids to state for all unchanged global variables v:
  ensures \old(v) == v;
- Avoids to forget one of them: explicit permission is required

Example 2 (Continued) - Solution

Run WP: `ivette -wp -rte 02-max-4.c`

This completely specified program is proved:

```
/*@ ensures \result >= a && \result >= b;
    ensures \result == a || \result == b;
    assigns \nothing;
*/
int max ( int a, int b ) {
  if ( a >= b )
    return a ;
  return b ;
}
```

Example 3

Specify and prove the following program:

```
// returns the maximum of *p and *q
int max_ptr ( int *p, int *q ) {
  if ( *p >= *q )
    return *p ;
  return *q ;
}
```

Example 3 (Continued) - A proof failure

Run WP: `ivette -wp -rte 03-max_ptr-1.c`

Explain the proof failure for the program:

```
/*@ ensures \result >= *p && \result >= *q;
    ensures \result == *p || \result == *q;
*/
int max_ptr ( int *p, int *q ) {
  if ( *p >= *q )
    return *p ;
  return *q ;
}
```

Example 3 (Continued) - A proof failure

Run WP: `ivette -wp -rte 03-max_ptr-1.c`

Explain the proof failure for the program:

```
/*@ ensures \result >= *p && \result >= *q;
    ensures \result == *p || \result == *q;
*/
int max_ptr ( int *p, int *q ) {
  if ( *p >= *q )
    return *p ;
  return *q ;
}
```

- Nothing ensures that pointers p, q are valid
- It must be ensured either by the function, or by its precondition

Safety warnings: invalid memory accesses

An invalid pointer or array access may result in a segmentation fault or memory corruption.

- WP can automatically generate VCs to check memory access validity
  - use the command `ivette -wp -rte file.c`
- They ensure that each pointer (array) access has a valid offset (index)
- If the function assumes that an input pointer is valid, it must be stated in its precondition, e.g.
  - **\valid**(p) for one pointer p
  - **\valid**(p+0..2) for a range of offsets p, p+1, p+2

Example 3 (Continued) - Another issue

Run WP: `ivette -wp -rte 03-max_ptr-2.c`

The following program is proved. Do you see any issue?

```
/*@ requires \valid(p) && \valid(q);
    ensures \result >= *p && \result >= *q;
    ensures \result == *p || \result == *q;
*/
int max_ptr ( int *p, int *q ) {
  if ( *p >= *q )
    return *p ;
  return *q ;
}
```

Example 3 (Continued) - A wrong version

Run WP: `ivette -wp -rte 03-max_ptr-3.c`

This is a wrong implementation that is also proved. Why?

```
/*@ requires \valid(p) && \valid(q);
    ensures \result >= *p && \result >= *q;
    ensures \result == *p || \result == *q;
*/
int max_ptr ( int *p, int *q ) {
  *p = 0;
  *q = 0;
  return 0 ;
}
```

Example 3 (Continued) - A wrong version

Run WP: `ivette -wp -rte 03-max_ptr-3.c`

This is a wrong implementation that is also proved. Why?

```c
/*@ requires \valid(p) && \valid(q);
    ensures \result >= *p && \result >= *q;
    ensures \result == *p || \result == *q;
*/
int max_ptr ( int *p, int *q ) {
  *p = 0;
  *q = 0;
  return 0 ;
}
```

- Our specification is incomplete
- Should say that the function cannot modify *p and *q

Example 3 (Continued) - Solution

Run WP: `ivette -wp -rte 03-max_ptr-4.c`

This completely specified program is proved:

```
/*@ requires \valid(p) && \valid(q);
    ensures \result >= *p && \result >= *q;
    ensures \result == *p || \result == *q;
    assigns \nothing;
*/
int max_ptr ( int *p, int *q ) {
  if ( *p >= *q )
    return *p ;
  return *q ;
}
```

The wrong version is not proved wrt. this specification.

Example 4

Specify and prove the following program (file `04-incr_a_by_b-0.c`):

```c
void incr_a_by_b(int* a, int* b){
  *a += *b;
}
```

Example 4 - Explain the proof failure

```c
#include <limits.h>

/*@
  requires INT_MIN <= *a + *b <= INT_MAX ;
  requires \valid(a) && \valid(b);
  assigns  *a;
  ensures  *a == \old(*a)+ *b; // CANNOT BE PROVED
*/
void incr_a_by_b (int* a, int* b){
  *a += *b;
}
```

Example 4 - Explain the proof failure

```c
#include <limits.h>

/*@
  requires INT_MIN <= *a + *b <= INT_MAX ;
  requires \valid(a) && \valid(b);
  assigns  *a;
  ensures  *a == \old(*a)+ *b; // CANNOT BE PROVED
*/
void incr_a_by_b (int* a, int* b){
  *a += *b;
}
```

- Our specification is incomplete
- Should say that a and b point to separated memory locations

Example 4 - Solution

Run WP: `ivette -wp -rte 04-incr_a_by_b-1.c`

This is the completely specified program:

```
#include <limits.h>


/*@
  requires INT_MIN <= *a + *b <= INT_MAX ;
  requires \valid(a) && \valid(b);
  requires \separated(a, b);
  assigns *a;
  ensures *a == \old(*a)+ *b;
*/
void incr_a_by_b (int* a, int* b){
  *a += *b;
}
```

Behaviors

Specification by cases

- Global precondition (**requires**) applies to all cases
- Global postcondition (**ensures**, **assigns**) applies to all cases
- Behaviors define contracts (refine global contract) in particular cases
- For each case (each **behavior**)
  - the subdomain is defined by **assumes** clause
  - the behavior's precondition is defined by **requires** clauses
    - it is supposed to be true whenever **assumes** condition is true
  - 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

Example 5

Specify using behaviors and prove the function `abs` (file `05-abs-0.c`):

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

Example 5 (Continued) - Solution

Run WP: `ivette -wp -rte 05-abs-1.c`

```c
#include<limits.h>
/*@ requires x > INT_MIN;
    assigns \nothing;
    behavior pos:
      assumes x >= 0;
      ensures \result == x;
    behavior neg:
      assumes x < 0;
      ensures \result == -x;
    complete behaviors;
     disjoint behaviors;
*/
int abs ( int x ) {
  if ( x >=0 )
    return x ;
  return -x ;
}
```

```c
#include<limits.h>
/*@ requires x > INT_MIN;
    assigns \nothing;
    behavior pos:
      assumes x >= 0;
      ensures \result == x;
    behavior neg:
      assumes x <= 0;
      ensures \result == -x;
    complete behaviors;
*/
int abs ( int x ) {
  if ( x >=0 )
    return x ;
  return -x ;
}
```

Contracts and function calls

```
// Pre_f assumed
f(<args>){
  code1;
// Pre_g to be proved
  g(<args>);
// Post_g assumed
  code2;
}
// Post_f to be proved
```

Pre/post of the caller and of the callee have dual roles in the caller's proof

- Pre of the caller is assumed, Post of the caller must be ensured
- Pre of the callee must be ensured, Post of the callee is assumed

Example 6

Specify and prove the function max_abs (file `06-max_abs-0.c`):

```
int abs ( int x );
int max ( int x, int y );

// returns maximum of absolute values of x and y
int max_abs( int x, int y ) {
  x=abs(x);
  y=abs(y);
  return max(x,y);
}
```

Example 6 (Continued) - Explain the proof failure

Run WP: `ivette -wp -rte 06-max_abs-1.c`

```
#include<limits.h>
/*@ requires x > INT_MIN;
    ensures (x >= 0 ==> \result == x) && (x < 0 ==> \result == -x);
    assigns \nothing; */
int abs ( int x );

/*@ ensures \result >= x && \result >= y;
    ensures \result == x || \result == y;
    assigns \nothing; */
int max ( int x, int y );

/*@ ensures \result >= x && \result >= -x && \result >= y && \result >= -y;
    ensures \result == x || \result == -x || \result == y || \result == -y;
    assigns \nothing; */
int max_abs( int x, int y ) {
  x=abs(x);
  y=abs(y);
  return max(x,y);
}
```

Example 6 (Continued) - Explain the proof failure

Run WP: `ivette -wp -rte 06-max_abs-2.c`

```
#include<limits.h>
/*@ requires x > INT_MIN;
    ensures (x >= 0 ==> \result == x) && (x < 0 ==> \result == -x);
    assigns \nothing; */
int abs ( int x );

/*@ ensures \result >= x && \result >= y;
    assigns \nothing; */
int max ( int x, int y );

/*@ requires x > INT_MIN;
    requires y > INT_MIN;
    ensures \result >= x && \result >= -x && \result >= y && \result >= -y;
    ensures \result == x || \result == -x || \result == y || \result == -y;
    assigns \nothing; */
int max_abs( int x, int y ) {
  x=abs(x);
  y=abs(y);
  return max(x,y);
}
```

Example 6 (Continued) - Solution

Run WP: `ivette -wp -rte 06-max_abs-3.c`

```c
#include<limits.h>
/*@ requires x > INT_MIN;
    ensures (x >= 0 ==> \result == x) && (x < 0 ==> \result == -x);
    assigns \nothing; */
int abs ( int x );

/*@ ensures \result >= x && \result >= y;
    ensures \result == x || \result == y;
    assigns \nothing; */
int max ( int x, int y );

/*@ requires x > INT_MIN && y > INT_MIN;
    ensures \result >= x && \result >= -x && \result >= y && \result >= -y;
    ensures \result == x || \result == -x || \result == y || \result == -y;
    assigns \nothing; */
int max_abs( int x, int y ) {
  x=abs(x);
  y=abs(y);
  return max(x,y);
}
```

Programs with loops

Loops and automatic proof

- What is the issue with loops? Unknown, variable number of iterations
- The only possible way to handle loops: proof by induction
- Induction needs a suitable inductive property, that is proved to be
    - satisfied just before the loop, and
    - satisfied after $k+1$ iterations whenever it is satisfied after $k \geq 0$ iterations
- Such inductive property is called loop invariant
- The verification conditions for a loop invariant include two parts
    - loop invariant initially holds
    - loop invariant is preserved by any iteration

Loop invariants - some hints (*)

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

- identify variables modified in the loop
  - variable number of iterations prevents from deducing their values (relationships with other variables)
  - 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

Some experience may be necessary to find appropriate loop invariants

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 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

Example 7

Specify and prove the function reset_array (file `07-reset_array-0.c`):

```
// writes 0 in each cell of the
// array a of len integers
void reset_array(int* a, int len){
  for(int i = 0 ; i < len ; ++i){
    a[i] = 0 ;
  }
}
```

Example 7 (Continued) - Solution

Run WP: `ivette -wp -rte 07-reset_array-1.c`

```
/*@ requires 0 <= len;
  requires \valid(a + (0 .. len-1));
  assigns a[0 .. len-1];
  ensures \forall integer i ; 0 <= i < len ==> a[i] == 0;
*/
void reset_array (int* a, int len){
  /*@
    loop invariant 0 <= i <= len ;
    loop invariant \forall integer j; 0 <= j < i ==> a[j] == 0 ;
    loop assigns i, a[0 .. len-1];
    loop variant len - i ;
  */
  for(int i = 0 ; i < len ; ++i){
    a[i] = 0 ;
  }
}
```

Example 8

Specify and prove the function binary_search :

```
/* takes as input a sorted array a, its length, and a value key to search,
   returns the index of a cell which contains 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;
}
```

Example 8 (Continued) - Solution (1/2)

```
/*@ predicate sorted{L}(int* a, int length) =
      \forall integer i,j; 0<=i<=j<length ==> a[i]<=a[j];
*/
/*@ requires \valid(a+(0..length-1));
    requires sorted(a,length);
    requires length >=0;

    assigns \nothing;

    behavior found:
      assumes \exists integer i; 0<=i<length && a[i] == key;
      ensures 0<=\result<length && a[\result] == key;

    behavior not_found:
      assumes \forall integer i; 0<=i<length ==> a[i] != key;
      ensures \result == -1;

    complete behaviors;
     disjoint behaviors;
*/
```

Example 8 (Continued) - Solution (2/2)

```
int binary_search(int* a, int length, int key) {
  int low = 0, high = length - 1;
  /*@ 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) {
    int mid = low+(high-low)/2;
    if (a[mid] == key) return mid;
    if (a[mid] < key) { low = mid+1; }
    else { high = mid - 1; }
  }
  return -1;
}
```

An application to Contiki

A lightweight OS for IoT

Contiki is a lightweight operating system for IoT

It provides a lot of features:

- (rudimentary) memory and process management
- networking stack and cryptographic functions
- ...

Typical hardware platform:

- 8, 16, or 32-bit MCU (little or big-endian),
- low-power radio, some sensors and actuators, ...

Note for security: there is *no* memory protection unit.

Overview of the `memb` Module

- No dynamic allocation in Contiki
  - to avoid fragmentation of memory in long-lasting systems
- Memory is pre-allocated (in arrays of blocks) and attributed on demand
- The management of such blocks is realized by the `memb` module

The `memb` module API allows the user to

- initialize a `memb` store (i.e. pre-allocate an array of blocks),
- allocate or free a block,
- check if a pointer refers to a block inside the store
- count the number of allocated blocks

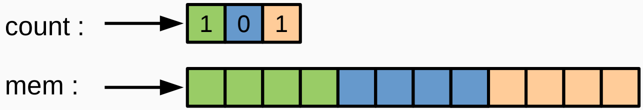
memb Data structure

```
struct memb {
  unsigned short size;
  unsigned short num;
  char *count;
  void *mem;
};
```

For example:

size = 4

num = 3



memb Data structure: dedicated predicate

```
predicate valid_memb(struct memb *m) =
  \valid(m)
  && \valid(m->count + (0 .. (m->num - 1)))
  && \valid((char*) m->mem + (0 .. (m->size * m->num - 1)))
  && m->size > 0
  && m->size * m->num <= INT_MAX
  && \separated(m->count + (0 .. (m->num - 1)),
                (char*) m->mem + (0 .. m->size * m->num - 1));
```

```
/*@ // Converting from pointer to index and backwards.
  logic integer _memb_index(struct memb *m, void *ptr) =
    (ptr - m->mem) / m->size;
  logic void * _memb_ptr(struct memb *m, integer index) =
    (void*) ((char*) m->mem + index * m->size);
  // Counting free elements.
  logic integer _memb_numfree(struct memb *m) = count(0, m->count, 0, m->num);
  // Helper predicates. For readability.
  predicate _memb_has(struct memb *m, void *ptr) =
    \exists integer i; 0 <= i < m->num && ptr == _memb_ptr(m, i);
  predicate _memb_allocated(struct memb *m, void *ptr) =
    _memb_has(m, ptr) && m->count[_memb_index(m, ptr)] != 0;
  predicate _memb_empty(struct memb *m) =
    \forall integer i; 0 <= i < m->num ==> m->count[i] == 0;
  predicate _memb_full(struct memb *m) =
    \forall integer i; 0 <= i < m->num ==> m->count[i] != 0;
*/
```

Counting occurrences in an Array

```
axiomatic Count{
  logic integer count{L}(integer e, char *t, integer from, integer to)
    reads t[from .. (to - 1)];
  axiom end_count{L}:
     \forall integer e, char *t, integer from, to;
      from >= to ==> count{L}(e, t, from, to) == 0;
  axiom iter_count_true{L}:
     \forall integer e, char *t, integer from, to;
      (from < to && t[to-1] == e) ==>
      count{L}(e, t, from, to) == count{L}(e, t, from, to-1) + 1;
  axiom iter_count_false{L}:
     \forall integer e, char *t, integer from, to;
      (from < to && t[to-1] != e) ==>
      count{L}(e, t, from, to) == count{L}(e, t, from, to-1);
}
```

```c
char memb_free(struct memb *m, void *ptr)
{
  int i;
  char *ptr2;
  /* Walk through the list of blocks and try to find the block to
     which the pointer "ptr" points to. */
  ptr2 = (char *)m->mem;
  for(i = 0; i < m->num; ++i) {
    if(ptr2 == (char *)ptr) {
      m->count[i] = 0;
      return m->count[i];
    }
    ptr2 += m->size;
  }
  return -1;
}
```

memb deallocation function: contract

```
requires valid_memb(m);
ensures valid_memb(m);
assigns m->count[_memb_index(m, ptr)];
behavior alloc_found :
  assumes _memb_has(m, ptr) && _memb_allocated(m, ptr);
  ensures !_memb_allocated(m, ptr);
  ensures _memb_numfree(m) == \old(_memb_numfree(m)) + 1;
  ensures \result == 0;
behavior already_free :
  assumes _memb_has(m, ptr) && !_memb_allocated(m, ptr);
  ensures !_memb_allocated(m, ptr);
  ensures _memb_numfree(m) == \old(_memb_numfree(m));
  ensures \result == 0;
behavior elem_notfound:
  assumes !_memb_has(m, ptr);
  ensures m->count[_memb_index(m, ptr)] == \old(m->count[_memb_index(m, ptr)]);
  ensures _memb_numfree(m) == \old(_memb_numfree(m));
  ensures \result == -1;
complete behaviors;
disjoint behaviors;
```

- There is a loop: loop invariant and variant
- Additional annotations needed, for e.g.:

```
/*@ assert whole_cut_before: count(0, m->count, 0, m->num) ==
        count(0, m->count, 0, i) + count(0, m->count, i, m->num); */
/*@ assert part_cut_before : count(0,m->count,i,m->num) ==
        count(0, m->count, i, i+1) + count(0 ,m->count, i+1, m->num); */
```

- As well as lemmas:

```
lemma count_split{L}:
  \forall integer e, char *t, integer from, cut, to;
   from <= cut <= to ==>
   count{L}(e,t,from,to) == count{L}(e,t,from,cut)+count{L}(e,t,cut,to);
predicate same_elems{L1,L2}(char *t, integer from, integer to) =
  \forall integer j; from <= j < to ==> \at(t[j], L1) == \at(t[j], L2);
lemma same_elems_means_same_count{L1, L2}:
  \forall integer e, char *t, integer from, to;
   same_elems{L1,L2}(t,from,to) ==>
   count{L1}(e, t, from, to) == count{L2}(e, t, from, to);
```

memb deallocation function: verification

- We also need a lemma on arithmetic:

```
/*@ lemma mult_simplification:
      \forall integer a, b; a >= 0 ==> b > 0 ==> (a * b) / b == a;
*/
```

- **With all that, memb_free is proved correct wrt its specification**
- What about lemmas?

Proving lemmas

- **Often interesting lemmas are not proved by SMT solvers**
- Interactive theorem proving: a 526 lines long file, with at the end

```
Theorem wp_goal :
  ∀ (t:addr → Numbers.BinNums.Z) (i:Numbers.BinNums.Z)
    (i1:Numbers.BinNums.Z) (i2:Numbers.BinNums.Z) (a:addr)
    (i3:Numbers.BinNums.Z), (i <= i3)%Z → (i2 <= i)%Z →
  (((L_count t i1 a i i3) + (L_count t i1 a i2 i))%Z = (L_count t i1 a i2 i3)).
Proof.
```

- Write ghost “lemma functions” instead:

```
/*@ requires from <= cut <= to;
  @ ensures occ_a(e,t,from,to) == occ_a(e,t,from,cut)+occ_a(e,t,cut,to);
  @ assigns \nothing; */
void occ_a_split (int e, char * t, int from, int cut, int to)
{ /*@ loop invariant cut<=i<=to;
    @ loop invariant occ_a(e,t,from,i) == occ_a(e,t,from,cut)+occ_a(e,t,cut,i);
    @ loop assigns i;
    @ loop variant to - i; */
  for(int i = cut; i<to; i++);
}
```

My proof fails... What to do?

Proof failures

A proof of a VC for some annotation can fail for various reasons:

- incorrect implementation (→ check your code)
- incorrect annotation (→ check your spec)
- missing or erroneous (previous) annotation (→ check your spec)
- insufficient timeout (→ try longer timeout)
- complex property that automatic provers cannot handle.

Analysis of proof failures

When a proof failure is due to the specification, the erroneous annotation may be not obvious to find. For example:

- proof of a "loop invariant preserved" may fail in case of
  - incorrect loop invariant
  - incorrect loop invariant in a previous, or inner, or outer loop
  - missing **assigns** or **loop assigns** clause
  - too weak precondition
  - ...
- proof of a postcondition may fail in case of
  - incorrect loop invariant (too weak, too strong, or inappropriate)
  - missing **assigns** or **loop assigns** clause
  - inappropriate postcondition in a called function
  - too weak precondition
  - ...

Analysis of proof failures (Continued)

- Additional statements (**assert**, **lemma**, ...) may help the prover
  - They can be provable by the same (or another) prover or checked elsewhere
- Separating independent properties (e.g. in separate, non disjoint behaviors) may help
  - The prover may get lost with a bigger set of hypotheses (some of which are irrelevant)

When nothing else helps to finish the proof:

- an interactive proof assistant can be used
- Coq, Isabelle, PVS, are not that scary: we may need only a small portion of the underlying theory
- Alternatively, in most cases lemma functions may be used

Conclusion

Conclusion

We have presented how to:

- formally specify functional properties with ACSL
- prove a programs respects its specification with WP

Much more inside Frama-C including

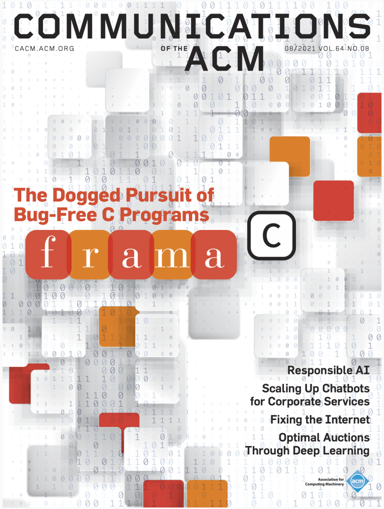
- verifying the absence of runtime errors with Eva
- verifying annotations at runtime or detect runtime errors with E-ACSL

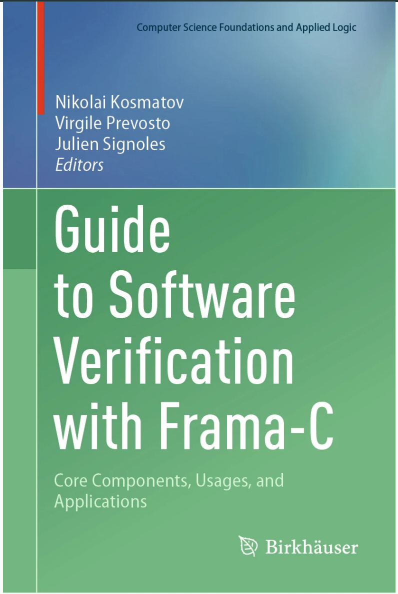
May be used for:

- teaching
- academic prototyping
- industrial applications

http://frama-c.com

Further Reading





Further reading

About the use of WP:

- Introduction to C program proof using Frama-C and its WP plugin
  Allan Blanchard
  `https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf`
- (ACSL by Example
  Jochen Burghardt, Jens Gerlach
  `https://github.com/fraunhoferfokus/acsl-by-example`)

Further reading

Tutorial papers:

- A. Blanchard, N. Kosmatov, and F. Loulergue. A Lesson on Verification of IoT Software with Frama-C (HPCS 2018)
- on deductive verification:
  N. Kosmatov, V. Prevosto, and J. Signoles. A lesson on proof of programs with Frama-C (TAP 2013)
- on runtime verification:
  - N. Kosmatov and J. Signoles. A lesson on runtime assertion checking with Frama-C (RV 2013)
- on test generation:
  N. Kosmatov, N. Williams, B. Botella, M. Roger, and O. Chebaro. A lesson on structural testing with PathCrawler-online.com (TAP 2012)
- on analysis combinations:
  N. Kosmatov and J. Signoles. Frama-C, A collaborative framework for C code verification: Tutorial synopsis (RV 2016)

Further reading

More details on the verification of Contiki:

- on the MEMB module:
  F. Mangano, S. Duquennoy, and N. Kosmatov. A memory allocation module of Contiki formally verified with Frama-C. A case study (CRiSIS 2016)
- on the AES-CCM* module:
  A. Peyrard, S. Duquennoy, N. Kosmatov, and S. Raza. Towards formal verification of Contiki: Analysis of the AES–CCM* modules with Frama-C (RED-IoT 2017)
- on the LIST module:
  - A. Blanchard, N. Kosmatov, and F. Loulergue. Logic against Ghosts: Comparison of two Proof Approaches for a List Module (SAC 2019)
  - A. Blanchard, F. Loulergue and N. Kosmatov. Towards Full Proof Automation in Frama-C using Auto-Active Verification. (NFM 2019)