Deductive Program Verification with Frama-C

Maria João Frade

HASLab - INESC TEC
Departmento de Informática, Universidade do Minho

2018/2019

Roadmap

- **Introduction**
  - Frama-C; WP plugin; ACSL; memory models; annotations; properties; local properties status; runtime errors;
- **Program Specification**
  - function contracts; safety; behaviours; function calls; logical predicates; state labels;
- **Program Verification**
  - loops and proof; loop invariants; termination policy; loop variants; proof failures;
- **Other Features**
  - using axiomatics; algebraic modeling; ghost code;

Frama-C (FRAmework for Modular Analysis of C programs)

- Frama-C is an open-source platform for static analysis of C code.
- Developed at CEA LIST and INRIA Saclay.
- Frama-C 1st release: 2008. Previous: Why+Caduceus (early 2000's), CAVEAT (90's).
- Plugin architecture. Various plugins: value analysis, deductive verification, slicing, dependency analysis, impact analysis, metrics calculation, ...
- Includes ACSL specification language.
- Extensible and collaborative platform.
  - One can add new plugins.
  - Allows collaboration of analyses over the same code.
  - Inter-plugin communication through ACSL formulas.
- http://frama-c.com/

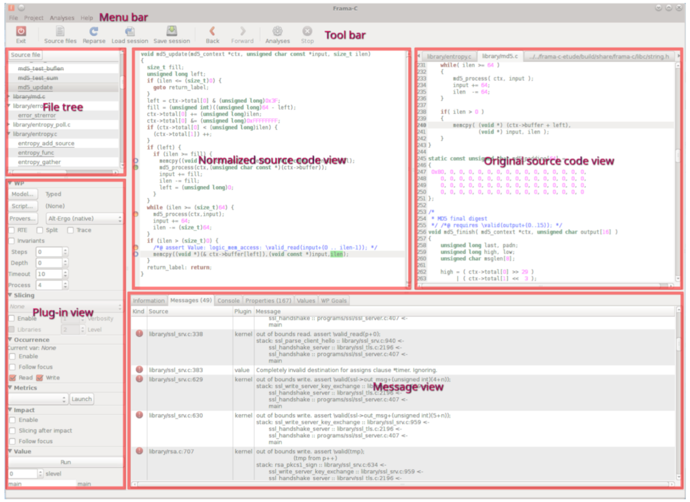
Deductive verification with Frama-C

- Frama-C has two plugins for deductive verification.
  - **Jessie** (developed at INRIA, 2009)
  - **WP** (developed at CEA LIST, 2012)
- Both plugins are based on Hoare logic and weakest precondition calculus.
  - Jessie relies on a separation memory model and operates by compiling to Why.
  - WP focuses on parametrization w.r.t. the memory model (different models are available).
- Both plugins allow to prove that C functions satisfy their specification (expressed in ACSL).
  - Proofs are modular: the specifications of the called functions are used to establish the proof without looking at their code.

Preparing the sources

- For the creation of an analysis project, Frama-C performs several steps for preparing the sources to be analyzed.
  - Pre-processing phase. Frama-C performs some pre-processing of the souces code.
  - Merging phase. Frama-C parses, type-checks and links the code. It also performs these operations for the ACSL annotations.
  - Normalization phase. Frama-C performs a number of local code transformations aiming at making further work easier for the analyzers.
- Analyses usually take place on the normalized version of the source code.
- Normalization gives a program which is *semantically equivalent* to the original one.

frama-c-gui



WP plugin

- Proof of safety and functional properties of C annotated programs.
- Implements an *Weakest Precondition calculus parameterized by a memory model* (to represent pointers and heap values).
- WP operates as follows:
  - receives as input a normalized C program with ACSL annotations;
  - generates the verification conditions (VCs) via WP VCgen;
  - discharges the VCs using external theorem provers via Why3.
- The WP plugin is cooperative, i.e., it allows to combine WP calculus with other techniques available via other plugins.

Memory models

- The essence of a weakest precondition calculus is to translate code annotations into mathematical properties.
- To apply the WP calculus to programs dealing with pointers one has to have a *memory model*.
- A memory model defines a mapping from values inside the C memory heap to mathematical terms.
- The WP has been designed to support different memory models:
  - **Hoare model**. A very efficient model that generates concise proof obligations. It simply maps each C variable to one pure logical variable. However, the heap cannot be represented in this model.
  - **Typed model**. Heap values are stored in several separated global arrays, one for each atomic type and an additional one for memory allocation. Pointer values are translated into an index into these arrays.
  - **Bytes model** (not implemented yet). This is a low-level memory model, where the heap is represented as a wide array of bytes.

Program annotations

- Frama-C supports writing code annotations with the ACSL language.
- The purpose of annotations is to formally specify the properties of C code.
- Annotations can originate from a number of different sources:
  - **the user** who writes his own annotations;
  - **some plugins** may generate code annotations (cf. RTE plugin);
  - **the kernel** of Frama-C, that attempts to generate as precise an annotation as it can, when none is present.

Properties

- A *property* is a logical statement bound to a precise code location.
- A property might originate from an ACSL code annotation or by a plugin-dependent meta-information.

Property validity

Consider a program point i, and call T the set of traces that run through i.

- A logical property P *is valid at* i if it is valid on all $t \in T$.
- Conversely, any trace u that does not validate P, stops at i: properties are *blocking*.

Local property status

- An important part of the interactions between Frama-C components rely on their capacity to emit a judgment on the validity of a property P at program point i.
- In Frama-C nomenclature, this judgment is called a *local property status*, which has two parts.
- The first part of a local status ranges over the following values:
  - True when the property is true for all traces;
  - False when there exists a trace that falsifies the property;
  - Maybe when the emitter e cannot decide the status of P .
- The second part of a local property status, an emitter can add a list of *dependencies*, which is the set of properties whose validity may be necessary to establish the judgment.
  - The dependencies are meant *as a guide* to safety engineers. They are neither correct, nor complete.

An example: **abs**

A specification for the abs function could be (**abs_init.c**):

```
/*@ ensures (x >= 0 ==> \result == x) &&
            (x < 0 ==> \result == -x);
    assigns \nothing;
*/
int abs (int x) {
  if (x >= 0) return x ;
  return -x ;
}
```

We can run Frama-C to determine if the implementation is correct against the specification using

- the command line interface of Frama-C, or
- the graphical user interface of Frama-C.

Invoking WP

```
$ frama-c -wp abs_init.c
    [kernel] Parsing abs_init.c (with preprocessing)
    [wp] Warning: Missing RTE guards
    [wp] 3 goals scheduled
    [wp] Proved goals:    3 / 3
       Qed:            3  (4ms)
```

It results in 3 VCs. The all discharged internally by the Qed simplifier of WP.

- Notice the warning "Missing RTE guards", emitted by the WP plugin.
  - The WP calculus implemented relies on the hypothesis that programs are runtime-error free.
  - By default, the WP plugin does not generate any proof obligation for verifying the absence of runtime errors in the code. They can be proved by generating all the necessary annotations with the RTE plugin.

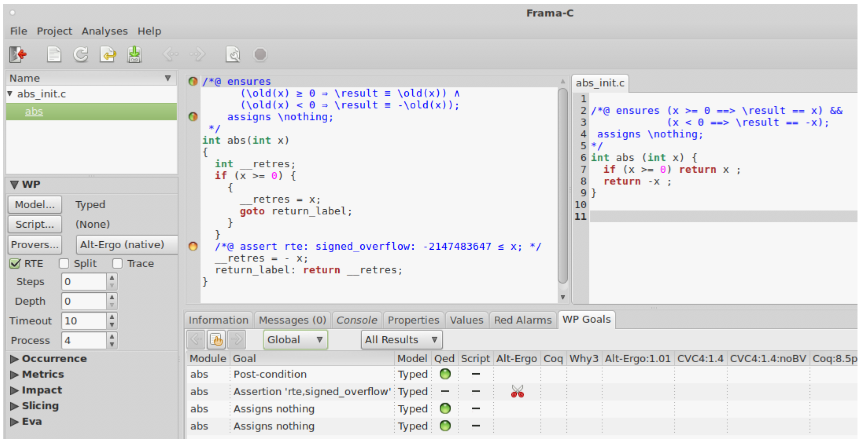
Runtime errors

```
$ frama-c -wp -wp-rte abs_init.c
[kernel] Parsing abs_init.c (with preprocessing)
[rte] annotating function abs
[wp] 4 goals scheduled
[wp] [Alt-Ergo] Goal typed_abs_assert_rte_signed_overflow : Unknown (52ms)
[wp] Proved goals:    3 / 4
  Qed:             3  (4ms)
  Alt-Ergo:        0  (unknown: 1)
```

It results in 4 VCs. TThree VCs discharged internally by the Qed simplifier and one sent to Alt-Ergo (with an inconclusive response).

Why the proof fails?

$ frama-c-gui -wp -wp-rte abs_init.c &



Frama-C GUI

- The options from the WP side panel correspond to some options of the plugin command-line.
- The *status of each code annotation* is reported in the left margin. The meaning of icons is the same for all plugins in Frama-C.

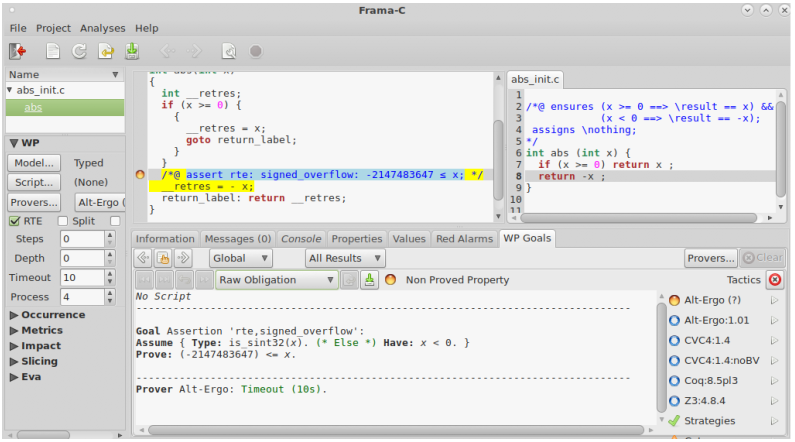
Icons for properties:

(blue)	No proof attempted.
(orange)	The property has not been validated.
(green/orange)	The property is *valid* but has dependencies.
(green)	The property and *all* its dependencies are *valid*.

- Left-clicking on an object in the normalized code view displays information about it in the "Information" tab of the Messages View and displays the corresponding object of the original source view,
- Right-clicking on an object opens a contextual menu. Items of this menu depend on the kind of the selected object and on plugin availability.
- Ctrl-clicking on the original source code opens an external editor.

Proof editor

This panel focus on one goal generated by WP, and allow the user to visualize the logical sequent to be proved, and to interactively decompose a complex proof into smaller pieces by applying *tactics*.



Proof editor

- The logical sequent consists of a formula to `Prove` under the hypotheses listed in the `Assume` section. Each hypothesis can consists of :

`Type:`	formula expressing a typing constraint;
`Init:`	formula characterizing global variable initialisation;
`Have:`	formula from an assertion or an instruction in the code;
`When:`	condition from a simplification performed by Qed;
`If:`	structured hypothesis from a conditional statement;
`Either:`	structured disjunction from a switch statement.
`Stmt:`	labels and C-like instructions representing the memory updates

- There are several modes to display the current goal:

`Autofocus:`	filter out clauses not mentioning *focused* terms (see below);
`Full Context:`	disable autofocus mode — all clauses are visible;
`Unmangled Memory:`	autofocus mode with low-level details of memory model;
`Raw Obligation:`	no autofocus and low-level details of memory model.

Example 1

```
/*@ ensures (x >= 0 ==> \result == x) &&
            (x < 0 ==> \result == -x);
    assigns \nothing;
*/
int abs (int x) {
  if (x >= 0) return x ;
  return -x ;
}
```

Why the proof fails?

- For x==INT_MIN, -x overflows
- Example: for 32-bit, x==INT_MIN$=-2^{31}$ while x==INT_MAX$=2^{31}-1$

Example 1 (overflow safety)

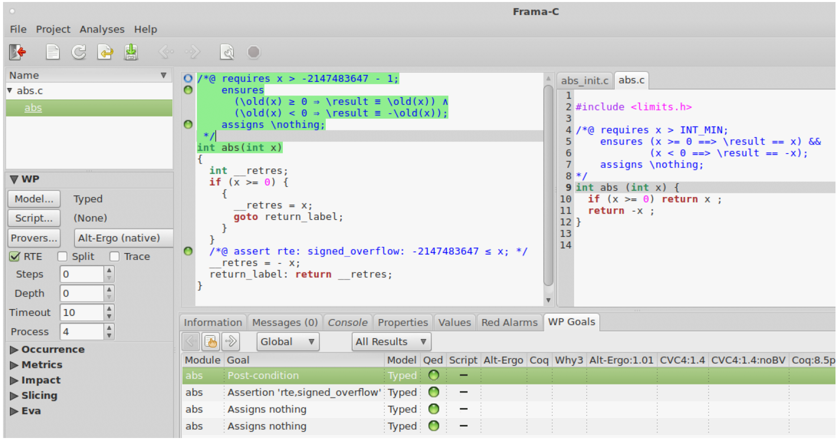
```
#include <limits.h>

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

It is sufficient to add as a precondition that `x` must be strictly greater than `INT_MIN` to guarantee that the underflow will never happen.

Notice that it is also necessary to include the header where INT_MIN is defined.

Example 1 (overflow safety)



ACSL (ANSI C Specification Language)

- Aims at specifying behavioral properties of C source code (inspired in JML).
- Based on the notion of contract.
  - Each function contract (safety included) is verified independently.
  - The correctness of all the remaining functions in the program is assumed.
- Specifications are given as annotations in comments written directly in C source files (`/*@ */` and `//@`).
- Basic features
  - First-order logic
  - Pure C expressions
  - C types + $\mathbb{Z}$ + $\mathbb{R}$
  - Buit-in predicates and logic functions, particularly for pointers: `\valid(p)`, `\valid(p+(0..n))`, `\separated(p+(0..5),q+(0..3))`, `\block_length(p)`, ...

ACSL annotations

- Global annotations
  - function contracts
  - global invariants
  - type invariants
  - logic specifications
- Statement annotations
  - loop annotations
  - assert clauses
  - statement contracts
  - ghost code
- We will learn the language through examples.

An example: **maxarray**

```
/*@ requires 0 < size && \valid(u+(0..size-1));
    ensures 0 <= \result < size;
    ensures
      \forall integer a; 0 <= a < size ==> u[a] <= u[\result];
    assigns \nothing;
*/
int maxarray(int u[], int size) {
  int i = 1;
  int max = 0;

  /*@ loop invariant \forall integer a;
                           0 <= a < i ==> u[a] <= u[max];
      loop invariant 0 <= max < i <= size;
      loop assigns max, i;
      loop variant size-i;   */
  while (i < size) {
    if (u[i] > u[max])  max = i;
    i++;
  }
  return max;
}
```

An example: **maxarray**

- Run Frama-C with the file **maxarray.c**
- Explore the WP plugin with this example.
- Observe VCs generated.
  - There are VCs related the verification of a function's default behavior, which includes verification of its postcondition, frame condition, loop invariants and intermediate assertions.
  - There are VCs guarding against safety violations such as null-pointer dereferencing, buffer overflow, arithmetic overflow, division by zero, termination, etc.

Program Specification

Example 2

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

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

Try **max_init.c**

This specification is incomplete. Why?
Give an example of a wrong implementation.

Example 2

This is the completely specified program:

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

Example 3

Why the proof of this program fails?

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

Run WP with **max_ptr_init.c**

Nothing ensures that p and q are valid pointers!

WP automatically generates VCs to check memory access validity.

Example 3 (memory safety)

Is this specification complete?

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

Give a valid implementation that does not work properly...

Example 3 (frame conditions)

This is the completely specified program:

```
/*@ 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;
}
```

- Lists of assigned variables explicitly included in contracts are called frame conditions.
- Avoids to state that for any unchanged global variable `v`, we have `ensures \old(v) == v`

Behaviours

The specification can be done by cases.

- Global preconditions and postconditions applies to all cases.
- Behaviours define contracts in particular cases.
- For each case (`behavior`):
  - `assumes` clause defines the subdomain.
  - the behaviour's precondition is defined by `requires` clauses.
  - the behaviour's postcondition is defined by `ensures`, `assigns` clauses.
- `complete behaviors` states that given behaviors cover all cases.
- `disjoint behaviors` states that given behaviors do not overlap.

Example 4

Try **abs_behavior_init.c** and fix the problems.

```
#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;
}
```


Example 4

This is the completely specified program:

```
#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;
}
```


Functions calls

Suppose function g contains a call to function f.

```
void g(...) {
  ...
  f(...);
  ...
}
```

Suppose we try to prove the caller g.

The function call is handled as follows:

- Before the call to f in g, the precondition of f must be ensured by g.
  - VCs are generated to prove that the precondition of f is respected.
- After the call to f in g, the postcondition of f is supposed to be true.
  - the postcondition of f is assumed in the proof below.
- Only a contract and a declaration of f are required.

Example 5

Run WP with **abs_calls.c** and see the problems.

```
#include <limits.h>

/*@ requires x > INT_MIN;
    ensures (x >= 0 ==> \result == x) &&
            (x < 0 ==> \result == -x);
    assigns \nothing;
*/
int abs (int x);

void foo(int a){
   int b = abs(42);
   int c = abs(-50);
   int d = abs(a);        // False : "a" can be INT_MIN
   int e = abs(INT_MIN); // False : the parameter must be
                          // strictly greater than INT_MIN
}
```

Example 6

Try **max_abs_init.c** and fix the problems.

```
#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

This is a solution:

```
#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;
    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 3 (memory safety)

- If we have a closer look to the assertions that WP adds in the `max_prt` function comprising RTE verification, we can notice that there exists another version of the `\valid` predicate, denoted `\valid_read`.
- The predicate `\valid_read` indicates that a pointer can be dereferenced, but only to read the pointed memory.
- Try the following annotation:

```
/*@ requires \valid_read(p) && \valid_read(q);
    ensures \result >= *p && \result >= *q;
    ensures \result == *p || \result == *q;
*/
int max_ptr (int *p, int *q);
```


Example 7

- Run WP with **unref_init.c**

```
/*@ requires \valid(p);
 */
int unref(int* p){
    return *p;
}

int const value = 42;

int main(){
    int i = unref(&value);
}
```

- Dereferencing `p` is valid, however the precondition of `unref` will not be verified by WP since dereferencing `value` is only legal for a read-access.
- Fix the problem.

Example 8

Run WP with **proc_mem_init.c**

```
/*@ requires \valid(a) && \valid(b);
    ensures *a==10 && *b==20;
    assigns *a, *b;
*/
void proc(int *a, int *b) {
  *a = 10;
  *b = 20;
}
```

Why it fails?

Recall that WP memory model does not make any assumptions about memory regions, and they can overlap.

With this in mind, fix the problem.

Example 8 (solution)

```
/*@ requires \valid(a) && \valid(b);
    requires a != b;
    ensures *a==10 && *b==20;
    assigns *a, *b;
*/
void proc(int *a, int *b) {
  *a = 10;
  *b = 20;
}
```


Example 9

Write the ACSL specification corresponding to the following informal specification of function `find_array`.

> `find_array(arr, len, x)` *returns any index* `idx` *of the sorted array* `arr` *of length* `len` *such that* `arr[idx] == x`. *If such an index does not exist, it returns* `-1`.

```
int find_array(int* arr, int len, int x);
```


Example 9

Here is a correct answer:

```
/*@ requires len >= 0;
    requires \valid(arr+(0..(len-1)));
    requires \forall integer i, j;
           0<=i<=j<len ==> arr[i]<=arr[j];
    ensures (\exists integer i; 0<=i<len && arr[i]==x)
           ==> 0<=\result<len && arr[\result]==x;
    ensures (\forall integer i; 0<=i<len ==> arr[i]!=x)
           ==> \result==-1;
    assigns \nothing;
*/
int find_array(int* arr, int len, int x);
```

Example 9

Here is a correct answer which defines two behaviours:

```
/*@ requires \forall integer i, j;
                0 <= i <= j < len ==> arr[i] <= arr[j];
    requires len >= 0;
    requires \valid(arr+(0..(len-1)));
    assigns \nothing;

    behavior belongs:
      assumes \exists integer i;
                 0 <= i < len && arr[i] == x;
      ensures 0 <= \result < len;
      ensures arr[\result] == x;

    behavior not_belongs:
      assumes \forall integer i;
                  0 <= i < len ==> arr[i] != x;
      ensures \result == -1;
*/
int find_array(int* arr, int len, int x);
```


Example 9 (logical predicates)

We can define two logical predicates:

- `sorted` which states that a given array is sorted
- `elem` which states that an element belongs to a given array

```
/*@ predicate sorted(int* arr, integer length) =
       \forall integer i, j; 0<=i<=j<length
                                  ==> arr[i]<=arr[j];

    predicate elem(int v, int* arr, integer length) =
       \exists integer i; 0<=i<length && arr[i]==v;
*/
```

Modify the previous specification to use these predicates.

Example 9

Here is a correct answer, with behaviours, using the logical predicates defined:

```
/*@ requires sorted(arr,len);
    requires len >= 0;
    requires \valid(arr+(0..(len-1)));

    assigns \nothing;

    behavior belongs:
      assumes elem(x, arr, len);
      ensures 0 <= \result < len;
      ensures arr[\result] == x;
    behavior not_belongs:
      assumes ! elem(x, arr, len);
      ensures \result == -1;
*/
int find_array(int* arr, int len, int x);
```


Assert annotations

- `assert` p means that p must hold in the current state (the sequence point where the assertion occurs).
- `for` $id_1, \ldots, id_n$: `assert` p associates the assertion to the named behaviours id_i. It means that this assertion must hold only for the considered behaviours.

Exemplo 10

Try **foo_assert_init.c** and fix the problems.

```
/*@ requires \valid_read(p) && \valid_read(q);
    ensures \result >= *p && \result >= *q;
    ensures \result == *p || \result == *q;
*/
int max_ptr (int *p, int *q);

void foo(){
    int a = 42;
    int b = 37;

    b += 10;
    int c = max_ptr(&a,&b);
    //@ assert c == 47;
}
```


State label mechanism

- Specification may require values at different program points.
- ACSL uses a state label mechanism that allows to refer to the value of a variable in any point of the program.
- Use `\at(e,L)` to refer to the value of expression `e` at label `L`.
- Predefined logic labels:
  - `Here` refers to the point where the annotation appears;
  - `Old` or `Pre` refers to the point before function call;
  - `Post` refers to the point after function call;
  - `LoopEntry` refers to the point at loop entry;
  - `LoopCurrent` refers to the point at the beginning of the current step of the loop.
- `\old(e)` is equivalent to `\at(e,Old)`, and `\at(e,Here)` to `e`.

Exemplo 10

Check **foo_assert_at.c**.

```
/*@ requires \valid_read(p) && \valid_read(q);
    ensures \result >= *p && \result >= *q;
    ensures \result == *p || \result == *q;
    assigns \nothing;
 */
int max_ptr (int *p, int *q);

void foo(){
    int a = 42;
    int b = 37;

  Label_b:
    b += 10;
    int c = max_ptr(&a,&b);
    //@ assert c == 47;
    b = c+b;
    //@ assert b == 94 && \at(b,Label_b) == 37;
}
```


Program Verification

Example 11 (binary search)

A possible implementation of the specification given for `find_array` is

```
int find_array(int* arr, int len, int x) {
  int low = 0;
  int high = len - 1;
  while (low <= high) {
    int mean = (low + high) / 2;
    if (arr[mean] == x) return mean;
    if (arr[mean] < x) low = mean + 1;
    else high = mean - 1;
  }
  return -1;
}
```

- Check **binary_search_init.c**.
- Prove the correction of this implementation w.r.t. its specification.
  - 7 unknown VCs remain (all located inside loop or after loop)
- This pinpoints a classic difficulty: reasoning about loops

Loops and proof

- Main difficulty: to find appropriate loop invariants for each loop of the program.
- The invariants are *the only thing* that is known about the state of the program after the loop.
  - They must thus be strong enough to allow us to prove postconditions.
  - But not too strong, or we will not be able to prove the invariants themselves.
- The proof of loop invariants is done by induction.
  - it must hold before the loop (0 iterations)
  - it must hold after k+1 iterations whenever it holds after k iterations
- The VCs for a loop invariant include two parts
  - loop invariant initially holds
  - loop invariant is preserved by any iteration

Loop invariants

Loop invariants may be tricky.

How to find a suitable loop invariant?

- identify variables modified in the loop
  - define their possible value intervals (relationships) after k iterations
  - `loop assigns` clause can be used 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?
  - 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.

Example 11 (loop invariants)

Write loop invariants to prove the safety properties for `find_array`

The following invariants show that `low` and `high` are within `arr`'s bounds:

```
/*@ loop invariant 0 <= low;
    loop invariant high < len;
*/
```

There is still an arithmetic overflow VC unknown. Why? Fix the problem!

```
   int mean = low + (high-low) / 2;
```

Example 11

Write the loop invariants that allow to prove the postconditions of behaviours for `find_array`.

```
/*@ loop invariant \forall integer i;
                        0 <= i < low ==> arr[i] < x;
    loop invariant \forall integer i;
                        high < i < len ==> arr[i] > x;
*/
```

There are still 2 safety VCs not proved which concerns to loop termination.

Loop termination

- Program termination is undecidable.
- For proving loop termination one has to give a loop variant.
  - A loop variant is an upper bound on the number of remaining loop iterations.
  - A loop variant is an integer expression with a non-negative value which decreases on each iteration of the loop.
- To find a variant, look at the loop condition.

Example 11 (loop variant)

Provide a loop variant that ensures that the loop always terminates.

```
//@  loop variant high - low + 1;
```


Example 11 (loop assigns)

- Considering loops, WP only reasons about what is provided by the user to perform its reasoning.
- In this example, the invariant does not specify anything about the way the variables are assigned.
- ACSL allows to add assigns annotations for loops. Any other variable is considered to keep its old value.

```
//@  loop assigns low, high, mean;
```

  - This should be read as follows: *if at the beginning of a givem iteration only* `low, high` *and* `mean` *have beem written, then after the execution of the iteration only* `low, high` *and* `mean` *will have been written.*
  - When the `loop assigns` clause is omitted, the VCGen assumes it is equal to the frame condition of the routine.

Example 11 (solution)

```
int find_array(int* arr, int len, int x) {
  int mean;
  int low = 0;
  int high = len - 1;
  /*@ loop invariant 0 <= low;
      loop invariant high < len;
      loop invariant \forall integer i;
                        0<=i<low ==> arr[i] < x;
      loop invariant \forall integer i;
                        high<i<len ==> arr[i] > x;
      loop assigns low, high, mean;
      loop variant high - low + 1; */
  while (low <= high) {
    mean = low + (high-low) / 2;
    if (arr[mean] == x) return mean;
    if (arr[mean] < x) low = mean + 1;
    else high = mean - 1;
  }
  return -1;
}
```


Proof failures

A proof of a VC can fail for **various reasons**

- erroneous implementation
- incorrect specification
- missing or erroneous (previous) annotation
- complexity of the proof
  - try different provers
  - split the VC in independent properties
  - try a longer timeout
  - additional statements (assert, lemma, ...) may help the provers
  - if nothing else helps try an interactive proof assistant...

Exercises

Write a contract and prove the correcteness of the following code.

```
void change(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

See **change_init.c**.

Exercises

For each of the following functions:

- tests if an array has negative values

```
int negs(int A[], int N);
```

- returns the index where the minimum of an array is

```
int minarray(int A[], int N);
```

- tests if the segments [a..b] of two different arrays are equal

```
int equal_seg(int A[],int B[],int a,int b,int N);
```

- returns an index which value is x, if it exists; -1 otherwise

```
int where(int A[],int N, int x);
```

1. Write a ACSL contract.
2. Write the function definition and prove its safety and functional correctness.
3. Write a main function that invokes it and check it.

Predicates and Logical Functions

Predicates and logical functions

- The language of logic expressions used in annotations can be extended by declarations of new logic types, and new constants, logic functions and predicates.
- New functions and predicates can be defined by explicit expressions.

```
//@ predicate is_positive(integer x) = x > 0;

//@ logic real doble(real x) = x+x;
```

- Instead of an explicit definition, one may introduce an axiomatic definitions.

It is up to the user to ensure that the introduction of axioms does not lead to a logical inconsistency.

Axiomatic definitions

```
/*@ axiomatic factorial {

    predicate isfact(integer n, integer r);

    axiom isfact0:  isfact(0,1);

    axiom isfactn:  \forall integer n, f;
                        isfact (n-1,f) ==> isfact(n,f*n);

    logic integer fact (integer n);

    axiom fact1:  \forall integer n; isfact (n,fact(n));

    axiom fact2:
         \forall integer n, f; isfact (n,f) ==> f==fact(n);
    }
*/
```


Inductive predicates

- A predicate may also be defined by an inductive definition.
- An alternative to define the predicate `isfact` would be:

```
/*@ inductive isfact(integer n, integer f) {
       case isfact0:  isfact(0,1);
       case isfactn:  \forall integer n, f; isfact
                          (n-1,f) ==> isfact(n,f*n);
    }
*/
```

- The predicate is inductively defined by a set of Horn clauses, *à la* Prolog.
- It is up to the user to ensure that the introduction of inductive definitions does not lead to a logical inconsistency.

Example 12 (fact)

Run WP with **fact_init.c** and complete de proof making the necessary annotations in the loop.

```
/*@ requires n >= 0;
    ensures \result == fact(n);
*/
int fact (int n) {
  int f = 1;
  int i = 1;

  while (i <= n) {
    f = f * i;
    i = i + 1;
  }
  return f;
}
```


Example 12 (fact: solution)

```
/*@ requires n >= 0;
    ensures \result == fact(n);
*/
int fact (int n) {
  int f = 1;
  int i = 1;
  /*@ loop invariant i<=n+1 && f == fact(i-1);
      loop assigns f, i;
      loop variant n+1-i;  */
  while (i <= n) {
    f = f * i;
    i = i + 1;
  }
  return f;
}
```


Example 13 (swap)

Write the contract for swap. See **swap_init.c**.

```
void swap(int t[], int i, int j) {
  int tmp = t[i];
  t[i] = t[j];
  t[j] = tmp;
}
```


Example 13 (swap: solution)

```
/*@ requires \valid(t+i) && \valid(t+j);
    ensures t[i] == \old(t[j]) && t[j] == \old(t[i]);
    assigns t[i], t[j];
*/
void swap(int t[], int i, int j) {
  int tmp = t[i];
  t[i] = t[j];
  t[j] = tmp;
}
```

Hybrid functions and predicates

- Logic functions and predicates may take both (pure) C types and logic types arguments.
- Hybrid functions and hybrid predicates can either be defined (or axiomatized) with the same syntax as before.
- An hybrid function (or predicate) usually depends on one or more program points, because it depends upon memory states.
- To make such definitions safe, it is mandatory to add after the declared identifier a set of labels, between curly braces.

Example 13 (hybrid predicate)

We can define the following hybrid predicate.

```
/*@ predicate Swap{L1,L2}(int* a, integer i, integer j) =
      \at(a[i],L1) == \at(a[j],L2) &&
      \at(a[i],L2) == \at(a[j],L1) &&
      \forall integer k; k!=i && k!=j
                     ==> \at(a[k],L1) == \at(a[k] ,L2);
*/
```

`Swap{L1,L2}(a,i,j)` has the meaning that the contents of array a in states L1 and L2 are the same, with the exception of indexes i and j, which are swapped.

Example 13 (hybrid predicate)

The contract for the function swap could now be written as follows (see **swap_predicate.c**)

```
/*@ requires \valid(t+i) && \valid(t+j);
    ensures Swap{Old,Here}(t,i,j);
    assigns t[i], t[j];
*/
void swap(int t[], int i, int j) {
  int tmp = t[i];
  t[i] = t[j];
  t[j] = tmp;
}
```


Example 14 (partition: contract)

Consider the following contract for partition.

```
/*@
requires 0 <= p <= r && \valid(A+(p..r));
ensures p<=\result<=r;
ensures \forall integer l; p<=l<\result ==> A[l]<=A[\result];
ensures \forall integer l; \result<l<=r ==> A[l]>A[\result];
ensures A[\result] == \old(A[r]);
assigns A[p..r];
*/
int partition (int A[], int p, int r)
```

What does it tell us?

Example 14 (partition: implementation)

This is a possible implementation of the contract.
See **partition_swap_init.c**

```
int partition (int A[], int p, int r){
  int x = A[r];
  int j, i = p-1;

  for (j=p; j<r; j++)
    if (A[j] <= x) {
      i++;
      swap(A,i,j);
    }
  swap(A,i+1,r);
  return i+1;
}
```

Find the invariants and prove the correction of this implementation.

Example 14 (partition: solution)

```
int partition (int A[], int p, int r) {
  int x = A[r];
  int j, i = p-1;

  /*@ loop invariant p <= j <= r && p-1 <= i < j;
      loop invariant \forall integer k; p<=k<=i ==> A[k]<=x;
      loop invariant \forall integer k; i<k<j ==> A[k]>x;
      loop invariant A[r] == x;
      loop assigns j, i, A[p..r];
      loop variant r-j;
  */
  for (j=p; j<r; j++)
    if (A[j] <= x) {
      i++;
      swap(A,i,j);
    }
  swap(A,i+1,r);
  return i+1;
}
```


Example 14 (permutation)

- The `partition` routine should preserve the elements contained in the original array. The contract of `partition` is incomplete... (give an example.)
- An important property that has been left out is that the multiset of elements in the input array must be preserved in the output.
- Another way of stating this is that there exists a bijection on the set of indices that establishes a permutation between the two arrays.
- It is not easy to formalize this property. Let us try...

Example 14 (permutation: tentative solutions)

Let us try to formalize this property. Let $B[]$ to denote $A[]$ in the poststate.

- First attempt. *Comments?*

  $$\forall k.\, p \leq k \leq r \rightarrow (\exists l.\, p \leq l \leq r \rightarrow A[k] = B[l]) \\ \wedge \\ \forall k.\, p \leq k \leq r \rightarrow (\exists l.\, p \leq l \leq r \rightarrow B[k] = A[l])$$

  Too weak! (give an counterexample)

  It does not take into account number of occurrences (i.e. preserves the set but not the multiset).

- Second attempt. *Comments?*

  $$\forall k.\, p \leq k \leq r \rightarrow (\exists l.\, p \leq l \leq r \rightarrow A[k] = B[l] \wedge A[l] = B[k])$$

  Too strong! (give an counterexample)

  It only covers the cases in which B is directly obtained from A by swaping pairs of elements. A sequence of swaps produces an array that is no longer related to the original in this way.

Example 14 (permutation: a solution)

- The property to be expressed as a postcondition is that the array in the poststate is a *permutation* of the original array.
- One possibility to treat permutations is to see them as sequences of pairwise swaps.
- The hybrid perdicate

  `Permut{L1,L2}(int *a, integer l, integer h)`

  means that array a contains in state L2, between indices l and h, a permutation of the elements contained in a, in the same range, in state L1.

Example 14 (permutation: a solution)

```
/*@
inductive Permut{L1,L2}(int *a, integer l, integer h) {
   case Permut_refl{L}:
        \forall int *a, integer l, h; Permut{L,L}(a,l,h);
   case Permut_sym{L1,L2}:
        \forall int *a, integer l, h;
            Permut{L1,L2}(a,l,h) ==> Permut{L2,L1}(a,l,h);
   case Permut_trans{L1,L2,L3}:
        \forall int *a, integer l, h;
            Permut{L1,L2}(a, l, h) && Permut{L2,L3}(a, l, h)
                 ==> Permut{L1,L3}(a,l,h);
   case Permut_swap{L1,L2}:
        \forall int *a, integer l, h, i, j;
            l<=i<=h && l<=j<=h && Swap{L1,L2}(a,i,j)
                 ==> Permut{L1,L2}(a,l,h);
}
*/
```


Example 14 (partition: complete contract)

```
/*@ requires 0 <= p <= r && \valid(A+(p..r));
    ensures p<=\result<=r;
    ensures \forall integer l;
                    p <= l < \result ==> A[l] <= A[\result];
    ensures \forall integer l;
                    \result < l <= r ==> A[l] > A[\result];
    ensures  A[\result] == \old(A[r]);
    ensures Permut{Old,Here}(A,p,r);
    assigns A[p..r];
*/
int partition (int A[], int p, int r);
```

Load **partion_permutation_init.c**, complete the contract and finish its proof.

Example 14 (complete solution)

```
int partition (int A[], int p, int r) {
  int x = A[r];
  int j, i = p-1;

  /*@ loop invariant p <= j <= r && p-1 <= i < j;
      loop invariant \forall integer k; p<=k<=i ==> A[k]<=x;
      loop invariant \forall integer k; i<k<j ==> A[k]>x;
      loop invariant A[r] == x;
      loop invariant Permut{Pre,Here}(A,p,r);
      loop assigns j, i, A[p..r];
      loop variant r-j;
  */
  for (j=p; j<r; j++)
    if (A[j] <= x) {
      i++;
      swap(A,i,j);
    }
  swap(A,i+1,r);
  return i+1;
}
```

Lemmas

- One can devise additional assertions or ACSL lemmas to guide the automatic provers.
- Lemmas are user-given propositions, a facility that might help theorem provers to establish validity of the VCs.
  - The reason for this is that ACSL lemmas usually have a much smaller set of hypotheses than proof obligations directly related to the C code.
- Lemmas will generate proof obligations (perhaps to be proved interactively, since they will possibly be complex).
- A complete verification of an ACSL specification has to provide a proof for each lemma.

Example 14 (a lemma about permutation)

```
/*@ lemma Permut_swap_sequence{L1,L2,L3}:
      \forall int *a, integer l, h, i, j;
             Permut{L1,L2}(a, l, h)
                 ==> l<=i<=h ==> l<=j<=h
                     ==> Swap{L2,L3}(a, i, j)
                         ==> Permut{L1,L3}(a, l, h);
*/
```

Prove the lemma present in **partion_permutation_init.c**.

Exemple 15 (`max_subarray`: a more complex example)

- Let us take a more complex example.

```
int max_subarray(int *a, int len) {
  int max = 0;
  int cur = 0;
  for (int i = 0; i < len; i++) {
    cur += a[i];
    if (cur < 0) cur = 0;
    if (cur > max) max = cur;
  }
  return max;
}
```

- We want to prove that this function returns the value of the maximal sum of subarrays (segments) of a given array.
- In order to specify this function, we will need an axiomatic definition about sum.

Exemple 15 (the predicate `sum`)

- Here is an axiomatic definition about predicate `sum`.

```
/*@
  axiomatic Sum_array{
    logic integer sum(int* array, integer begin, integer end)
                                       reads array[begin .. (end-1)];

    axiom empty:
      \forall int* a, integer b, e; b >= e ==> sum(a,b,e) == 0;
    axiom range:
      \forall int* a, integer b, e; b < e ==> sum(a,b,e) == sum(a,b,e-1)+a[e-1];
  }
*/
```

- The `reads` clause allows specifying the *footprint* of a hybrid predicate or function, that is, the set of memory locations that it depends on.
  - From such information, one might deduce properties of the form *f{L1}(args) = f{L2}(args)* if it is known that between states *L1* and *L2*, the memory changes are disjoint from the declared footprint.

Exemple 15 (max_subarray: the specification)

The specification of the function max_subarray is the following:

```
/*@
  requires \valid(a+(0..len-1));
  assigns \nothing;
  ensures \forall integer l, h;
                 0 <= l <= h <= len ==> sum(a,l,h) <= \result;
  ensures \exists integer l, h;
                 0 <= l <= h <= len &&  sum(a,l,h) == \result;
*/
int max_subarray(int *a, int len);
```


Exemple 15 (max_subarray: proving the specification)

```
/*@
  requires \valid(a+(0..len-1));
  assigns \nothing;
  ensures \forall integer l, h;
                 0 <= l <= h <= len ==> sum(a,l,h) <= \result;
  ensures \exists integer l, h;
                 0 <= l <= h <= len &&  sum(a,l,h) == \result;
*/
int max_subarray(int *a, int len) {
  int max = 0;
  int cur = 0;
  for (int i = 0; i < len; i++) {
    cur += a[i];
    if (cur < 0) cur = 0;
    if (cur > max) max = cur;
  }
  return max;
}
```


Exemple 15 (max_subarray: proving the specification)

About the proof of this specification:

- When we want to add the loop invariant, we will realize that we miss some information.
- We want to express what are the values `max` and `cur` and what are the relations between them, but we cannot do it!
- Basically, our postcondition needs to know that there exists some bounds `low` and `high` such that the computed sum corresponds to these bounds. However, in our code, we do not have anything that express it from a logic point of view.
- We can then use **ghost code** to record these bounds and express the loop invariant.

Ghost code

- **Ghost code** is regular C code, only visible from the specifications, that is only allowed to modify ghost variables.
- The idea is to add variables and source code that will not be part of the actual program but will model logic states that will only be visible from a proof point of view.
- Using it, we can make explicit some logic properties that were previously only known implicitly.
- Ghost code is added using annotations that will contain C code introduced using the `ghost` keyword:

```
/*@ ghost
  // code in C language
*/
```

We must be careful using ghost code! The tool will not perform any verification to ensure that we do not write in the memory of the program by mistake.

Exemple 15 (max_subarray: ghost code)

- We will first need two variables, low and high, that will allow us to record the bounds of the maximum sum range.
  - Every time we will find a range where the sum is greater than the current one, we will update our ghost variables.
  - This bounds will then corresponds to the sum currently stored by max.
- We need other bounds: the ones that corresponds to the sum store by the variable cur from which we will build the bounds corresponding to current low bound.
  - For these bounds, we will only add a single ghost variable: the current low bound cur_low, the high bound being the variable i of the loop.

Exemple 15 (max_subarray: ghost code)

```
int max_subarray(int *a, int len) {
  int max = 0;
  int cur = 0;
  //@ ghost int cur_low = 0;
  //@ ghost int low = 0;
  //@ ghost int high = 0;

  for (int i = 0; i < len; i++) {
    cur += a[i];
    if (cur < 0) {
      cur = 0;
      /*@ ghost cur_low = i+1; */
    }
    if (cur > max) }
      max = cur;
      /*@ ghost low = cur_low; */
      /*@ ghost high = i+1; */
    }
  }
  return max;
}
```


Exemple 15 (max_subarray: loop annotations)

```
int max_subarray(int *a, int len) {
  int max = 0;
  int cur = 0;
  //@ ghost int cur_low = 0;
  //@ ghost int low = 0;
  //@ ghost int high = 0;

  /*@
    loop invariant BOUNDS: low <= high <= i <= len && cur_low <= i;

    loop invariant REL:    cur == sum(a,cur_low,i) <= max == sum(a,low,high);
    loop invariant POS:    \forall integer l;    0 <= l <= i       ==> sum(a,l,i) <= cur;
    loop invariant POS:    \forall integer l, h; 0 <= l <= h <= i ==> sum(a,l,h) <= max;

    loop assigns i, cur, max, cur_low, low, high;
    loop variant len - i;
  */
  for (int i = 0; i < len; i++) {
    cur += a[i];
    if (cur < 0) {
      cur = 0;
      /*@ ghost cur_low = i+1; */
    }
    if (cur > max) }
      max = cur;
      /*@ ghost low = cur_low; */
      /*@ ghost high = i+1; */
    }
  }
  return max;
}
```


Exemple 15 (max_subarray: check **ghost.c**)

```
/*@ requires \valid(a+(0..len-1));
    assigns \nothing;
    ensures \forall integer l, h; 0 <= l <= h <= len ==> sum(a,l,h) <= \result;
    ensures \exists integer l, h; 0 <= l <= h <= len &&  sum(a,l,h) == \result;
*/
int max_subarray(int *a, int len) {
  int max = 0;
  int cur = 0;
  //@ ghost int cur_low = 0;
  //@ ghost int low = 0;
  //@ ghost int high = 0;
  /*@
    loop invariant BOUNDS: low <= high <= i <= len && cur_low <= i;
    loop invariant REL:    cur == sum(a,cur_low,i) <= max == sum(a,low,high);
    loop invariant POS:    \forall integer l;    0 <= l <= i       ==> sum(a,l,i) <= cur;
    loop invariant POS:    \forall integer l, h; 0 <= l <= h <= i ==> sum(a,l,h) <= max;
    loop assigns i, cur, max, cur_low, low, high;
    loop variant len - i;
  */
  for (int i = 0; i < len; i++) {
    cur += a[i];
    if (cur < 0) {
      cur = 0;
      /*@ ghost cur_low = i+1; */
    }
    if (cur > max) }
      max = cur;
      /*@ ghost low = cur_low; */
      /*@ ghost high = i+1; */
    }
  }
  return max;
}
```

Exercises

- Consider the following function that sorts an array in increasing order.

```
void maxSort (int *a, int size) {
  int i, j;

  for (i=size-1; i>0; i--) {
    j = maxarray(a,i+1);
    swap(a,i,j);
  }
}
```

1. Write a contract that guarantees the safety of this function and prove it.
2. Improve the function contract in order to guarantee that the array produced by the function is sorted in increasing order. Write the loop invariants in order to prove it.
3. Complete the contract in order to claim that the function implements a sorting algorithm. Then complete the proof.

Exercises

- The following function counts the occurrences of `x` in the array `a` of size `n`.

```
int numOccur (int *a, int n, int x);
```

1. Declare a logical function `count` that determines the number of occurrences of a value in an array, and present an axiomatic definition for it.
2. Write a contract for `numOccur`.
3. Write the function definition and prove its safety and functional correctness.

- The following function reverse an array `a` of size `n`.

```
void reverse (int a[], int n);
```

1. Write a contract for `reverse`.
2. Write the function definition and prove its safety and functional correctness.