Checking Program Properties

Using static Analysis (and Frama-C/WP)

Jorge López

Acknowledgements

- These slides follow the structure and use some examples of [1,2], great sources of information.

- A complete ACSL specification can be found in [3]

[1] Prevosto, V. ACSL Mini-Tutorial: URL - *https://frama-c.com/download/acsl-tutorial.pdf* - *Last visited 2018-05-30*, 2013

[2] Kosmatov, N., Signoles, J. Frama-C, A Collaborative Framework for C Code Verification: Tutorial Synopsis, 16th International Conference on Runtime Verification, 2016

[3] Baudin, P., Filliâtre, J. C., Marché, C., Monate, B., Moy, Y., & Prevosto, V. ACSL: ANSI/ISO C Specification Language, version 1.4, 2009

What is ACSL?

- The ANSI/ISO C Specification Language (ACSL) is used to specify properties of C programs in a **formal** language
- It is then possible to verify that a program does not violate the specified set of properties (sometimes called a specification)
- To formally prove the desired properties, **Frama-C** (using the WP plugin) uses value analysis, weakest precondition calculus, and theorem provers (e.g., Alt-Ergo)

Function Contracts

- Basic notion of ACSL specifications, pre-/post- conditions

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

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

- `ensures` is a postcondition
- `requires` is a precondition

Function contracts (II)

- Pointers also work in Frama-C, e.g.:

```
/*@ requires \valid(x) && \valid(y);
    ensures *x <= *y;
*/
void sort(int *x, int *y)
```

- The function `\valid` requires a valid memory address which is large enough to allocate an int

Degree of "completeness"

- The degree of completeness of properties in a function contract (referred to as degree of completeness of the specification)
- The following holds!

```
/*@ requires \valid(x) && \valid(y);
    ensures *x <= *y;
*/
void sort(int *x, int *y)
{
        *x = *y = 0;
}
```

Degree of "completeness" (II)

- A ***complete*** function contract (specification)

- The following fails!
  - `\old` is a function that evaluates the value of the variable in the pre-state (before the function call)

```
/*@ requires \valid(x) && \valid(y);
    ensures *x <= *y;
    ensures *x == \old(*y) && *y == \old(*x)
         || *x == \old(*x) && *y == \old(*y);
*/
void sort(int *x, int *y)
{
        *x = *y = 0;
}
```

Degree of "completeness" (exercise)

- How to complete the contract for the function `max`? help me 😀

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

int max(int x, int y)
{
 return (x > 0 && y > 0)?x + y:(x > y)?x-y:y-x;
}
```

Degree of “completeness” (exercise)

- That will fail!

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

int max(int x, int y)
{
 return (x > 0 && y > 0)?x + y:(x > y)?x-y:y-x;
}
```

Behaviors

- `assume` clause triggers a behavior
- All possible behaviors and disjoint (mutually exclusive ones can be specified)

```
/*@ requires \valid (x) && \valid (y);
ensures  *x <= *y;
behavior  ascending:
assumes  *x < *y;
ensures  *x == \old (*x) && *y == \old (*y);
behavior  descending:
assumes  *x >= *y;
ensures  *x == \old (*y) && *y == \old (*x);
complete behaviors  ascending, descending;
disjoint behaviors  ascending, descending;
*/
void sort(int* x, int* y)
```

Arrays

- We can, for example, use quantifiers and require valid “blocks” of memory

```
/*@
   requires size > 0;
   requires \valid(arr + (0 .. size - 1));
   ensures \forall int i; 0 <= i < size ==> \result >= arr[i];
   ensures \exists int e; 0 <= e < size && \result = arr[e];

*/
int max_array (int arr[], size_t size)
```

- Is it true?

Arrays (II)

- Something's missing

```
/*@
   requires size > 0;
   requires \valid(arr + (0 .. size - 1));
   ensures \forall int i; 0 <= i < size ==> \result >= arr[i];
   ensures \exists int e; 0 <= e < size && \result == arr[e];
*/
int max_array (int arr[], size_t size)
{
        int i = 0;
        while (i < size)
                arr[i++] = 0;

        return 0;
}
```

Arrays (III)

- Something's missing

```
/*@
   requires size > 0;
   requires \valid(arr + (0 .. size - 1));
   ensures \forall int i; 0 <= i < size ==> \result >= arr[i];
   ensures \exists int e; 0 <= e < size && \result == arr[e];
   ensures \forall int i; 0 <= i < size ==> arr[i] == \old(arr[i]);
*/
int max_array (int arr[], size_t size)
{
        int i = 0;
        while (i < size)
                arr[i++] = 0;

        return 0;
}
```

Assigns and termination clauses

- `assigns` clauses list the allowed memory locations a problem is able to modify.

```
assigns \nothing;
assigns *p;
```

- The terminates clause describes the conditions in which a function must terminate

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

Small array exercise

- How to specify and prove a function that assigns an array of size *n* values in the range [2-100]?

```
int *my_func(int* arr, int n)
```

Small array exercise (II)

- How to specify and prove a function that assigns an array of size *n* values in the range [2-100]?

```
/*@ requires n > 0;
    ensures \forall int i; 0 <= i <= n -1 ==> 2
<= *(\result + i) <= 100;
*/
int *my_func(int n)
```

Loop Invariants

- Handling programs with loops is complicated to handle statically!
  - Unknown number of iterations
  - Proving some properties is only possible with adequate loop invariants!
- A Loop invariant must: hold initially and must be preserved by any iteration
- How to identify the invariants?
  - Identify what actually changes and use a `loop assigns` clause
  - What can the loop work guarantee on each iteration?
- A `loop variant` helps to know how many iterations remain in the loop
  - Look at why the loop terminates!

Handling loops by example

```
/*@  requires size > 0 && \valid(arr+(0..size-1));
     assigns \nothing;
     ensures \forall int i; 0 <= i <= size-1 ==>
\result >= arr[i];
     ensures \exists int e; 0 <= e <= size-1 &&
\result == arr[e];
*/
int max_array(int arr[], size_t size)
```

Handling loops by example (II)

```
int max_array(int arr[], size_t size) {
        int res = *arr;
        int *curr = arr;
        int element = 0;
        //loop invariants go here
        for(int i = 0; i < size; i++)
        {
                if (res < *curr)
                {
                        element =i;
                        res = *curr;
                }
                curr++;
        }
        return res;
}
```

```
/*@ loop invariant 0 <= i < size && 0 <= element < size;
    loop invariant \forall integer j; 0 <= j < i ==> res >= *(arr + j);
    loop invariant \forall int k; 0 <= k < i ==> *(arr + element) >=  *(arr + k);
    loop invariant res == *(arr + element);
    loop assigns i, res, element, curr;
    loop variant size - i ;
*/
```

Using Frama-C 101

- Most basic
  - `frama-c <file.c>`
- GUI
  - `frama-c-gui <file.c>`
- Enabling modules
  - `frama-C [-module]* <file.c>`, module names: val (value analysis), wp (weakest preconditions), rte (runtime annotation errors)
  - A good example: **`frama-c-gui -val -wp <file.c>`**

Warning!

- Many versions of Alt-ergo / Frama-c WP are broken!
  - Check simple properties and if they are failed to be proven not due to a timeout (you can use frama-c and not frama-c-gui to check this) check another version, carefully!

- It might help to install other provers after frama-c:

```
opam install altgr-ergo coq coqide why3 why
why3 config --detect provers
```

Lab

Specify and prove (if possible) the following:

- A leap year function returns 1 only when?..

```
int is_leap (year)
```

- An array is properly sorted if?..

```
void sort_array (int arr[], size_t size)
```

- Takes a prime number *p* in the range [2-13] and if $2^p - 1$ is also a prime number returns a *perfect number*; otherwise, if $2^p - 1$ is not a prime number it returns 0. *Hint*: you need to check `\sum` and `\lambda` (we did not cover this 😁). *Hint 2*: Shit left logical (<<) exists in ACSL.

```
int perfect (int p)
```

Submissions

- Both labs are due on June 6, 2018 @ 11:59:59 Moscow time!
- Send both of your labs to my email address.

Thank you!