ANSI/ISO C Specification Language (ACSL)

19CSE205 : PROGRAM REASONING

Dr. Swaminathan J

Assistant Professor

Department of Computer Science and Engineering

Jul - Dec 2020

Contents

What is ACSL?

ACSL is a specification language for C programs developed by Commissariat à l'Énergie Atomique and INRIA, France.

- Follows design by contract paradigm. Pre- and postconditions are stated for functions, commonly referred to as function contracts.
- Contracts are enclosed within special type of comments /*@ ... */ or //@ ... just above the function definition/declaration.
- Includes many more predicates to cater to the needs of the language and expressivity of the specification.

Only a few basic ACSL constructs to get started are discussed here.

The ensures predicate is used to specify the postcondition.

```
//@ ensures \result > a;
int increment(int a) {
   return a + 1;
}
```

\result is a generic way to refer to the return value of a function.

- The ensures keyword is followed by a logical condition that be to true/false followed by semi-colon.
- There can be more than one ensures instance for a function specified in multiple lines.
- Equivalently you can also concatenate them within a single ensures predicate instance by using logical operators &&, ||, ! operators.

2. The requires predicate

The requires predicate is used to specify the precondition.

```
/*@ requires a > 0;
      ensures \result > 1;
*/
int increment(int a) {
   return a + 1;
}
```

- requires and ensures together form the building block for specifying function contracts. There are more.
- If there is no requires predicate specified, it implies requires true;. i.e. precondition remains satisfied always.
- There can be one or more instances requires predicate for a function.

3. The assigns clause

The assigns clause is used to specify which global variable(s) can be modified by the function. It is part of function contract.

Example 1: Incorrect use

```
int g; // global
//@ assigns \nothing;
void setg() {
   g = 1;
}
```

Example 2: Correct use

```
int g, h = 0;
//@ assigns g;
void modifyg() {
   g = h + 1;
}
```

- In the absence of assigns clause in a function contract, the function is free to modify any visible global variable.
- assigns \nothing disallows modification of any global variable. This clause can be used as a means to avoid/minimize side-effects.
- assigns g allows only the variable g to be modified. Other global variables cannot be modified.

4. The built-in function \old

The built-in function \old is used to access the previous state of a variable.

```
int a; // global variable

//@ ensures a == \old(a) + 1;
void increment() {
   a++;
}
```

- The \old function evaluates its argument in the pre-state. i.e. as per the state before the function begins.

5. The built-in function \valid

The built-in function \valid is used to specify that the given argument points to a valid address. i.e. it can be de-referenced.

```
/*@ requires \valid(ptr);
      ensures \result == *ptr + 1;
*/
int increment(int * ptr) {
   return *ptr + 1;
}
```

- Though the increment logic is correct, if ptr happens to be a null pointer, it will only result in memory (segmentation) fault.
- \valid here states that increment contractually agrees to work correctly provided the argument ptr points to a valid memory location.
- It is required while working with arrays also since array boundaries cannot be exceeded. We will see the arrays later.

6. behavior and assumes

A function can exhibit more than one behavior. Verification must hence be different depending on the particular behavior.

- behavior is used to specify each behavior.
- assumes serves as the trigger for checking each.

```
/*@ behavior positive_a:
        assumes a > 0;
        ensures \result > a+1;
*/
int increment(int a) {
   if (a > 0)
      return a+1;
   else
      return a;
}
```

The behavior positive_a states that only if a is positive, the return value must be checked for the specified condition.

7. complete and disjoint behaviors

The previous example states only a partial behavior.

- Multiple behaviors can be stated to cover all possibilities so as to make the specification complete.
- The behaviors can be stated as disjoint so that one possibility does not result in triggering of two behaviors.

```
/*@ behavior positive_a:
        assumes a > 0;
        ensures \result > a+1;
    behavior negative_a:
        assumes a <= 0;
        ensures \result > a+2;
    complete behaviors positive_a, negative_a;
    disjoint behaviors positive_a, negative_a;
*/
int increment(int a) {
   return a>0 ? a+1 : a+2;
}
```

The behaviors positive_a and negative_a are stated to be complete and disjoint. So appropriate behavior check is triggered based on the assumes predicate.

8. The assert predicate

The assert predicate can be used to check for the truth of a condition at any point in the program.

```
...
//@ assert x >= 0;
x = x + 1;
//@ assert x > 0;
...
```

- The assert can be thought of as a statement level contract.
- It can be specified anywhere in the code.

Verfication constructs relating to loops and arrays will be covered later.