ACSL Mini-Tutorial

Virgile Prevosto[1]

[1] CEA LIST, Software Security Laboratory, Saclay, F-91191

This work has been partially supported by the CAT ANR project (ANR-05-RNTL-0030x) and by the FP6 European project Open TC.

Chapter 1

Foreword

This document is a brief introduction to the ANSI/ISO C Specification Language (ACSL). ACSL allows to formally specify the properties of a C program, in order to be able to formally verify that the implementation respects these properties. This verification is done *via* tools that are able to take into account ACSL annotations attached to the C code. This tutorial focuses on the most important ACSL constructs and gives an intuitive grasp of their semantics, through short and illustrative examples of C code annotated with ACSL formulas. A complete reference of the ACSL language can be found in [1]. ACSL is inspired from the specification language used by Caduceus [2], which is itself derived from the Java Modeling Language (JML, see [3]).

Chapter 2

A First ACSL Example

The most important ACSL concept is the *function contract*. A function contract for a C function `f` is a set of requirements over the arguments of `f` and/or a set of properties that are ensured at the end of the function. The formula that expresses the requirements is called a *pre-condition*, whereas the formula that expresses the properties ensured when `f` returns is a *post-condition*. Together, these conditions form a contract between `f` and its callers: each caller must guarantee that the pre-condition holds before calling `f`. In exchange, `f` guarantees that the post-condition holds when it returns.

Let us consider the example of the `max` function. Informally, its specification can be expressed this way: the function `max` takes two `int` as arguments, and returns the greatest one. Let us see how this can be expressed in ACSL:

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

As can be seen above, ACSL annotations are written in special C comments, the difference with plain comments being that annotations begin with '`/*@`'. It is also possible to write one-line annotations introduced by '`//@`'. The function contract is written immediately above the function declaration. In this example, the contract contains only post-conditions (`ensures` clauses), as `max` does not have any particular requirement. The first `ensures` clause expresses the fact that the result of `max` is greater than both `x` and `y`, the arguments of `max`. The second clause mandates that the result is equal to either `x` or `y`. Since both clauses must hold, our ACSL specification indeed expresses that the result of `max` is the greatest of its two arguments.

It should be noted already that there is an intuitive demarcation between "complete" and "partial" specifications. The above specification for `max` can be thought of as "complete", meaning that any function that satisfies the specification should be deemed a satisfactory implementation for `max`. Partial specifications on the other hand express some of the properties that are expected to hold for any implementation, but satisfying them is not sufficient for an implementation to be satisfactory. Generally speaking, partial formal specifications are the most likely to be encountered in practice for real-life examples, complemented by informal specifications. For instance, the above specification for `max` is in fact partial, for reasons that will become clearer later.

Chapter 3

Pointers

3.1 A First Specification

Let us now consider a small program involving pointers. Informally, the `max_ptr` function takes two pointers as argument, and if necessary swaps the two pointed values so that the value stored in the first pointer is the minimal one and the value stored in the second pointer is the maximal one. A specification can be the following:

```
/*@ requires \valid(p) && \valid(q);
    ensures *p<=*q;
*/
void max_ptr(int *p, int* q);
```

Here, we have a pre-condition (the `requires` clause). Namely, we want our function to be called with valid pointers as arguments. This is what the built-in ACSL predicate `\valid` says. Note that `\valid` takes into account the static type of its argument: in our context, `\valid(p)` indicates that there address `p` is included in an allocated block which is large enough to store an `int` starting at `p`. This is thus different from `\valid((char *)p)` for instance.

Our post-condition is that when the function returns, the value pointed to by `p` is less than or equal to the value pointed to by `q`.

A correct implementation for `max_ptr` is then

```
void max_ptr(int*p, int*q) {
  if (*p > *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
  }
}
```

Namely, since we require `max_ptr` to be called with valid pointers, there is no need to perform any check on them. Then, if `*p` is already less than or equal to `*q`, there's nothing to do. If not, we just swap the content of the two pointers.

3.2 Building a Complete Specification

The implementation seen in the previous section is correct with respect to the specification of `max_ptr`. Unfortunately, this is not the only conforming implementation: the specification is only partial, and is for instance met by the following function:

```
void max_ptr(int* p, int*q) {
  *p = *q = 0;
}
```

Indeed, since $0 \leq 0$, we have `*p<=*q` at the end of the function.

Depending on the verification tasks we have in mind (see next section), we may want to refine our specification to avoid an implementation such as above. Namely, we want to enforce a relation between the values pointed to by `p` and `q` at the beginning and at the end of the function. A possible specification is then

```
/*@ requires \valid(p) && \valid(q);
    ensures *p <= *q;
    ensures (*p == \old(*p) && *q == \old(*q)) ||
            (*p == \old(*q) && *q == \old(*p));
*/
void max_ptr(int* p, int*q);
```

The `\old` built-in function says that its argument must be evaluated in the pre-state (*i.e.* at the beginning) of the function. The second `ensures` clauses thus says that we either leave `p` and `q` unchanged or swap the two pointed values, which, together with the first clause implies that we meet exactly the informal specification.

3.3 Degree of Completeness

The previous section has teached us that writing a fairly complete specification (in fact we could still add some clauses to the specification above, as we will see in the next chapters) is not immediate, and thus that it is easy to come up with only a partial specification. Hence, it raises two frequently asked questions: how can we be sure that our specification is complete, and how complete must a specification be.

The answers however do not lie in ACSL itself. For the first one, one must reason on some model of the specified application. For the second one, there is no definite answer. It depends on the context in which the specification is written and the kind of properties that must be established: the amount of specification required for a given function is very different when verifying a given property for a given application in which calls to the function always occur in a well-defined context and when specifying it as a library function which should be callable in as many contexts as possible.

Chapter 4

Behaviors

The second `ensures` clause of our final specification of `max_ptr` is a bit complicated, and does not explain immediately in which case we will end up. It is possible to express that differently, by using ACSL's *behaviors*. A function can have several behaviors in addition to a general specification. A behavior can have additional `ensures` clauses, but in addition, it can also have `assumes` clauses, which indicate when the behavior is triggered. There is no requirement for the behaviors to cover all contexts in which a function can be called, and behaviors need not to cover disjoint cases. This can be further specified by the `complete behaviors` and `disjoint behaviors` clauses, as in the following specification.

```
/*@ requires \valid(p) && \valid(q);
    ensures *p <= *q;
    behavior p_minimum:
      assumes *p < *q;
      ensures *p == \old(*p) && *q == \old(*q);
    behavior q_minimum:
      assumes *p >= *q;
      ensures *p == \old(*q) && *q == \old(*p);
    complete behaviors p_minimum, q_minimum;
    disjoint behaviors p_minimum, q_minimum;
*/
void max_ptr(int* p, int*q);
```

We explain here precisely in which case we keep the same values and in which case we swap. Note that the global `ensures` clause is implied by the `ensures` clauses of the two behavior and the fact that `p_minimum` and `q_minimum` form a complete set of behaviors.

Chapter 5

Arrays

5.1 Basic Specification

Now that we have specified a `max_ptr` function, we can use it to extract the maximal value found in a sequence of `int`s. A first step is to write the prototype of the corresponding function with its specification.

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

The function takes two arguments: a pointer `p` to the block containing the `int`s, and the number `n` of elements in the sequence. This time, there are pre-conditions on these arguments. First, it is not possible to compute the maximal value of an empty sequence, and so `n` must be positive. Moreover, the block pointed to by `p` must contain at least `n` elements. In other words, `p[0]`, `p[1]`, ... `p[n-1]` must all be valid memory accesses. This is what the second `requires` clause expresses.

The two `ensures` clauses display some similarities with the contract of the `max` function above: the result is greater than or equal to every value in the sequence, and there exists an index for which the equality is attained. Note that the formulas in the post-condition only make sense under the assumption that the pre-condition holds: the validity condition ensures that it makes sense to speak about `p[i]`, and if `n` could be `0`, it would not be possible to find an index where `\result` is attained.

5.2 Advanced specification

Note: This section can be skipped on a first reading

In addition, a more advanced ACSL construction allows to provide a shorter specification of `max_seq`: `\max` is a built-in constructor (together with `\min`, `\sum`, and a few others), that returns the maximal value taken by a function in an interval of integers. With `\max` and the `\lambda` construction to write functions as first-class expressions, our specification becomes:

```
/*@ requires n > 0 &&\valid(p + (0..n-1));
    ensures \result == \max(0,n-1,\lambda integer i; p[i]);
*/
int max_seq(int* p, int n);
```

5.3 Implementation

The implementation of the `max_seq` function is pretty straightforward. For instance, we can use the following code.

```
int max_seq(int* p, int n) {
  int res = *p;
  for(int i = 0; i < n; i++) {
    if (res < *p) { res = *p; }
    p++;
  }
  return res;
}
```

The specification we have given in the preceding section defines an expected behavior for the function `max_seq`. We will see later in this document that actually verifying that the function `max_seq` implements the specification from the preceding section may require additional work.

Chapter 6

Assigns clauses

As with the initial specification of `max_ptr`, an implementation of `max_seq` could zero all the locations `p[0]`,`p[1]`,...,`p[n-1]`, return zero, and would still satisfy the post-conditions in the specification from section 5.1. Again, we can use the `\old` keyword to avoid that.

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

It would be possible, but tedious, to use the same approach to specify that global variables do not change during the execution of `max_seq`. The ACSL language provides a special clause to specify that a function is not allowed to change memory locations other than the ones explicitly listed. This clause is the `assigns` clause, and it is part of the function contract. When no `assigns` clauses are specified, the function is allowed to modify every visible variable. In presence of such clauses, the function can only modify the content of the locations that are explicitly mentioned in these clauses. In our case, we do not expect `max_seq` to have any visible side-effect, so that the contract becomes:

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

Again, it is not necessary to use `\old` for the values of the expressions `p[i]` and `p[e]` in the the post-conditions, since the specification forces them to stay unchanged during the execution of `max_seq`.

`assigns` clauses can be found in behaviors too. Writing appropriate `assigns` clauses for `p_minimum` and `q_minimum` to complete the specification of `max_ptr` is left as an exercise for the reader.

Chapter 7

Termination

There is yet another property that is implicitely expected from a satisfactory implementation of `max_seq`. Namely, this function, when called with arguments that satisfy its pre-conditions, should eventually terminate (and return a result that satisfies its post-conditions). The `assigns` clause in itself only guarantees that each time the function terminates, only the specified locations may have been modified. Similarly, the post-conditions only apply when the function terminates, and so a function that never terminates would for instance satisfy the `assigns` and `ensures` parts of the specification for `max_seq`. The termination of the function is a separate property that can be specified in its contract using the `terminates` clause.

Because in practice many functions are implicitely expected to terminate, the default in ACSL is to expect functions to terminate in all the contexts that satisfy their pre-conditions. It is possible to relax a particular function's specification by providing a formula that describes the conditions in which the function is guaranteed to terminate. An implementation is then allowed not to terminate when it is called in contexts that do not satisfy this condition.

In the following example, the function `f` can be called with any argument `c`, but the function is not guaranteed to terminate if `c<=0`.

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

Another valid contract for the same implementation of `f` is the following one, where the function implicitely guarantees to terminate whenever it is called, but expects to be called only with non-zero arguments.

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

Chapter 8

Predicates and Logic Functions

So far, we have only used logical built-ins operators and relations. It is often needed to define new logic predicates and logic functions. For instance, if we define (simply) linked lists as such:

```
typedef struct _list { int element; struct _list* next; } list;
```

there are some common properties of lists that we want to be able to deal with in the logic. In particular, the notion of reachability (can a given node be attained from some root through a chain of `next` fields) plays an important role. It can be defined in ACSL through the following annotation:

```
/*@
inductive reachable{L} (list* root, list* node) {
  case root_reachable{L}:
       \forall list* root; reachable(root,root);
  case next_reachable{L}:
       \forall list* root, *node;
       \valid(root) ==> reachable(root->next, node) ==>
       reachable(root,node);
}
*/
```

The notion of reachability is defined in ACSL by an `inductive` predicate. Namely, `root_reachable` and `next_reachable` can be seen as axioms indicating the cases under which `reachable` must hold, and the fact that `reachable` is inductive implies that these are the only cases under which it can hold. `root_reachable` indicates that `reachable` holds as soon as both its arguments are equal. `next_reachable` indicates that if `root` is valid and `node` can be proved reachable from `root->next`, then it is also reachable from `root`.

The `{L}` notation indicates that the predicate takes as parameter a label, which represent a memory state of the program. Indeed, as soon as we deal with pointers or arrays, we must refer to a particular memory state: `reachable` has no absolute meaning, it must be tied to a point of the program. In a case like this, there is no need to explicitely instantiate the label when we use `reachable`: the memory state to which it refers will usually be inferred from the context (*e.g.* for a precondition, it is the state at the start of the function, and for a post-condition, it is the state just before returning to the caller). More complex predicate may relate two or more memory states, but such definitions are out of scope in this tutorial.

Now, the `reachable` predicate can be used to discriminate between circular and finite lists: in a finite list, we will ultimately reach `\null`;

```
/*@ predicate finite{L}(list* root) = reachable(root,\null); */
```

Here, it is not necessary to embed `finite` as inductive. Instead, we directly give its definition in terms of reachability.

We can also note that `finite` is also parameterized by a label `L`. In fact, it is implicitely the memory state in which `reachable` itself is evaluated. It would have been possible to make that explicit by writing `reachable{L}(root,\null)` instead, but this is not necessary here, as `L` denotes the only state in the environment at this level.

Similarly, we can define a logical function computing the length of a `\null` terminated list. This time, we have to avoid circular lists, since the notion of length has little meaning for them. In order to do that, we can use an axiomatic:

```
/*@ axiomatic Length {
  logic integer length{L}(list* l);

  axiom length_nil{L}: length(\null) == 0;

  axiom length_cons{L}:
      \forall list* l, integer n;
        finite(l) && \valid(l) ==>
          length(l) == length(l->next) + 1;
 }
*/
```

An axiomatic is merely a set of logical declarations tied together. This way, it is possible to speak about `length(l)` for any list `l`, but if `l` is circular, the value of this expression will remain undefined: the only way to prove that `length(l)==n` for a given `n` is to prove first `finite(l)`.

Now that we know how to define logic predicates and logic functions, we can go back to our maximum example, this time on (finite) lists. A possible specification for `max_list` would be the following:

```
/*@
    requires \valid(root);
    assigns \nothing;
    terminates finite(root);
    ensures
     \forall list* l;
       \valid(l) && reachable(root,l) ==>
         \result >= l->element;
    ensures
     \exists list* l;
       \valid(l) && reachable(root,l) && \result == l->element;
*/
int max_list(list* root);
```

As with arrays, we have as pre-condition that the list is non-empty. The post-conditions are also quite similar to the ones of `max_array`, except that indices have been replaced by the reachability of the node from the root of the list. In addition, we find a `terminates` clause, that indicates that the implementation may loop forever on circular lists (but could choose not to, even though this would imply an huge overhead in this setting). A possible implementation is thus the following:

```
int max_list(list* root) {
  int max = root->element;
  while(root->next) {
    root = root -> next;
    if (root->element > max) max = root->element;
  }
  return max;
}
```

Chapter 9

Data Invariants

This chapter can be skipped on a first reading.

9.1 Type Invariants

The specification of the `max_list` function given above is complete, but it could be written differently. In fact, if we are writing a program that manipulates `NULL`-terminated linked lists, a lot a function will rely on the fact that the manipulated lists are finite. This means that each function must have a precondition or a `terminates` clause similar to the one of `max_list`, but also that the functions that build lists must ensure (with a post-condition) that the returned list is finite. Writing all these clauses might be tedious for a large set of functions. Thus, ACSL provide a way to express that all the elements of a given datatype respects some *type invariants*. By default, these invariants are *weak invariants* in the ACSL sense, that is, they must be valid at the entrance and at the exit of each function, but can be temporarily broken inside a function's body (provided they are restored before the function returns, or before calling another function which relies on them). ACSL has also a notion of *strong invariant*, which must hold at every point of the program, but they are harder to use in practice, and beyond the scope of this tutorial.

In our `list` example, we can impose that each list is finite with the following annotation:

```
/*@ type invariant finite_list(list* root) = finite(root); */
```

The type invariant is a unary predicate. The type of its argument is the static type to which the invariant applies (in our case pointers to list). With this invariant, we can get rid of the `terminates` clause in the specification of `max_list`, since we are guaranteed that only finite lists will be passed to the function. Its specification will thus be the following:

```
/*@
    requires \valid(root);
    assigns \nothing;
    ensures
     \forall list* l;
       \valid(l) && reachable(root,l) ==>
         \result >= l->element;
    ensures
     \exists list* l;
       \valid(l) && reachable(root,l) && \result == l->element;
*/
int max_list(list* root);
```

9.2 Global Invariants

There is another kind of data invariant in ACSL, called *global invariant*. As indicated by its name, a global invariant is a property of global variables that must hold at each entrance or exit of a function (for weak invariants) or at each point of the program (for strong invariants). For instance, suppose that we have a global list `GList` that is

always non-empty and maintained in decreasing order, so that `max_list(GList)` can be replaced in the code by `if(GList) GList->element`. A possible way to express that in ACSL is the following:

```
/*@
inductive sorted_decreasing{L}(list* root) {
  case sorted_nil{L}: sorted_decreasing(\null);
  case sorted_singleton{L}:
  \forall list* root;
     \valid(root) && root->next == \null ==>
        sorted_decreasing(root);
  case sorted_next{L}:
  \forall list* root;
     \valid(root) && \valid(root->next) &&
     root->element >= root->next->element &&
        sorted_decreasing(root->next) ==> sorted_decreasing(root);
}
*/

list* GList;

/*@ global invariant glist_sorted: sorted_decreasing(GList); */

void insert_GList(int x);
```

Because of the invariant of `GList`, a partial specification of `insert_GList` is that it can assume that GList is a sorted list, and that it must ensure that it is still the case when it returns. A complete specification, for instance that all the elements that were present at the entrance of the function are still there, and that `x` is present (inserted according to the ordering since the global invariant must still hold) can be expressed as such:

```
/*@
axiomatic Count {
  logic integer count(int x,list* l);

  axiom count_nil: \forall int x; count(x,\null) == 0;

  axiom count_cons_head{L}:
  \forall int x,list* l;
    \valid(l) && l->element == x ==>
       count(x,l) == count(x,l->next)+1;

 axiom count_cons_tail{L}:
  \forall int x, list* l;
     \valid(l) && l->element != x ==>
        count(x,l) == count(x,l->next);
}
*/

/*@ ensures \forall int y; y != x ==>
               count(y,GList) == count(y,\old(GList));
    ensures count(x,GList) == count(x,\old(GList))+1;
*/
void insert_GList(int x);
```

Chapter 10

Verification activities

The preceding examples have shown us how to write the specification of a C function in ACSL. However, at verification time, it can be necessary to write additional annotations in the implementation itself in order to guide the analyzers.

10.1 Assertions

The simplest form of code annotation is an *assertion*. An assertion is a property that must be verified each time the execution reaches a given program point. Some assertions may be discharged directly by one analyzer or another. When this happens, it means that the analyzer has concluded that there was no possibility of the assertion not being respected when the arguments satisfy the function's pre-conditions. Conversely, when the analyzer is not able to determine that an assertion always holds, it may be able to produce a pre-condition for the function that would, if it was added to the function's contract, ensure that the assertion was verified.

In the following example, the first assertion can be verified automatically by many analyzers, whereas the second one can't. An analyzer may suggest to add the pre-condition `n > 0` to `f`'s contract.

```
/*@ requires n >= 0 && n < 100;
*/
int f(int n)
{
  int tmp = 100 - n;
  //@ assert tmp > 0;
  //@ assert tmp < 100;
  return tmp;
}
```

Let us now move on to a more interesting example. The function `sqsum` below is meant to compute the sum of the squares of the elements of an array. As usual, we have to give some pre-conditions to ensure the validity of the pointer accesses, and a post-condition expressing what the intended result is:

```
/*@ requires n >= 0;
    requires \valid(p+ (0..n-1));
    assigns \nothing;
    ensures \result == \sum(0,n-1,\lambda integer i; p[i]*p[i]);
*/
int sqsum(int* p, int n);
```

A possible implementation is the following:

```
int sqsum(int* p, int n) {
  int S=0, tmp;
  for(int i = 0; i < n; i++) {
    tmp = p[i] * p[i];
    S += tmp;
```

```
  }
  return S;
}
```

This implementation seems to be correct with respect to the specification. However, this is not the case. Indeed, the specification operates on mathematical (unbounded) `integers`, while the C statements uses modulo arithmetic within the `int` type. If there is an overflow, the post-condition will not hold. In order to overcome this issue, a possible solution is to add some assertions before each `int` operation ensuring that there is no overflow. In the annotations, the arithmetic operations +,−,... are the mathematical addition, substraction,... in the `integer` domain. It is therefore possible to compare their results to `INT_MAX`. The code with its annotations is the following:

```
#include <limits.h>

int sqsum(int* p, int n) {
  int S=0, tmp;
  for(int i = 0; i < n; i++) {
    //@ assert p[i] * p[i] <= INT_MAX;
    tmp = p[i] * p[i];
    //@ assert tmp >= 0;
    //@ assert S + tmp <= INT_MAX;
    S += tmp;
  }
  return S;
}
```

The assertion concerning `tmp` may be discharged automatically by some analyzers, but the other two assertions would require `sqsum`'s contract to be completed with additional pre-conditions. Ideally, the necessary pre-conditions will be infered automatically from the assertions by an analyzer. Even if they are not, and if the pre-conditions need to be written by hand, it is still useful to write the assertions first. In cases like this one, it is easier to get the assertions right than the corresponding pre-conditions. Some analyzers may then be able to check that the assertions are verified under the pre-conditions, providing trust in the latter (In fact, to some analyzers, checking the assertions once the corresponding pre-conditions are provided is much easier than infering the pre-conditions from the assertions).

10.2 Loop Invariants

Another kind of code annotations is dedicated to the analysis of loops. The treatment of loops is a difficult part of static analysis, and many analyzers need to be provided with hints in the form of an *invariant* for each loop. A loop invariant can be seen as a special case of assertion, which is preserved across the loop body. If we look back to the `max_seq` function, a useful invariant for proving that the implementation satisfies the formal specification would be that `res` contains the maximal value seen so far.

Let us now try to formalize this invariant property. Part of the formal invariant that we are trying to build is that at any iteration `j`, the variable `res` is greater or equal to `p[0]`,`p[1]`,...,`p[j]`. This part of the invariant is written:

```
int max_seq(int* p, int n) {
  int res = *p;
  /*@ loop invariant \forall integer j;
        0 <= j < i ==> res >= *(\at(p,Pre)+j); */
  for(int i = 0; i < n; i++) {
    if (res < *p) { res = *p; }
    p++;
  }
  return res;
}
```

We use here the `\at()` construct, which is a generalization of `\old`. Namely, it says that its argument must be evaluated in a given state of the program. A state is represented by a label, which can be a C label (the

corresponding state being the last time this label was reached) or some pre-defined logic labels. `Pre` indicates the pre-state of the function. `\old` is not admitted in loop invariant to avoid confusion with an evaluation at the beginning of the previous loop step.

The other part of the invariant property that should be expressed formally is that there exists an element in `p[0]`,`p[1]`,...,`p[n-1]` that is equal to `res`. In other words, this second part expresses that there exists an integer `e` such that `0 <= e < n` and `p[e] == res`. In order to prove the existence of such an integer `e`, the simplest way is to keep track of the index for which the maximal value is attained. This can be done in ACSL with extra statements called *ghost code*. Ghost code is C code written inside `//@ ghost ..` or `/*@ ghost .. */` comments. The original program must have exactly the same behavior with and without ghost code. In other word, ghost code must not interfere with the concrete implementation. The variables defined and assigned in ghost code (*ghost variables*) can be used in the ACSL properties.

The complete annotated function `max_seq` then becomes:

```
int max_seq(int* p, int n) {
  int res = *p;
  //@ ghost int e = 0;
  /*@ loop invariant \forall integer j;
           0 <= j < i ==> res >= \at(p[j],Pre);
      loop invariant
           \valid(\at(p,Pre)+e) && \at(p,Pre)[e] == res;
      loop invariant 0<=i<=n;
      loop invariant p==\at(p,Pre)+i;
      loop invariant 0<=e<n;
  */
  for(int i = 0; i < n; i++) {
    if (res < *p) {
      res = *p;
      //@ ghost e = i;
    }
    p++;
  }
  return res;
}
```

Bibliography

[1] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language. Preliminary Design, version 1.4*, 2008. http://www.frama-c.cea.fr/download/acsl_1.4.pdf.

[2] Caduceus. http://why.lri.fr/caduceus/.

[3] JML. http://www.cs.iastate.edu/~leavens/JML/.