Software Analyzers

WP Tutorial

WP Tutorial

WP 0.6 for Oxygen-20120901

Patrick Baudin, Loïc Correnson, Philippe Hermann

CEA LIST, Software Safety Laboratory

This work is based on the book "ACSL by Example" published by the Fraunhofer FIRST Institute for the Device-Soft project with the support of the Programme Inter Carnot Fraunhofer from BMBF (Grant 01SF0804) and the ANR.

Contents

Chapter 1

Introduction

This book is a guided tour on how to use WP plug-in of Frama-C for proving C programs annotated with ACSL notations. It is based on the excellent "ACSL By Example" book produced by the Fraunhofer FIRST Institute for the Device-Soft project[1].

We assume the reader to be familiar with ACSL in general and already equipped with the WP plug-in, which is distributed with Frama-C releases. Please refer to the WP user's manual[2] for installation and general overview of the plug-in.

1.1 Library

The case studies presented in this document are exact copies of the ones presented in the original "ACSL By Example" book. Sometimes, we indicate some modifications of the specification that makes WP better prove the programs.

All examples uses the following *libary* of C and ACSL definitions:

File **library.h**

```
#ifndef _LIBRARY_H
#define _LIBRARY_H
#include "library.spec"
#endif
```

1.2 Examples

Source of case studies are generally presented in two separated files: one for the header and specification of the algorithm, and one of its implementation. To ease the presentation in the book, we have omitted the necessary `include` lines from sources.

Thus, a header file `A.h` should be enclosed by the following lines:

```
#ifndef _A_H
#define _A_H
#include "../library.h"
[...]
#endif
```

Similarily, an implementation file `A.c` should start by including its header:

[1] Version 7.1.0 of December 2011, see http://www.first.fraunhofer.de

[2] http://frama-c.com/download/frama-c-wp-manual.pdf

```
#include "A.h"
[...]
```

Chapter 2

Non-Mutating Algorithm

We focus here on the non-mutating algorithm presented in "ACSL By Example", chapter §3.

2.1 The equal Algorithm

This algorithm implements a comparison over two generic sequences. The specification of the algorithm is:

File **equal.h**

```
/*@
  requires IsValidRange(a, n);
  requires IsValidRange(b, n);

  assigns \nothing;

  ensures \result <==> IsEqual{Here,Here}(a, n, b);
*/
bool equal(const value_type* a, size_type n, const value_type* b);
```

The implementation is:

File **equal.c**

```
bool equal(const value_type* a, size_type n, const value_type* b)
{
  /*@
    loop invariant 0 <= i <= n;
    loop invariant \forall int k; 0 <= k < i ==> a[k] == b[k];
    loop assigns i;
    loop variant n-i;
  */
  for (int i = 0; i < n; i++)
    if (a[i] != b[i])
      return 0;
  return 1;
}
```

2.1.1 Correctness

The implementation is proved correct against its specification by simply running the WP plug-in:

```
# frama-c -wp [...]
[kernel] preprocessing with "gcc -C -E -I.  equal/equal.c"
[wp] warning: Missing RTE guards
------------------------------------------------------
Function            #VC    WP  Alt-Ergo    Success
equal                 9     2     7       100% (1s)
------------------------------------------------------
```

The reader should notice the warning emitted by WP. Actually, the plug-in is not responsible for proving the absence of runtime errors during program execution, since other plug-ins can be used for this, for instance *Value Analysis*.

2.1.2 Safety

However, it is still possible to completely prove the program with WP thanks to RTE plug-in. This last plug-in generates assertions in the program wherever a runtime error might occur. Then, the WP plug-in can try to discharge the generated assertions.

This all-together method is easily performed with `wp-rte` option of WP:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  equal/equal.c"
[rte] annotating function equal
------------------------------------------------------
Function            #VC    WP  Alt-Ergo    Success
equal                12     2    10       100% (1s)
------------------------------------------------------
```

In this book, we will always use this technique to prove the correctness and the safety of studied algorithm.

2.2 The mismatch Algorithm

We now present the `mismatch` algorithm that returns the index of the first different element between two sequences, and (-1) otherwise.

File **mismatch.h**

```
/*@
  requires IsValidRange(a, n);
  requires IsValidRange(b, n);

  assigns \nothing;

  behavior all_equal:
    assumes IsEqual{Here,Here}(a, n, b);
    ensures \result == n;

  behavior some_not_equal:
    assumes !IsEqual{Here,Here}(a, n, b);
    ensures 0 <= \result < n;
    ensures a[\result] != b[\result];
    ensures IsEqual{Here,Here}(a, \result, b);

  complete behaviors;
  disjoint behaviors;
*/
size_type mismatch(const value_type* a, size_type n, const value_type* b);
```

The implementation is simply:

File **mismatch.c**

```
size_type mismatch(const value_type* a, size_type n, const value_type* b)
{
  /*@
    loop invariant 0 <= i <= n;
    loop invariant IsEqual{Here,Here}(a, i, b);
    loop assigns i;
    loop variant n-i;
  */
  for (size_type i = 0; i < n; i++)
    if (a[i] != b[i])
      return i;
  return n;
}
```

Once again, WP simply proves the algorithm:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  mismatch/mismatch.c"
[rte] annotating function mismatch
----------------------------------------------------
Function            #VC    WP  Alt-Ergo    Success
mismatch             17     2    15       100% (1s)
----------------------------------------------------
```

2.3 Alternate equal with mismatch

It is also possible to implement the `equal` algorithm in terms of `mismatch`. Using the same specification file given for `equal` in 2.1, the implementation is now:

File **equal.c [alt]**

```
#include "mismatch.h"
#include "../equal/equal.h"
bool equal(const value_type* p, size_type m, const value_type* q)
{
  return mismatch(p, m, q) == m;
}
```

The entire program is proven correct by WP plug-in (here, the proofs steps for function `mismatch` are omitted):

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  mismatch/equal.c"
[rte] annotating function equal
--------------------------------------------------
Function             #VC    WP  Alt-Ergo    Success
equal                  5     2     3       100% (1s)
--------------------------------------------------
```

As the reader may observe, the WP has proven the precondition of `mismatch` from the one of `equal`, and the post-condition of `equal` from the one of `mismatch`.

2.4 The find Algorithm

We study now the *reconsidered* version of the `find` algorithm from “ACSL By Example”. This algorithm looks for the first occurrence of an element into a sequence. We makes use of the following helper predicate:

File **find.h**

```
/*@
  predicate HasValue{A}(value_type* a, integer n, value_type val) =
    \exists integer i; 0 <= i < n && a[i] == val;
*/
```

Then follows the specification of the `find` algorithm:

File **find.h**

```
/*@
  requires IsValidRange(a, n);
  assigns \nothing;

  behavior some:
    assumes HasValue(a, n, val);
    ensures 0 <= \result < n;
    ensures a[\result] == val;
    ensures !HasValue(a, \result, val);

  behavior none:
    assumes !HasValue(a, n, val);
    ensures \result == n;

  complete behaviors;
  disjoint behaviors;
*/
size_type find(const value_type* a, size_type n, value_type val);
```

The implementation of the algorithm is:

File **find.c**

```
size_type find(const value_type* a, size_type n, value_type val)
{
  /*@
    loop invariant 0 <= i <= n;
    loop invariant !HasValue(a, i, val);
    loop assigns i;
    loop variant n-i;
  */
  for (size_type i = 0; i < n; i++)
    if (a[i] == val)
      return i;
  return n;
}
```

The implementation is proved correct against its specification by simply running the WP plug-in:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  find/find.c"
[rte] annotating function find
---------------------------------------------------
Function              #VC    WP  Alt-Ergo    Success
find                   16     2    14       100% (1s)
---------------------------------------------------
```

2.5 The find-first-of Algorithm

This algorithm is an extension of `find`: it looks for the first element of sequence a that belongs to sequence b. We also makes use of the `HasValue` predicate. But we also need its extension to sequence, as in the original specification from "ACSL By Example":

File **findfirst.h**

```
/*@
  predicate HasValueOf{A}(value_type* a, integer m,
                          value_type* b, integer n) =
    \exists integer i; 0 <= i < m &&
      HasValue{A}(b, n, \at(a[i],A));
*/
```

The specification of the algorithm is:

File **findfirst.h**

```
/*@
  requires IsValidRange(a, m);
  requires IsValidRange(b, n);
  assigns \nothing;

  behavior found:
    assumes HasValueOf(a, m, b, n);
    ensures 0 <= \result < m;
    ensures HasValue(b, n, a[\result]);
    ensures !HasValueOf(a, \result, b, n);

  behavior not_found:
    assumes !HasValueOf(a, m, b, n);
    ensures \result == m;

  complete behaviors;
  disjoint behaviors;
*/
size_type find_first_of(const value_type* a, size_type m,
                        const value_type* b, size_type n);
```

The implementation of the algorithm is:

File **findfirst.c**

```
size_type find_first_of(const value_type* a, size_type m,
                        const value_type* b, size_type n)
{
  /*@
    loop invariant 0 <= i <= m;
    loop invariant !HasValueOf(a, i, b, n);
    loop assigns i;
    loop variant m-i;
   */
  for (size_type i = 0; i < m; i++)
    if (find(b, n, a[i]) < n)
      return i;
  return m;
}
```

The implementation is proved correct against its specification by simply running the WP plug-in:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  findfirst/findfirst.c"
[rte] annotating function find_first_of
--------------------------------------------------
Function           #VC   WP  Alt-Ergo   Success
find_first_of       20    3   17        100% (1s)
--------------------------------------------------
```

2.6 The **adjacent-find** Algorithm

This algorithm looks for the first two consecutive equal elements in a sequence. The formal definition of equal neighbors is:

File **adjacent.h**

```
/*@
  predicate HasEqualNeighbors{A}(value_type* a, integer n) =
    \exists integer i; 0 <= i < n-1 && a[i] == a[i+1];
*/
```

The specification of the algorithm is:

File **adjacent.h**

```
/*@
  requires IsValidRange(a, n);
  assigns \nothing;

  behavior some:
    assumes HasEqualNeighbors(a, n);
    ensures 0 <= \result < n-1 ;
    ensures a[\result] == a[\result+1];
    ensures !HasEqualNeighbors(a, \result);

  behavior none:
    assumes !HasEqualNeighbors(a, n);
    ensures \result == n;

  complete behaviors;
  disjoint behaviors;
*/
size_type adjacent_find(const value_type* a, size_type n) ;
```

The implementation of the algorithm is:

File **adjacent.c**

```
size_type adjacent_find(const value_type* a, size_type n)
{
  if (0==n) return n ;

  /*@
    loop invariant 0 <= i < n;
    loop invariant !HasEqualNeighbors(a, i+1);
    loop assigns i;
    loop variant n-i;
  */
  for (size_type i = 0; i < n-1; i++)
    if (a[i] == a[i+1])
      return i;
  return n;
}
```

The implementation is proved correct against its specification by simply running the WP plug-in:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  adjacent/adjacent.c"
[rte] annotating function adjacent_find
--------------------------------------------------
Function              #VC    WP  Alt-Ergo    Success
adjacent_find          19     2    17        100% (1s)
--------------------------------------------------
```

2.7 The count Algorithm

The algorithm presented here counts the number of occurences for a given element in a sequence. The axiomatization of counting occurrences proposed in “ACSL By Example” is:

File **count.axioms**

```
/*@
  axiomatic Counting
  {
    logic integer Count{L}(value_type* a, value_type val,
                           integer i, integer j)
      reads a[i..j-1];

    axiom Count0{L}:
      \forall value_type *a, v, integer i;
        Count(a, v, i, i) == 0;

    axiom Count1{L}:
      \forall value_type* a, v, integer i, integer i, j, k;
        i <= j <= k ==> Count(a, v, i, k) ==
                        Count(a, v, i, j) + Count(a, v, j, k);

    axiom Count2{L}:
      \forall value_type* a, v, integer i;
        (a[i] != v ==> Count(a, v, i, i+1) == 0) &&
        (a[i] == v ==> Count(a, v, i, i+1) == 1) ;
  }
*/
```

The predicate `Count` is defined by a `read` clause. The current version of WP plug-in only implements a limited subset of `read` clauses. Hence, we must adapt a bit this specification.

2.7.1 Adapting the Axiomatics

To be handled correctly, we must provide WP with enough *patterns* in `read` clauses to access all the heap values actually *reads*.

For the memory models used in this case study, it is sufficient to access at least one element of the sequence to properly define the `Count` predicate. Hence, we modified the `Count` axiomatic as follows:

File **count.axioms [adapted]**

```
/*@
  axiomatic Counting
  {
    logic integer Count{L}(value_type* a, value_type val,
                           integer i, integer j)
      reads a[i],a[j-1];

    axiom Count0{L}:
      \forall value_type *a, v, integer i;
        Count(a, v, i, i) == 0;

    axiom Count1{L}:
      \forall value_type* a, v, integer i, integer i, j, k;
        i <= j <= k ==> Count(a, v, i, k) ==
                        Count(a, v, i, j) + Count(a, v, j, k);

    axiom Count2{L}:
      \forall value_type* a, v, integer i;
        (a[i] != v ==> Count(a, v, i, i+1) == 0) &&
        (a[i] == v ==> Count(a, v, i, i+1) == 1) ;
  }
*/
```

This problem will be solved in future release of WP since there is no theoretical limitation for handling arbitrary *reads* definitions.

2.7.2 Proving Lemmas

From the original `Count` axiomatics, the following lemma should hold:

File **count.lemma**

```
/*@
  lemma CountLemma: \forall value_type *a, v, integer i;
     0 <= i ==> Count(a, v, 0, i+1) ==
                Count(a, v, 0, i) + Count(a, v, i, i+1);
*/
```

Unfortunately, the current version of WP plug-in does not generate proof obligation for lemmas. We can, however, seek a proof for the lemma by embedding it into the post-condition of a function that does nothing. For instance:

File **count/lemma.c**

```
#include "../library.h"
#include "count.axioms"

/*@ ensures \forall value_type *a, v, integer i;
     0 <= i ==> Count(a, v, 0, i+1) ==
                Count(a, v, 0, i) + Count(a, v, i, i+1);
  */
void lemma(void)
{ }
```

The lemma is easily proved in this form:

```
# frama-c -wp [...]
[kernel] preprocessing with "gcc -C -E -I.  count/lemma.c"
[wp] warning: Missing RTE guards
[wp] warning: Interpreting reads-definition as expressions rather than tsets
------------------------------------------------------
```

```
Function            #VC   WP  Alt-Ergo   Success
lemma                 1    -      1       100% (1s)
---------------------------------------------------
```

Future version of WP plug-in will automatically generate the necessary proof obligation for lemmas, with no more need for such *dummy* functions.

2.7.3 Proving the Algorithm

Now that we have axiomatized the `Count` predicate, we may study the algorithm proposed in "ACSL By Example" for counting the occurrences of a a given element in a sequence.

The specification, which also includes the adapted axiomatics and the lemma of previous sections, is as follows:

File **count.h**

```
/*@
  requires IsValidRange(a, n);

  ensures \result == Count(a, val, 0, n);

  assigns \nothing;
*/
size_type count(const value_type* a, size_type n, value_type val) ;
```

Then follows the specification of the `count` algorithm:

File **count.c**

```
size_type count(const value_type* a, size_type n, value_type val)
{
  size_type cnt = 0 ;
  /*@
    loop invariant 0 <= i <= n;
    loop invariant 0 <= cnt <= i;
    loop invariant cnt == Count(a, val, 0, i);
    loop assigns i,cnt;
    loop variant n-i;
  */
  for (size_type i = 0; i < n; i++)
    if (a[i] == val)
      cnt++;
  return cnt;
}
```

The algorithm is proved correct against its specification by simply running the WP plug-in:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  count/count.c"
count/count.lemma:2:[wp] warning: Proof obligation for lemma 'CountLemma' not generated.
[rte] annotating function count
[wp] warning: Interpreting reads-definition as expressions rather than tsets
---------------------------------------------------
Function            #VC   WP  Alt-Ergo   Success
count                14    3    11        100% (1s)
---------------------------------------------------
```

2.8 The search Algorithm

We study now the `search` algorithm from "ACSL By Example". This algorithm looks for a subsequence inside a sequence. We use the same definition as in the original specification:

File search.h

```
/*@
  predicate HasSubRange{A}(value_type* a, integer m,
                           value_type* b, integer n) =
    \exists size_type k; (0 <= k <= m-n) && IsEqual{A,A}(a+k, n, b);
*/
```

Then follows the specification of the `search` algorithm:

File search.h

```
/*@
  requires IsValidRange(a, m);
  requires IsValidRange(b, n);
  assigns \nothing;

  behavior has_match:
    assumes HasSubRange(a, m, b, n);
    ensures (n == 0 || m ==0) ==> \result == 0;
    ensures 0 <= \result <= m-n;
    ensures IsEqual{Here,Here}(a+\result, n, b);
    ensures !HasSubRange(a, \result+n-1, b, n);

  behavior no_match:
    assumes !HasSubRange(a, m, b, n);
    ensures \result == m;

  complete behaviors;
  disjoint behaviors;
*/
size_type search(const value_type* a, size_type m, const value_type* b, size_type n);
```

The implementation of the algorithm is:

File search.c

```
#include "../equal/equal.h"

size_type search(const value_type* a, size_type m, const value_type* b, size_type n)
{
  if ((n == 0) || (m == 0)) return 0;
  if (n > m) return m;
  /*@
     loop invariant 0 <= i <= m-n+1;
     loop invariant !HasSubRange(a, n+i-1, b, n);
     loop assigns i;
     loop variant m-i;
  */
  for(size_type i = 0; i <= m-n; i++) {
    if (equal(a+i, n, b)) // Is there a match?
      return i;
  }
  return m;
}
```

Remark that `search` is defined in terms of `equal`, and its specification should be included from the `equal.h` header file.

From the original specification from "ACSL By Example", we only add the *loop assigns* clause. The implementation is proved correct against its specification by simply running the WP plug-in:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  search/search.c"
[rte] annotating function search
-----------------------------------------------------
Function             #VC   WP  Alt-Ergo   Success
search                24    3   21       100% (1s)
-----------------------------------------------------
```

Chapter 3

Maximum and Minimum Algorithms

In this chapter we study the algorithms for computing the extremum elements in a sequence.

3.1 Partial Order Properties

In the case study, the partial order `<` is used with integer types only. With decision procedures supported by WP plug-in, the *irrreflexivity*, *antisymmetry* and *transitivity* properties naturally holds and are not necessary to be included in the specifications.

However, for *genericity* purpose, we can still manage to prove them using the same syntactic sugar already used for `count` lemma in section 2.7.

For `<` we introduce the following dummy declaration:

File **less.lemma**

```
/*@
  ensures LessIrreflexivity:
    \forall value_type a; !(a < a);
  ensures LessAntisymetry:
    \forall value_type a, b; (a < b) ==> !(b < a);
  ensures LessTransitivity:
    \forall value_type a, b, c; (a < b) && (b < c) ==> (a < c);
*/
void less(void) { }
```

For `>`, `<=` and `>=`, we introduce the following dummy declaration:

File **greater.lemma**

```
/*@
  ensures Greater:
    \forall value_type a, b; (a > b) <==> (b < a);
  ensures LessOrEqual:
    \forall value_type a, b; (a <= b) <==> !(b < a);
  ensures GreaterOrEqual:
    \forall value_type a, b; (a >= b) <==> !(a < b);
*/
void greater(void) { }
```

All these lemmas are easily discharged:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  compare/greater.c"
[kernel] preprocessing with "gcc -C -E -I.  compare/less.c"
[rte] annotating function greater
[rte] annotating function less
------------------------------------------------------
Function            #VC    WP  Alt-Ergo    Success
greater               3    -      3       100% (1s)
less                  3     1     2       100% (1s)
------------------------------------------------------
```

3.2 The max-element Algorithm

We study now the first version of the `max-element` algorithm from "ACSL By Example". The specification of the `max-element` algorithm is as follows:

File **maxelt.h**

```
/*@
  requires IsValidRange(a, n);
  assigns \nothing;

  behavior empty:
    assumes n == 0;
    ensures \result == 0;

  behavior not_empty:
    assumes 0 < n;
    ensures 0 <= \result < n;
    ensures \forall integer i; 0 <= i < n ==> a[i] <= a[\result];
    ensures \forall integer i; 0 <= i < \result ==> a[i] < a[\result];

  complete behaviors;
  disjoint behaviors;
*/
size_type max_element(const value_type* a, size_type n);
```

The implementation of the algorithm is:

File **maxelt.c**

```
size_type max_element(const value_type* a, size_type n)
{
  if (n == 0) return 0;

  size_type max = 0;
  /*@
    loop invariant 0 <= i <= n;
    loop invariant 0 <= max < n;
    loop invariant \forall integer k; 0 <= k < i ==> a[k] <= a[max];
    loop invariant \forall integer k; 0 <= k < max ==> a[k] < a[max];
    loop assigns max,i;
    loop variant n-i;
   */
  for (size_type i = 0; i < n; i++)
    if (a[max] < a[i])
      max = i;

  return max;
}
```

The implementation is proved correct against its specification by simply running the WP plug-in:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  maxelt/maxelt.c"
[rte] annotating function max_element
--------------------------------------------------
Function            #VC    WP  Alt-Ergo   Success
max_element          21     3    18       100% (1s)
--------------------------------------------------
```

3.3 The max-element Algorithm with Predicates

We study now the *reconsidered* version of the `max-element` algorithm from “ACSL By Example” with predicates:

File **maximum.spec**

```
/*@
  predicate IsMaximum{L}(value_type* a, integer n, integer max) =
    !(\exists integer i; 0 <= i < n && (a[max] < a[i]));

  predicate IsFirstMaximum{L}(value_type* a, integer max) =
    \forall integer i; 0 <= i < max ==> a[i] < a[max];
*/
```

Remark that we rephrased the original specification of `IsMaximum`. The specification of the `max-element` algorithm becomes:

File **maxelt.h**

```
/*@
  requires IsValidRange(a, n);
  assigns \nothing;

  behavior empty:
    assumes n == 0;
    ensures \result == 0;

  behavior not_empty:
    assumes 0 < n;
    ensures 0 <= \result < n;
    ensures IsMaximum(a, n, \result);
    ensures IsFirstMaximum(a, \result);

  complete behaviors; disjoint behaviors;
*/
size_type max_element(const value_type* a, size_type n);
```

The implementation of the algorithm is now:

File **maxelt.c**

```
size_type max_element(const value_type* a, size_type n)
{
  if (n == 0) return 0;

  size_type max = 0;
  /*@
    loop invariant 0 <= i <= n;
    loop invariant 0 <= max < n;
    loop invariant IsMaximum(a, i, max);
    loop invariant IsFirstMaximum(a, max);
    loop assigns i,max;
    loop variant n-i;
   */
  for (size_type i = 0; i < n; i++)
    if (a[max] < a[i])
      max = i;

  return max;
}
```

The implementation is proved correct against its specification by running the WP plug-in:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  maxeltp/maxelt.c"
[rte] annotating function max_element
------------------------------------------------------
Function              #VC    WP  Alt-Ergo    Success
max_element            21     3    18       100% (1s)
------------------------------------------------------
```

3.4 The max-seq Algorithm

We study now the `max-seq` algorithm from "ACSL By Example". Its specification is to return the maximal value in a sequence, not its index like in the `max-element` algorithm.

The specification is as follows:

File **maxseq.h**

```
/*@
  requires n>0 ;
  requires IsValidRange(a, n);
  assigns \nothing;

  ensures \forall integer i; 0 <= i <= n-1 ==> \result >= a[i];
  ensures \exists integer e; 0 <= e <= n-1 && \result == a[e];
*/
size_type max_seq(const value_type* a, size_type n);
```

The algorithm is implemented in terms of algorithm `max-element`, as follows:

File **maxseq.c**

```
size_type max_seq(const value_type* a, size_type n)
{
  return a[max_element(a,n)];
}
```

The implementation is proved correct against its specification thanks to the specification of the `max-element` function:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  maxseq/maxseq.c"
[rte] annotating function max_seq
------------------------------------------------------
Function              #VC    WP  Alt-Ergo    Success
max_seq                 6     2     4       100% (1s)
------------------------------------------------------
```

3.5 The min-element Algorithm

We study now the first version of the `min-element` algorithm from “ACSL By Example”. The specification of the `min-element` algorithm:

File **minelt.h**

```
/*@
  requires IsValidRange(a, n);
  assigns \nothing;

  behavior empty:
    assumes n == 0;
    ensures \result == 0;

  behavior not_empty:
    assumes 0 < n;
    ensures 0 <= \result < n;
    ensures \forall integer i; 0 <= i < n ==> a[\result] <= a[i];
    ensures \forall integer i; 0 <= i < \result ==> a[\result] < a[i];

  complete behaviors;
  disjoint behaviors;
*/
size_type min_element(const value_type* a, size_type n);
```

The implementation of the algorithm is:

File **minelt.c**

```
size_type min_element(const value_type* a, size_type n)
{
  if (n == 0) return 0;

  size_type min = 0;
  /*@
    loop invariant 0 <= i <= n;
    loop invariant 0 <= min < n;
    loop invariant \forall integer k; 0 <= k < i   ==> a[min] <= a[k];
    loop invariant \forall integer k; 0 <= k < min ==> a[min] <  a[k];
    loop assigns i,min;
    loop variant n-i;
   */
  for (size_type i = 0; i < n; i++)
    if (a[i] < a[min])
      min = i;

  return min;
}
```

The implementation is proved correct against its specification by simply running the WP plug-in:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  minelt/minelt.c"
[rte] annotating function min_element
---------------------------------------------------
Function             #VC    WP  Alt-Ergo    Success
min_element           21     3    18       100% (1s)
---------------------------------------------------
```

Chapter 4

Binary Search Algorithms

In this chapter, we study the binary search algorithms defined in "ACSL By Example".

4.1 Specification Helpers

Like in the original book, we must introduce in our library the definition for sorted sequences:

File **binary.h**

```
/*@
  predicate IsSorted{L}(value_type* a, integer n) =
    \forall integer i,j; 0<=i<j<n ==> a[i]<=a[j];
*/
```

We also require the introduction of an helper axiom for division:

File **binary.h**

```
/*@
  axiomatic Division {
    predicate NonNegative(integer i) = (0 <= i);
    axiom div: \forall integer i; NonNegative(i) ==> 0 <= 2*(i/2) <= i;
  }
*/
```

4.2 By-Reference Arguments

The generated proof obligations for lowerbound and upperbound functions are rather difficult to discharge with the `Store` memory model. It is better here to use the `Logic` memory model that combines Store and arrays passed by references detection, which is quite effective here.

This experimental memory model is activated by `-wp-model Logic`. However, the proofs can be handled in `Store` model with a timeout of one minute.

4.3 The lower-bound algorithm

We study here the `lower-bound` algorithm from "ACSL By Example". Its specification is:

File lowerbound.h

```
/*@
  requires IsValidRange(a, n);
  requires IsSorted(a, n);

  assigns \nothing;

  ensures 0 <= \result <= n;
  ensures \forall integer k; 0 <= k < \result ==> a[k] < val;
  ensures \forall integer k; \result <= k < n ==> val <= a[k];
*/
size_type lower_bound(const value_type* a, size_type n, value_type val);
```

The implementation of the algorithm is:

File lowerbound.c

```
size_type lower_bound(const value_type *a, size_type n, value_type val)
{
  size_type left = 0;
  size_type right = n;
  size_type middle = 0;

  /*@
    loop invariant 0 <= left <= right <= n;
    loop invariant \forall integer i; 0 <= i < left ==> a[i] < val;
    loop invariant \forall integer i; right <= i < n ==> val <= a[i];
    loop assigns middle,left,right;
    loop variant right - left;
  */
  while (left < right) {
    middle = left + (right - left) / 2;
    if (a[middle] < val)
      left = middle + 1;
    else
      right = middle;
  }

  return left;
}
```

Remark: the original specification of loop-assigns in "ACSL By Example" is *wrong* since the loop *do* assigns the local variables of the function. The original specification works with Jessie since local variables rarely live in the assignable heap, but it is not correct for WP where assigns clauses are more strictly verified.

The implementation is proved correct against its specification by simply running the WP plug-in:

```
# frama-c -wp -wp-rte -wp-model Logic [...]
[kernel] preprocessing with "gcc -C -E -I.  lowerbound/lowerbound.c"
[rte] annotating function lower_bound
------------------------------------------------------
Function            #VC    WP  Alt-Ergo    Success
lower_bound          17     2   15        100% (5s)
------------------------------------------------------
```

4.4 The upper-bound algorithm

We study here the `upper-bound` algorithm from "ACSL By Example". Its specification is:

File **upperbound.h**

```
/*@
  requires IsValidRange(a, n);
  requires IsSorted(a, n);

  assigns \nothing;

  ensures 0 <= \result <= n;
  ensures \forall integer k; 0 <= k < \result ==> a[k] <= val;
  ensures \forall integer k; \result <= k < n ==> val < a[k];
*/
size_type upper_bound(const value_type* a, size_type n, value_type val);
```

The implementation of the algorithm is:

File **upperbound.c**

```
size_type upper_bound(const value_type *a, size_type n, value_type val)
{
  size_type left = 0;
  size_type right = n;
  size_type middle = 0;

  /*@
    loop invariant 0 <= left <= right <= n;
    loop invariant \forall integer i; 0 <= i < left ==> a[i] <= val;
    loop invariant \forall integer i; right <= i < n ==> val < a[i];
    loop assigns middle,left,right;
    loop variant right - left;
  */
  while (left < right) {
    middle = left + (right - left) / 2;
    if (a[middle] <= val)
      left = middle + 1;
    else
      right = middle;
  }

  return right;
}
```

The implementation is proved correct against its specification by simply running the WP plug-in:

```
# frama-c -wp -wp-rte -wp-model Logic [...]
[kernel] preprocessing with "gcc -C -E -I.  upperbound/upperbound.c"
[rte] annotating function upper_bound
------------------------------------------------
Function          #VC   WP  Alt-Ergo   Success
upper_bound        17    2    15       100% (3s)
------------------------------------------------
```

4.5 The binary-search algorithm

We study here the `binary-search` algorithm from "ACSL By Example". Its specification is:

File **binarysearch.h**

```
/*@
  requires IsValidRange(a, n);
  requires IsSorted(a, n);

  assigns \nothing;

  ensures \result <==> HasValue(a, n, val);
*/
bool binary_search(const value_type* a, size_type n, value_type val);
```

The implementation of the algorithm is:

File **binarysearch.c**

```
bool binary_search(const value_type* a, size_type n, value_type val)
{
  size_type lwb = lower_bound(a, n, val);
  return lwb < n && a[lwb] <= val;
}
```

Remark: the original specification of loop-assigns in "ACSL By Example" is *wrong* since the loop *do* assigns the local variables of the function. The original specification works with Jessie since local variables rarely live in the assignable heap, but it is not correct for WP where assigns clauses are more strictly verified.

The implementation is proved correct against its specification by simply running the WP plug-in:

```
# frama-c -wp -wp-rte -wp-model Logic [...]
[kernel] preprocessing with "gcc -C -E -I.  binarysearch/binarysearch.c"
[rte] annotating function binary_search
--------------------------------------------------
Function            #VC    WP  Alt-Ergo    Success
binary_search         6     2     4       100% (1s)
--------------------------------------------------
```

Chapter 5

Mutating Algorithms

In this chapter, we study the mutating algorithms defined in "ACSL By Example".

5.1 The fill Algorithm

We now study the `fill` algorithm from "ACSL By Example". This algorithm initializes a sequence with a particular value.

The specification of the `fill` algorithm is as follows:

File **fill.h**

```
/*@
  requires IsValidRange(a, n);

  assigns a[0..n-1];

  ensures \forall integer i; 0 <= i < n ==> a[i] == val;
*/
void fill(value_type* a, size_type n, value_type val);
```

The implementation of the algorithm is:

File **fill.c**

```
void fill(value_type* a, size_type n, value_type val)
{
  /*@
    loop invariant 0 <= i <= n;
    loop invariant \forall integer k; 0 <= k < i ==> a[k] == val;
    loop assigns a[0..n-1], i;
    loop variant n-i;
  */
  for (size_type i = 0 ; i < n ; i++)
    a[i] = val;
}
```

From the original specification from "ACSL By Example", we only add the *loop assigns* clause. The implementation is proved correct against its specification by simply running the WP plug-in:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  fill/fill.c"
[rte] annotating function fill
---------------------------------------------------
Function            #VC    WP  Alt-Ergo    Success
fill                 11     1    10       100% (1s)
---------------------------------------------------
```

5.2 The iota Algorithm

We now study the `iota` algorithm from "ACSL By Example". This algorithm assigns sequentially increasing values to a range, with a user-defined start value.

We slightly modify the specification of the `iota` algorithm to replace the `INT_MAX` macro-definition by its value on 32-bit architectures:

File **iota.h**

```
/*@
  requires IsValidRange(a, n);
  requires val + n < 2147483647 ; // INT_MAX

  assigns a[0..n-1];

  ensures \forall integer k; 0 <= k < n ==> a[k] == val + k;
*/
void iota(value_type* a, size_type n, value_type val);
```

The implementation of the algorithm is:

File **iota.c**

```
void iota(value_type* a, size_type n, value_type val)
{
  /*@

    loop invariant 0 <= i <= n;
    loop assigns i, a[0..i-1];

    loop invariant \forall integer k; 0 <= k < i
                      ==> a[k] == val + k;
    loop variant n-i;
  */
  for (size_type i = 0; i < n; i++)
    a[i] = val + i;
}
```

The implementation is proved correct against its specification by running the WP plug-in:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  iota/iota.c"
[rte] annotating function iota
--------------------------------------------------
Function            #VC    WP  Alt-Ergo    Success
iota                 13    -      13       100% (1s)
--------------------------------------------------
```

5.3 The swap Algorithm

We now study the `swap` algorithm from "ACSL By Example". This algorithm exchanges the value of two variables.

The specification of the `swap` algorithm is as follows:

File **swap.h**

```
/*@
  requires \valid(p);
  requires \valid(q);
  requires \separated(p,q);

  assigns *p;
  assigns *q;

  ensures *p == \old(*q);
  ensures *q == \old(*p);
*/
void swap(value_type* p, value_type* q);
```

The implementation of the algorithm is:

File **swap.c**

```
void swap(value_type* p, value_type* q) {
  const value_type save = *p;
  *p = *q;
  *q = save;
}
```

The implementation is proved correct against its specification by simply running the WP plug-in:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  swap/swap.c"
[rte] annotating function swap
--------------------------------------------------
Function            #VC    WP  Alt-Ergo    Success
swap                  7     1     6       100% (1s)
--------------------------------------------------
```

5.4 The swap-values Algorithm

We now study the `swap-values` algorithm from "ACSL By Example". This algorithm exchanges the value of two variables.

The specification of the `swap-values` algorithm is as follows:

File swapvalues.h

```
/*@
  predicate
    SwapValues{L1,L2}(value_type* a, size_type i, size_type j) =
      0 <= i && 0 <= j &&
      \at(a[i],L1) == \at(a[j],L2) &&
      \at(a[j],L1) == \at(a[i],L2) &&
      (\forall integer k; 0 <= k && k != i && k != j ==>
          \at(a[k],L1) == \at(a[k],L2));
*/

/*@
  requires IsValidRange(a, n);
  requires 0 <= i < n;
  requires 0 <= j < n;

  assigns a[i];
  assigns a[j];

  ensures SwapValues{Old,Here}(a, i, j);
*/
void swap_values(value_type* a, size_type n,
                 size_type   i, size_type j);
```

The implementation of the algorithm is:

File swapvalues.c

```
void swap_values(value_type* a, size_type n,
                 size_type   i, size_type j) {
  value_type tmp = a[i];
  a[i] = a[j];
  a[j] = tmp;
}
```

The implementation is proved correct with detection of arrays passed by reference (option `-wp-model Logic`):

```
# frama-c -wp -wp-rte -wp-model Logic [...]
[kernel] preprocessing with "gcc -C -E -I.  swapvalues/swapvalues.c"
[rte] annotating function swap_values
--------------------------------------------------
Function            #VC    WP  Alt-Ergo    Success
swap_values           6     1     5       100% (1s)
--------------------------------------------------
```

With `Store` memory model, the proof obligations are not discharged by Alt-Ergo:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  swapvalues/swapvalues-withassert.c"
[rte] annotating function swap_values
--------------------------------------------------
Function            #VC    WP  Alt-Ergo    Success
swap_values           6     1     4      83.3% (1s)
--------------------------------------------------
```

5.5 The swap-ranges Algorithm

We now study the `swap-ranges` algorithm from "ACSL By Example". This algorithm exchanges the contents of two ranges element-wise. The specification of the `swap-ranges` algorithm is as follows:

File **swapranges.h**

```
/*@
  requires IsValidRange(a, n);
  requires IsValidRange(b, n);
  requires \separated(a+(0..n-1), b+(0..n-1));

  assigns a[0..n-1];
  assigns b[0..n-1];

  ensures IsEqual{Here,Old}(a, n, b);
  ensures IsEqual{Old,Here}(a, n, b);
*/
void swap_ranges(value_type* a, size_type n, value_type* b);
```

The implementation of the algorithm is:

File **swapranges.c**

```
void swap_ranges(value_type* a, size_type n, value_type* b) {
  /*@
    loop invariant 0 <= i <= n;

    loop assigns a[0..i-1];
    loop assigns b[0..i-1];
    loop assigns i;

    loop invariant IsEqual{Pre,Here}(a, i, b);
    loop invariant IsEqual{Here,Pre}(a, i, b);
    loop variant n-i;
  */
  for (size_type i = 0 ; i < n ; i++) {
    swap(&a[i], &b[i]);
  }
}
```

Within `Store` model, the preservation of loop invariants are hardly discharged by Alt-Ergo. The implementation is proven correct with model `Logic` that takes benefit from arrays passed by reference:

```
# frama-c -wp -wp-rte -wp-model Logic [...]
[kernel] preprocessing with "gcc -C -E -I.  swapranges/swapranges.c"
[rte] annotating function swap_ranges
--------------------------------------------------
Function            #VC   WP  Alt-Ergo   Success
swap_ranges          24    1    23       100% (1s)
--------------------------------------------------
```

5.6 The copy Algorithm

We now study the `copy` algorithm from “ACSL By Example”. This algorithm copies the contents from a source sequence to a destination sequence.

The specification of the `copy` algorithm is as follows:

File **copy.h**

```
/*@
  requires IsValidRange(a, n);
  requires IsValidRange(b, n);
  requires \separated(a+(0..n−1), b+(0..n−1));

  assigns b[0..n−1];

  ensures IsEqual{Here,Here}(a, n, b);

*/
void copy(const value_type* a, size_type n, value_type* b);
```

The implementation of the algorithm is:

File **copy.c**

```
void copy(const value_type* a, size_type n, value_type* b) {
  /*@
    loop invariant 0 <= i <= n;

    loop assigns i,b[0..i−1];

    loop invariant IsEqual{Here,Here}(a, i, b);
    loop variant n−i;
  */
  for (size_type i = 0; i < n; i++)
    b[i] = a[i];
}
```

The implementation is proved correct against its specification by simply running the WP plug-in:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  copy/copy.c"
[rte] annotating function copy
------------------------------------------------------
Function            #VC    WP  Alt-Ergo    Success
copy                 13    -     13       100% (3s)
------------------------------------------------------
```

5.7 The reverse-copy and reverse Algorithms

5.7.1 reverse-copy

We now study the `reverse-copy` algorithm from “ACSL By Example”. This algorithm copies the contents from a source sequence to a destination sequence in reverse order.

The specification of the `reverse-copy` algorithm is as follows:

File **reversecopy.h**

```
/*@
  requires IsValidRange(a, n);
  requires IsValidRange(b, n);

  assigns b[0..(n-1)];

  ensures \forall integer i; 0 <= i < n ==> b[i] == a[n-1-i];
*/
void reverse_copy(const value_type* a, size_type n, value_type* b);
```

The implementation of the algorithm is:

File **reversecopy.c**

```
void reverse_copy(const value_type* a, size_type n, value_type* b) {
  /*@
    loop invariant 0 <= i <= n;
    loop invariant \forall integer k; 0 <= k < i ==>
                     b[k] == a[n-1-k];

    loop assigns b[0..i-1], i ;
    loop variant n-i;
  */
  for (size_type i = 0; i < n; i++)
    b[i] = a[n-1-i];
}
```

The implementation is proved correct against its specification by simply running the WP plug-in:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  reversecopy/reversecopy.c"
[rte] annotating function reverse_copy
--------------------------------------------------
Function              #VC    WP  Alt-Ergo    Success
reverse_copy           15    -      15       100% (1s)
--------------------------------------------------
```

5.7.2 reverse

The `reverse` algorithm is an in place version of the `reverse-copy` algorithm.

Its specification is as follows:

File **reverse.h**

```
/*@
  requires IsValidRange(a, n);

  assigns a[0..(n-1)];

  ensures \forall integer i; 0 <= i < n ==>
            a[i] == \old(a[n-1-i]);
*/
void reverse(value_type* a, size_type n);
```

The implementation is:

File **reverse.c**

```
void reverse(value_type* a, size_type n) {
  size_type first = 0;
  size_type last = n-1;

  /*@
    loop invariant 0 <= first;
    loop invariant last < n;
     // next 2 are added for proving "normal" assigns with alt-ergo
    loop invariant n > 0 ==> first <= n ;
    loop invariant n <= 0 ==> first == 0;

     // false, though simplify is ok with it ...
     //  loop invariant n > 0 ==> first <= last;

    loop invariant first + last == n - 1;

    loop invariant \forall integer k;
      0 <= k < first ==> a[k] == \at(a[n-1-k], Pre);

    loop invariant \forall integer k;
      last < k < n ==> a[k] == \at(a[n-1-k], Pre);

    loop assigns a[0..(first-1)];
    loop assigns a[(last+1)..(n-1)];
    loop assigns first, last;

    loop variant last;
  */
  while (first < last) {
    swap_values(a, n, first++, last--);
  }

}
```

Running the WP plug-in does not allow proving some loop invariants preservation nor certain loop assigns:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  reversecopy/reverse.c"
[rte] annotating function reverse
--------------------------------------------------
Function              #VC    WP  Alt-Ergo    Success
reverse                33     2    29      93.9% (1s)
--------------------------------------------------
```

5.8 The rotate-copy Algorithm

We now study the `rotate-copy` algorithm from “ACSL By Example”. This algorithm rotates a source sequence by a certain number of positions and copies the result to a destination sequence of same size.

The specification of the `rotate-copy` algorithm is as follows:

File **rotatecopy.h**

```
/*@
  requires IsValidRange(a, n);
  requires IsValidRange(b, n);
  requires \separated(a+(0..n-1), b+(0..n-1));
  requires 0 <= m <= n;

  assigns b[0..(n-1)];
```

```
  ensures IsEqual{Here,Here}(a, m, b+(n-m));
  ensures IsEqual{Here,Here}(a+m, n-m, b);
*/
void rotate_copy(const value_type* a, size_type m, size_type n,
                 value_type* b);
```

The implementation of the algorithm is:

File **rotatecopy.c**

```
void rotate_copy(const value_type* a, size_type m, size_type n,
                 value_type* b) {
  copy(a, m, b+(n-m));
  copy(a+m, n-m, b);
}
```

A partial proof of correctness is obtained by running the WP plug-in:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  rotatecopy/rotatecopy.c"
[rte] annotating function rotate_copy
---------------------------------------------------
Function           #VC   WP  Alt-Ergo    Success
rotate_copy         15    -     14      93.3% (3s)
---------------------------------------------------
```

5.9 The **replace-copy** Algorithm

We now study the `replace-copy` algorithm from "ACSL By Example". This algorithm copies a source sequence to a destination sequence whilst substituting every occurrence of a given value by another value.

The specification of the `replace-copy` algorithm is as follows:

File **replacecopy.h**

```
/*@
  requires IsValidRange(a, n);
  requires IsValidRange(b, n);
  requires \separated(a+(0..n-1), b+(0..n-1));

  assigns b[0..(n-1)];

  ensures \forall integer j; 0 <= j < n ==>
          (a[j] == old_val && b[j] == new_val) ||
          (a[j] != old_val && b[j] == a[j]);
  ensures \result == n;
*/
size_type replace_copy(const value_type* a, size_type n,
                       value_type* b,
                       value_type old_val, value_type new_val);
```

The implementation of the algorithm is:

File **replacecopy.c**

```
size_type replace_copy(const value_type* a, size_type n,
                       value_type* b,
                       value_type old_val, value_type new_val)
{
  /*@
    loop invariant 0 <= i <= n;

    loop assigns i,b[0..i-1];

    loop invariant \forall integer j; 0 <= j < i ==>
                (a[j] == old_val && b[j] == new_val) ||
```

```
                        (a[j] != old_val && b[j] == a[j]);
      loop variant n-i;
    */
    for(size_type i = 0; i < n; i++)
      b[i] = (a[i] == old_val ? new_val : a[i]);

    return n;
}
```

A partial proof of correctness is obtained by running the WP plug-in:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  replacecopy/replacecopy.c"
[rte] annotating function replace_copy
------------------------------------------------------
Function             #VC    WP  Alt-Ergo    Success
replace_copy          18     1    16      94.4% (1s)
------------------------------------------------------
```

5.10 The remove-copy Algorithm

We now study the `remove-copy` algorithm from "ACSL By Example". This algorithm copies the contents from a source sequence to a destination sequence whilst removing elements having a given value. The return value is the length of the range.

5.10.1 Adapting the Axiomatics

The axiomatization provided in "ACSL By Example" is:

File **remove-copy.axioms**

```
/*@
  axiomatic WhitherRemove_Function
  {
    logic integer WhitherRemove{L}(value_type* a, value_type v,
                        integer i) reads a[0..(i-1)];

    axiom whither_1:
      \forall value_type *a, v; WhitherRemove(a, v, 0) == 0;

    axiom whither_2:
      \forall value_type *a, v, integer i; a[i] == v ==>
        WhitherRemove(a, v, i+1) == WhitherRemove(a, v, i);

    axiom whither_3:
      \forall value_type *a, v, integer i; a[i] != v ==>
        WhitherRemove(a, v, i+1) == WhitherRemove(a, v, i) + 1;

    axiom whither_lemma:
      \forall value_type *a, v, integer i, j; i < j && a[i] != v ==>
        WhitherRemove(a, v, i) < WhitherRemove(a, v, j);
  }
*/
```

The predicate `WhitherRemove` is defined by a `read` clause. It must be adapted for the current version of the WP plug-in which only implements a limited subset of `read` clauses:

File **remove-copy.axioms [adapted]**

```
/*@
  axiomatic WhitherRemove_Function
  {
    logic integer WhitherRemove{L}(value_type* a, value_type v,
                        integer i) reads a[0], a[i-1];
```

```
    axiom whither_1:
      \forall value_type *a, v; WhitherRemove(a, v, 0) == 0;

    axiom whither_2:
      \forall value_type *a, v, integer i; a[i] == v ==>
        WhitherRemove(a, v, i+1) == WhitherRemove(a, v, i);

    axiom whither_3:
      \forall value_type *a, v, integer i; a[i] != v ==>
        WhitherRemove(a, v, i+1) == WhitherRemove(a, v, i) + 1;

    axiom whither_lemma:
      \forall value_type *a, v, integer i, j; i < j && a[i] != v ==>
        WhitherRemove(a, v, i) < WhitherRemove(a, v, j);
  }
*/
```

5.10.2 Proving the Algorithm

The specification of the `remove-copy` algorithm bases itself on the `RemoveCopy` predicate, itself based on the `WhitherRemove` function:

File **removecopy.h**

```
/*@
  predicate
    RemoveCopy{L}(value_type* a, integer n,
                  value_type* b, integer m, value_type v) =
      m == WhitherRemove(a, v, n) &&
      \forall integer i;
        0 <= i < n && a[i] != v ==> b[WhitherRemove(a, v, i)] == a[i];
*/

/*@
  requires IsValidRange(a, n);
  requires IsValidRange(b, n);
  requires \separated(a+(0..n-1), b+(0..n-1));

  assigns b[0..n-1];

  ensures \forall integer k; \result <= k < n ==> b[k] == \old(b[k]);

  ensures RemoveCopy(a, n, b, \result, val);
*/
size_type remove_copy(const value_type* a, size_type n,
                      value_type* b, value_type val);
```

The implementation of the algorithm is:

File **removecopy.c**

```
size_type remove_copy(const value_type* a, size_type n,
                      value_type* b, value_type v)
{
  size_type j = 0;
  /*@
    loop assigns b[0..j-1], i, j;
    loop invariant 0 <= j <= i <= n;
    loop invariant RemoveCopy(a, i, b, j, v);
    loop variant n-i;
  */
  for (size_type i = 0; i < n; i++) {
    if (a[i] != v)
      b[j++] = a[i];
  }
  return j;
}
```

The implementation can be partially proved correct against its specification by running the WP plug-in:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  removecopy/removecopy.c"
[rte] annotating function remove_copy
[wp] warning: Interpreting reads-definition as expressions rather than tsets
------------------------------------------------------------
Function              #VC    WP  Alt-Ergo    Success
remove_copy            16     -     15      93.8% (1s)
------------------------------------------------------------
```

5.11 The unique-copy Algorithm

We now study the `unique-copy` algorithm from "ACSL By Example". This algorithm copies the contents from a source sequence to a destination sequence whilst removing elements consecutive groups of duplicate elements once the first one is copied. The return value is the length of the range.

5.11.1 Adapting the Axiomatics

The axiomatization provided in "ACSL By Example" is:

File **unique-copy.axioms**

```
/*@
  axiomatic WhitherUnique_Function
  {
    logic integer WhitherUnique{L}(value_type* a, integer i)
          reads a[0..i];

    axiom unique_1:
      \forall value_type *a; WhitherUnique(a, 0) == 0;

    axiom unique_2:
      \forall value_type *a, integer i;
        a[i] == a[i+1] ==>
          WhitherUnique(a, i+1) == WhitherUnique(a, i);

    axiom unique_3:
      \forall value_type *a, v, integer i;
        a[i] != a[i+1] ==>
          WhitherUnique(a, i+1) == WhitherUnique(a, i) + 1;

    axiom unique_lemma_4:
      \forall value_type *a, integer i, j;
        i < j && a[i] != a[i+1] ==>
          WhitherUnique(a, i) < WhitherUnique(a, j);

    axiom unique_lemma_5:
      \forall value_type *a, integer i, j;
        i < j ==> WhitherUnique(a, i) <= WhitherUnique(a, j);
  }
*/
```

The predicate `WhitherUnique` is defined by a `read` clause. It must be adapted for the current version of the WP plug-in which only implements a limited subset of `read` clauses:

File **unique-copy.axioms [adapted]**

```
/*@
  axiomatic WhitherUnique_Function
  {
    logic integer WhitherUnique{L}(value_type* a, integer i)
```

```
        reads a[0], a[i];

    axiom unique_1:
      \forall value_type *a; WhitherUnique(a, 0) == 0;

    axiom unique_2:
      \forall value_type *a, integer i;
        a[i] == a[i+1] ==>
          WhitherUnique(a, i+1) == WhitherUnique(a, i);

    axiom unique_3:
      \forall value_type *a, v, integer i;
        a[i] != a[i+1] ==>
          WhitherUnique(a, i+1) == WhitherUnique(a, i) + 1;

    axiom unique_lemma_4:
      \forall value_type *a, integer i, j;
        i < j && a[i] != a[i+1] ==>
          WhitherUnique(a, i) < WhitherUnique(a, j);

    axiom unique_lemma_5:
      \forall value_type *a, integer i, j;
        i < j ==> WhitherUnique(a, i) <= WhitherUnique(a, j);
  }
*/
```

5.11.2 Proving the Algorithm

The specification of the `unique-copy` algorithm bases itself on the `UniqueCopy` predicate, itself based on the `WhitherUnique` function:

File **uniquecopy.h**

```
/*@
  predicate
    UniqueCopy{L}(value_type* a, integer n,
                  value_type* b, integer m) =
      (n == 0 ==> m == 0) &&
      (n >= 1 ==> m-1 == WhitherUnique(a, n-1)) &&
      \forall integer i;
        0 <= i < n ==> a[i] == b[WhitherUnique(a,i)];
*/

/*@
  requires IsValidRange(a, n);
  requires IsValidRange(b, n);
  requires \separated(a+(0..n-1), b+(0..n-1));

  assigns b[0..n-1];

  ensures \forall integer k; \result <= k < n ==> b[k] == \old(b[k]);

  ensures 0 <= \result <= n;
  ensures UniqueCopy(a, n, b, \result);
*/
size_type unique_copy(const value_type* a,
                      size_type n, value_type* b);
```

The implementation of the algorithm is:

File **uniquecopy.c**

```
size_type unique_copy(const value_type* a, size_type n,
                      value_type* b)
{
  if (n <= 0) return 0;

  size_type j = 0;
  b[j++] = a[0];
  /*@
    loop assigns b[1..j-1], i, j;
    loop invariant 1 <= j <= i <= n;

    // loop invariant \forall integer k; 0 <= k <= i-1
    //                ==> WhitherUnique(a,k) <= WhitherUnique(a,i-1);
    loop invariant \forall integer k; 0 <= k < i
                   ==> a[k] == b[WhitherUnique(a,k)];
    loop invariant UniqueCopy(a, i, b, j);
    loop variant n-i;
  */
  for (size_type i = 1; i < n; i++) {
    if (a[i] != a[i-1])
      b[j++] = a[i];
  }
  return j;
}
```

The implementation can be partially proved correct against its specification by running the WP plug-in:

```
# frama-c -wp -wp-rte [...]
[kernel] preprocessing with "gcc -C -E -I.  uniquecopy/uniquecopy.c"
[rte] annotating function unique_copy
[wp] warning: Interpreting reads-definition as expressions rather than tsets
--------------------------------------------------
Function           #VC    WP  Alt-Ergo    Success
unique_copy         25     1    20       84.0% (3s)
--------------------------------------------------
```

Chapter 6

Rationale

This chapter summurize the results of using WP for proving the algorithms of "ACSL By Example" book.

6.1 General Observations

Missing Assigns. Most of assigns clauses of the original specifications version 5.1.1 were incomplete. This is not a problem when using Jessie because its memory model handles local variables in a different way than WP does. It should be noticed that in pure ACSL, the absence of an assigns clause means *everything* is assigned. Actually, it is always a correct over-approximation, but in general, it is very difficult to prove something after everything has been assigned. WP complains with a warning for missing assigns clauses.

Proving Safety. The WP plug-in does not prove the absence of runtime errors ny its own. In this book, we use RTE plug-in to generate the necessary assertions to guarantee the absence of runtime erros, and finally prove them by WP. This is done very simply by using the `-wp-rte` option of WP.

6.2 Tool Chain

All examples presented in this book are automatically produced by Frama-C. Dedicated *wp-reports* have been used to produce the logs and summary reports of this book.

The reference version and tools are:

Frama-C	Nitrogen-20111001
WP plug-in	0.5
Alt-Ergo	0.93
Processor	2.6 Ghz 4-cores
Memory	4 Go

6.3 Non-Mutating Algorithms

Algorithm	#VC	WP	Alt-Ergo	Success
`equal`	12	2	10	100% (1s)
`mismatch`	17	2	15	100% (1s)
`equal` (mismatch)	5	2	3	100% (1s)
`find`	16	2	14	100% (1s)
`find_first_of`	20	3	17	100% (1s)
`adjacent_find`	19	2	17	100% (1s)
`count`	14	3	11	100% (1s)
`search`	24	3	21	100% (1s)

6.4 Maximum and Minimum Algorithms

Algorithm	#VC	WP	Alt-Ergo	Success
partial orders	6	1	5	100% (1s)
`max_element`	21	3	18	100% (1s)
max with predicates	21	3	18	100% (1s)
`min_element`	21	3	18	100% (1s)
`max_seq`	6	2	4	100% (1s)

6.5 Binary Search Algorithms

Algorithm	#VC	WP	Alt-Ergo	Success
`lower_bound`	17	2	15	100% (5s)
`upper_bound`	17	2	15	100% (3s)
`binary_search`	6	2	4	100% (1s)

6.6 Mutating Algorithms

Algorithm	#VC	WP	Alt-Ergo	Success
`fill`	11	1	10	100% (1s)
`iota`	13	-	13	100% (1s)
`swap`	7	1	6	100% (1s)
`swap_values`	6	1	5	100% (1s)
`swap_ranges`	24	1	23	100% (1s)
`copy`	13	-	13	100% (3s)
`reverse_copy`	15	-	15	100% (1s)
`reverse`	33	2	29	93.9% (1s)
`rotate_copy`	15	-	14	93.3% (3s)
`replace_copy`	18	1	16	94.4% (1s)
`remove_copy`	16	-	15	93.8% (1s)
`unique_copy`	25	1	20	84.0% (3s)