Frama-C Arrays more examples

Dr.S.Padmavathi

CSE, Amrita School of Engineering

Coimbatore

GCD –FRAMA-C

```
#include <stdio.h>
/*@
 axiomatic GCD {
 logic integer Gcd(integer p, integer q);

 axiom Gcd1:
   \forall integer m, n;
   m > n ==> Gcd(m,n) == Gcd(m-n, n);
 axiom Gcd2:
   \forall integer m, n;
   n > m ==> Gcd(m,n) == Gcd(m, n-m);
 axiom Gcd3:
   \forall integer m, n;
   m == n ==> Gcd(m,n) == m;
 }
*/
```

GCD- CONTINUED

```
/*@
  requires  p >= 1;
  requires  p >= 1;
  ensures   \result == Gcd(\old(p), \old(q));
  assigns   \nothing;
*/

int gcd(int p, int q) {

/*@ loop invariant Gcd(p,q) ==
                        Gcd(\at(p,Pre), \at(q,Pre));
   loop assigns p, q;
*/

while (p != q) {
  if (p > q)
              p = p - q;
  if (q > p)
              q = q - p;
}

return p;
}
```

Sum Squares

Frama-C theorem provers for this example are more easily able to work with functional axioms than without them.

```
File  Edit  View  Search  Terminal  Help
#include <stdio.h>

/*@
  requires  n >= 1;
  ensures
  assigns   \nothing ;

*/

int sumsquares(int n) {

  int i = 1;
  int s = 1;

  /*@
    loop invariant s*6 == i*(i+1)*(2*i+1);
    loop invariant 1 <= i <= n;
    loop assigns i, s;
  */

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

WITHOUT AXIOMS

```
/*@
*/
/*@
  requires  n >= 1;
  ensures
  assigns   \nothing ;
*/
int sumsquares2(int n) {
  int i = 1; int s = 1;
  /*@
    loop invariant
    loop invariant 1 <= i <= n;
    loop assigns i, s ;
  */
  while (i < n) {
    i = i + 1;
    s = s + i*i;
  }
  return s;
```

WITH AXIOMS

Specify and prove the function `all_zeros`:

```
// returns a non-zero value iff all elements
// in a given array t of n integers are zeros
int all_zeros(int t[], int n) {
  int k;
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

Identify the error

```
/*@ requires        && \valid
    ensures \result != 0 <==>

*/
int all_zeros(int t[], int n) {
  int k;
  /*@ loop invariant
      loop variant
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

Corrected Specification

```
/*@ requires n>=0 && \valid(t+(0..n-1));

    ensures \result != 0 <==>
      (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
  int k;
  /*@ loop invariant 0 <= k <= n;
      loop invariant
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

\forall and \exists - hints and examples

- Do not confuse && and ==> inside \forall and \exists
- Some common patterns:
  - `\forall integer j; 0 <= j && j < n ==> t[j] == 0;`
  - `\exists integer j; 0 <= j && j < n && t[j] != 0;`
  - Each one here is negation of the other
- A shorter form:
  - `\forall integer j; 0 <= j < n ==> t[j] == 0;`
  - `\exists integer j; 0 <= j < n && t[j] != 0;`
- With several variables:
  - `\forall integer i,j; 0 <= i <= j < length ==> a[i]<=a[j];`
  - `\exists integer i,j; 0 <= i <= j < length && a[i]>a[j]`

Specify and prove the function find_min:

```
// returns the index of the minimal element
// of the given array a of size length
int find_min(int* a, int length) {
  int min, min_idx;
  min_idx = 0;
  min = a[0];
  for (int i = 1; i<length; i++) {
    if (a[i] < min) {
      min_idx = i;
      min = a[i];
    }
  }
  return min_idx;
}
```

Specification Array

```
/*@ requires

    ensures 0<=\result<length &&

int find_min(int* a, int length) {
  int min, min_idx;
  min_idx = 0;
  min = a[0];
  /*@ loop invariant
      loop invariant
      loop invariant
      loop assigns
      loop variant
  for (int i = 1; i<length; i++) {
    if (a[i] < min) {
      min_idx = i;
      min = a[i];
    }
  }
  return min_idx;
}
```

ArraySearch in Frama-C

```
/*@
requires n > 0;
[redacted]
assigns \nothing;
[redacted]
disjoint behaviors;
complete behaviors;
*/
```

```
int arraysearch[redacted] int x, int n) {
[redacted]
for (int p = 0; p < n; p++) {
      if (a[p] == x)
         return 1;
}
return 0;
}
```

In C, the array name is a pointer to the starting location of the array. Hence, Frama-C requires us to state that it is valid to read consecutive n consecutive locations starting from ‘a’.

Immutable array: Array search

```
#include <stddef.h>

/*@
  requires \valid_read(array + (0 .. length-1));

  assigns  \nothing;

  behavior in:
    assumes \exists size_t off ; 0 <= off < length && array[off] == element;
    ensures array <= \result < array+length && *\result == element;

  behavior notin:
    assumes \forall size_t off ; 0 <= off < length ==> array[off] != element;
    ensures \result == NULL;

  disjoint behaviors;
  complete behaviors;
*/
int* search(int* array, size_t length, int element){
  /*@
    loop invariant 0 <= i <= length;
    loop invariant \forall size_t j; 0 <= j < i ==> array[j] != element;
    loop assigns i;
    loop variant length-i;
  */
  for(size_t i = 0; i < length; i++)
    if(array[i] == element) return &array[i];
  return NULL;
}
```

Find element

- /*@ **requires** \valid_read(a + (0..n-1));
- **assigns** \nothing;
- **ensures** 0 <= \result <= n;
- **behavior** some:
  - **assumes** \exists **integer** i; 0 <= i < n && a[i] == v;
  - **assigns** \nothing;
  - **ensures** 0 <= \result < n;
  - **ensures** a[\result] == v;
  - **ensures** \forall **integer** i; 0 <= i < \result ==> a[i] != v;
- **behavior** none:
  - **assumes** \forall **integer** i; 0 <= i < n ==> a[i] != v;
  - **assigns** \nothing;
  - **ensures** \result == n;
- **complete behaviors**;
- **disjoint behaviors**;*/
- **size_type**
- find(**const value_type*** a, **size_type** n, **value_type** v);

Find -continued

```
size_type
find(const value_type* a, size_type n, value_type v)
{
/*@
loop invariant 0 <= i <= n;
loop invariant \forall integer k; 0 <= k < i ==> a[k] != v;
loop assigns i;
loop variant n-i;
*/
for (size_type i = 0u; i < n; i++) {
if (a[i] == v) {
return i;
}
}
return n;
}
```

Specify and prove the function binary_search:

```
/* takes as input a sorted array a, its length,
   and a value key to search,
   returns the index of a cell which contains key,
   returns -1 iff key is not present in the array
*/
int binary_search(int* a, int length, int key) {
  int low = 0, high = length - 1;
  while (low<=high) {
    int mid = (low+high)/2;
    if (a[mid] == key) return mid;
    if (a[mid] < key) { low = mid+1; }
    else { high = mid - 1; }
  }
  return -1;
}
```

- the following, bsearch(t,n, v) searches for element v in array t between indices 0 and n - 1.
- /*@ **requires** n >= 0 ^ **\valid**(t+(0..n-1));
- @ **assigns \nothing**;
- @ **ensures -**1 <=**\result<=** n-1;
- @ **behavior** success:
- @ **ensures \result** >=0 ==> t[**\result**] <==>v;
- @ **behavior** failure:
- @ **assumes** t_is_sorted : forall integer k1, integer k2;
- @ 0 <= k1 <= k2 <= n-1 ==> t[k1] <= t[k2];
- @ **ensures \result** <==> 1 ==>
- @ forall integer k; 0<= k < n ==> t[k] !<==> v;
- @*/
- int bsearch(double t[], int n, double v);

```
/*@ predicate sorted{L}(int* a, int length) =
      \forall integer i,j; 0<=i<=j<length ==> a[i]<=a[j];
*/
/*@ requires \valid(a+(0..length-1));
    requires sorted(a,length);
    requires length >=0;

    assigns \nothing;

    behavior exists:
      assumes \exists integer i; 0<=i<length && a[i] == key;
      ensures 0<=\result<length && a[\result] == key;

    behavior not_exists:
      assumes \forall integer i; 0<=i<length ==> a[i] != key;
      ensures \result == -1;

    complete behaviors;
    disjoint behaviors;
*/
```

Mutable array: array reset

```
1  #include <stddef.h>
2
3  /*@
4    requires \valid(array + (0 .. length-1));
5    assigns  array[0 .. length-1];
6    ensures  \forall size_t i; 0 <= i < length ==> array[i] == 0;
7  */
8  void reset(int* array, size_t length){
9    /*@
10     loop invariant 0 <= i <= length;
11     loop invariant \forall size_t j; 0 <= j < i ==> array[j] == 0;
12     loop assigns i, array[0 .. length-1];
13     loop variant length-i;
14   */
15   for(size_t i = 0; i < length; ++i)
16     array[i] = 0;
```

- The following function sets to 0 each cell of an array.
- /*@ **assigns** t[0..n-1];
- @ **assigns** *(t+(0..n-1));
- @ **assigns** *(t+{ i | integer i ; 0 <= i < n });
- @*/
- void reset_array(int t[],int n) {
- int i;
- **for** (i=0; i < n; i++) t[i] = 0;
- }
- It is annotated with three equivalent assigns clauses, each one specifying that only the set of cells
- {t[0], ... , t[n- 1]} is modified.

Mutable array: replace

```
1  #include <stddef.h>
2
3  /*@
4    requires \valid(array + (0 .. length-1));
5    assigns array[0 .. length-1];
6
7    ensures \forall size_t i; 0 <= i < length && \old(array[i]) == old
8                ==> array[i] == new;
9    ensures \forall size_t i; 0 <= i < length && \old(array[i]) != old
10               ==> array[i] == \old(array[i]);
11 */
12 void search_and_replace(int* array, size_t length, int old, int new){
13   /*@
14     loop invariant 0 <= i <= length;
15     loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) == old
16                      ==> array[j] == new;
17     loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) != old
18                      ==> array[j] == \at(array[j], Pre);
19     loop assigns i, array[0 .. length-1];
20     loop variant length-i;
21   */
22   for(size_t i = 0; i < length; ++i){
23     if(array[i] == old) array[i] = new;
24   }
25 }
```

Correct specification

```
1  for(size_t i = 0; i < length; ++i){
2      //@assert array[i] == \at(array[i], Pre); // proof failure
3      if(array[i] == old) array[i] = new;
4  }
```

We can add this information as an invariant:

```
13   /*@
14     loop invariant 0 <= i <= length;
15     loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) == old
16                       ==> array[j] == new;
17     loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) != old
18                       ==> array[j] == \at(array[j], Pre);
19     loop invariant \forall size_t j; i <= j < length
20                       ==> array[j] == \at(array[j], Pre);
21     loop assigns i, array[0 .. length-1];
22     loop variant length-i;
23   */
24   for(size_t i = 0; i < length; ++i){
25     if(array[i] == old) array[i] = new;
26   }
```

Specify and prove the function **sort**:

```
// sorts given array a of size length > 0
void sort (int* a, int length) {
  int current;
  for (current = 0; current < length - 1; current++) {
    int min_idx = current;
    int min = a[current];
    for (int i = current + 1; i < length; i++) {
      if (a[i] < min) {
        min = a[i];
        min_idx = i;
      }
    }
    if (min_idx != current){
      L: a[min_idx]=a[current];
      a[current]=min;
    }
  }
}
```

Referring to another state

- Specification may require values at differents program points
- Use `\at(e,L)` to refer to the value of expression `e` at label `L`
- Some predefined labels:
  - `\at(e,Here)` refers to the current state
  - `\at(e,Old)` refers to the pre-state
  - `\at(e,Post)` refers to the post-state
- `\old(e)` is equivalent to `\at(e,Old)`

```
/*@ predicate sorted{L}(int* a, integer length) =
      \forall integer i,j; 0<=i<=j<length ==> a[i]<=a[j];
*/
/*@ predicate swap{L1,L2}(int* a,integer i,integer j,integer length)=
      0<=i<j<length
      && \at(a[i],L1) == \at(a[j],L2)
      && \at(a[i],L2) == \at(a[j],L1)
      && \forall integer k; 0<=k<length && k!=i && k!=j ==>
           \at(a[k],L1) == \at(a[k],L2);
*/
/*@ inductive same_elements{L1,L2}(int*a , integer length) {
      case refl{L}:
        \forall int*a, integer length; same_elements{L,L}(a,length);
      case swap{L1,L2}: \forall int*a, integer i,j,length;
        swap{L1,L2}(a,i,j,length) ==> same_elements{L1,L2}(a,length);
      case trans{L1,L2,L3}: \forall int*a, integer length;
        same_elements{L1,L2}(a,length)
        ==> same_elements{L2,L3}(a,length)
        ==> same_elements{L1,L3}(a,length);
}
*/
```

```
/*@ requires \valid(a+(0..length-1));
    requires length > 0;
    assigns a[0..length-1];
    behavior sorted:
      ensures sorted(a,length);
    behavior same_elements:
      ensures same_elements{Pre,Here}(a,length);
*/
void sort (int* a, int length) {
  int current;
  /*@ loop invariant 0<=current<length;
      loop assigns a[0..length-1],current;
      for sorted: loop invariant sorted(a,current);
      for sorted: loop invariant
        \forall integer i,j; 0<=i<current<=j<length ==> a[i] <= a[j];
      for same_elements: loop invariant
        same_elements{Pre,Here}(a,length);
      loop variant length-current;
   */
```

```
  for (current = 0; current < length - 1; current++) {
    int min_idx = current;
    int min = a[current];
    /*@ loop invariant current+1<=i<=length;
        loop assigns i,min,min_idx;
        loop invariant current<=min_idx<i;
        loop invariant a[min_idx] == min;
        for sorted: loop invariant
          \forall integer j; current<=j<i ==> min <= a[j];
        loop variant length -i;
    */
    for (int i = current + 1; i < length; i++) {
      if (a[i] < min) {
        min = a[i];
        min_idx = i;
      }
    }
    if(min_idx != current) {
       L: a[min_idx]=a[current];
       a[current]=min;
/*@for same_elements:assert swap{L,Here}(a,current,min_idx,length);*/
    }
  }
}
```

Proof failures

- A proof of a VC for some annotation can fail for various reasons:
  - incorrect implementation (! check your code)
  - incorrect annotation (! check your spec)
  - missing or erroneous (previous) annotation (! check your spec)
  - insuficient timeout (! try longer timeout)
  - complex property that automatic provers cannot handle.

Analysis of proof failures

- When a proof failure is due to the specification, the erroneous annotation may be not obvious to find. For example:
  - proof of a \loop invariant preserved" may fail in case of
    - incorrect loop invariant
    - incorrect loop invariant in a previous, or inner, or outer loop
    - missing assumes or loop assumes clause
    - too weak precondition
  - proof of a post condition may fail in case of
    - incorrect loop invariant (too weak, too strong, or inappropriate)
    - missing assumes or loop assumes clause
    - inappropriate post condition in a called function
    - too weak precondition

FRAMA-C

As a result, a consolidated property status can either be a *simple* status:

- `never_tried`: when no status is available for the property.
- `unknown`: whenever the status is `Maybe`.
- `surely_valid`: when the status is `True`, and dependencies have the consolidated status `surely_valid` or `considered_valid`.
- `surely_invalid`: when the status is `False`, and all dependencies have the consolidated status `surely_valid`.
- `inconsistent`: when there exist two conflicting consolidated statuses for the same property, for instance with values `surely_valid` and `surely_invalid`. This case may also arise when an invalid cyclic proof is detected. This is symptomatic of an incoherent axiomatization.

or an *incomplete* status:

- `considered_valid`: when there is no possible way to prove the property (e.g., the post-condition of an external function). We assume this property will be validated by external means.
- `valid_under_hyp`: when the local status is `True` but at least one of the dependencies has consolidated status `unknown`. This is typical of proofs in progress.
- `invalid_under_hyp`: when the local status is `False`, but at least one of the dependencies has status `unknown`. This is a telltale sign of a dead code property, or of an erroneous annotation.

and finally:

- `unknown_but_dead`: when the status is locally `Maybe`, but in a dead or incoherent branch.
- `valid_but_dead`: when the status is locally `True`, but in a dead or incoherent branch.
- `invalid_but_dead`: when the status is locally `False`, but in a dead or incoherent branch.

The fact that two arrays `a[0]..a[n-1]` and `b[0]..b[n-1]` are equal when compared element by element, is a property we might need again in other specifications, as it describes a very basic property.

We start with introducing several *overloaded* versions of the predicate `EqualRanges` [4.28].

```
/*@
  axiomatic EqualRanges
  {
    predicate
    EqualRanges{K,L}(value_type* a, integer n, value_type* b) =
      \forall integer i; 0 <= i < n  ==>  \at(a[i],K) == \at(b[i],L);

    predicate
    EqualRanges{K,L}(value_type* a, integer m, integer n, value_type* b) =
      \forall integer i; m <= i < n  ==>  \at(a[i],K) == \at(b[i],L);

    predicate
    EqualRanges{K,L}(value_type* a, integer m, integer n,
                     value_type* b, integer p) = EqualRanges{K,L}(a+m, n-m, b+p);

    predicate
    EqualRanges{K,L}(value_type* a, integer m, integer n, integer p) =
      EqualRanges{K,L}(a, m, n, a, p);
  }
*/
```

defined label `Here`. When used in an `ensures` clause, the label `Here` refers to the post-state of a function. Note that the equivalence is needed in the ensures clause. Putting an equality instead is not legal in ACSL, because `EqualRanges` is a predicate, not a function.

```
/*@
  requires valid:  \valid_read(a + (0..n-1));
  requires valid:  \valid_read(b + (0..n-1));
  assigns          \nothing;
  ensures result:  \result <==> EqualRanges{Here,Here}(a, n, b);
*/
bool
equal(const value_type* a, size_type n, const value_type* b);
```

return value of mismatch provides more information than just reporting whether the two arrays are equal.

```
/*@
  requires valid:   \valid_read(a + (0..n-1));
  requires valid:   \valid_read(b + (0..n-1));
  assigns           \nothing;
  ensures  result:  0 <= \result <= n;

  behavior all_equal:
    assumes         EqualRanges{Here,Here}(a, n, b);
    assigns         \nothing;
    ensures result: \result == n;

  behavior some_not_equal:
    assumes         !EqualRanges{Here,Here}(a, n, b);
    assigns         \nothing;
    ensures bound:  0 <= \result < n;
    ensures result: a[\result] != b[\result];
    ensures first:  EqualRanges{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 of `equal` [4.31] consists of a simple call of `mismatch`.

```
bool
equal(const value_type* a, size_type n, const value_type* b)
{
  return mismatch(a, n, b) == n;
}
```

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

  return n;
}
```

```
/*@
  axiomatic Reverse
  {
    predicate
    Reverse{K,L}(value_type* a, integer n, value_type* b) =
      \forall integer i; 0 <= i < n  ==>  \at(a[i],K) == \at(b[n-1-i], L);

    predicate
    Reverse{K,L}(value_type* a, integer m, integer n,
                 value_type* b, integer p) = Reverse{K,L}(a+m, n-m, b+p);

    predicate
    Reverse{K,L}(value_type* a, integer m, integer n, value_type* b) =
      Reverse{K,L}(a, m, n, b, m);

    predicate
    Reverse{K,L}(value_type* a, integer m, integer n, integer p) =
      Reverse{K,L}(a, m, n, a, p);

    predicate
    Reverse{K,L}(value_type* a, integer m, integer n) =
      Reverse{K,L}(a, m, n, m);

    predicate
    Reverse{K,L}(value_type* a, integer n) = Reverse{K,L}(a, 0, n);
  }
*/
```

```
/*@
  requires valid:       \valid_read(a + (0..n-1));
  requires valid:       \valid(b + (0..n-1));
  requires sep:         \separated(a + (0..n-1), b + (0..n-1));
  assigns               b[0..(n-1)];
  ensures reverse:      Reverse{Old,Here}(a, n, b);
  ensures unchanged:    Unchanged{Old,Here}(a, n);
*/
void
reverse_copy(const value_type* a, size_type n, value_type* b);
```

```
void
reverse_copy(const value_type* a, size_type n, value_type* b)
{
  /*@
    loop invariant bound:    0 <= i <= n;
    loop invariant reverse: Reverse{Here,Pre}(b, 0, i, a, n-i);
    loop assigns i, b[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    b[i] = a[n - 1u - i];
  }
}
```

following listing the overloaded predicate `Unchanged` [7.1]. The expression `Unchanged{K,L}(a,f,l)` is true if the range `a[f..l-1]` in state `K` is element-wise equal to that range in state `L`.

```
/*@
  axiomatic Unchanged
  {
    predicate
    Unchanged{K,L}(value_type* a, integer m, integer n) =
      \forall integer i; m <= i < n ==>  \at(a[i],K) == \at(a[i],L);

    predicate
    Unchanged{K,L}(value_type* a, integer n) = Unchanged{K,L}(a, 0, n);
  }
```

```
/*@
  requires valid:  \valid(a + (0..n-1));
  assigns          a[0..n-1];
  ensures reverse: Reverse{Old,Here}(a, n);
*/
void
reverse(value_type* a, size_type n);
```

```
void
reverse(value_type* a, size_type n)
{
  const size_type half = n / 2u;

  //@ assert half: half <= n - half;
  //@ assert half: 2*half <= n <= 2*half + 1;
  /*@
    loop invariant bound:    0 <= i <= half <= n-i;
    loop invariant left:     Reverse{Pre,Here}(a, 0, i, n-i);
    loop invariant middle: Unchanged{Pre,Here}(a, i, n-i);
    loop invariant right:    Reverse{Pre,Here}(a, n-i, n, 0);
    loop assigns i, a[0..n-1];
    loop variant half - i;
  */
  for (size_type i = 0u; i < half; ++i) {
    swap(&a[i], &a[n - 1u - i]);
  }
}
```