Testing and Formal Verification of the Algorithm `unique_copy` from the C++ Standard Library

Jens Gerlach   Tim Sikatzki

November 2018

Fraunhofer
FOKUS

This report was funded by the VESSEDIA project.

The project VESSEDIA has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453. Project duration: 2017–2019, see https://vessedia.eu

In this document we take a closer look on testing and formal verification of the algorithm unique_copy from the C++ standard library. We start in Chapter 1 with a careful analysis of unique_copy's informal requirements. We use the identified requirements to guide our testing and verification artefacts in the subsequent chapters.

Although testing is rightly considered easier than formal verification, *designing* both good test code and convincing test data is far from trivial. This is particularly true when one has to deal with testing properties that rely on *implicit* relationships between the input and output data of an algorithm. In the case of unique_copy, the main problem is to show that the first (and only the first) element of each consecutive range of equal elements is copied. We tackle this problem in Chapter 2 by exploiting the fact that we deal with a *generic* algorithm. This allows us to transport additional data through the algorithm under test and subsequently use this data to establish that the original data are processed according to the requirements.

In Chapter 3 we then proceed to formally verify a non-generic C version of unique_copy with the Frama-C [1] verification platform. In particular, we are writing formal function contract in Frama-C's specification language ACSL [2]. Here, we emphasize that there are different levels of how formal one wants to be. Thus, we consider both the formal verification of the *absence of undefined behavior* and the verification of the *functional correctness* of unique_copy.

We, unsurprisingly, conclude in Chapter 4 that testing and formal verification are complementary and not conflicting activities and also point out the existing support in Frama-C to approach both techniques in a coordinated way.

Contents

Code listings

List of logic definitions

List of Figures

List of Tables

1. Informal specification

This chapter deals with the informal specification of `unique_copy` and its behavior. In Section 1.1 we present the requirements of `unique_copy` as they are derived from the C++ standard library. In Section 1.2 we look at specific examples to provide a better understanding of `unique_copy`. To take it one step further we finally present in Section 1.3 a more formal analysis of `unique_copy`'s behavior.

1.1. Informal specification of `unique_copy`

The `unique_copy` algorithms of the C++ standard library[3, §25.3.9], whose signature is shown in Listing 1.1, is a template function which copies certain values from a sequence given by the right-open interval of iterators `[first, last)` into a sequence that starts at the iterator `result`.

```
template<class InputIterator, class OutputIterator>
OutputIterator
unique_copy(InputIterator first, InputIterator last,
          OutputIterator result);
```

Listing 1.1: Signature of `unique_copy` from the C++ standard library

For the purposes of this report we do not consider the wide possibilities of ranges covered by this signature, rather we assume the two ranges to be arrays[1] `a[0..n-1]` and `b[0..n-1]` of length `n`. Listing 1.2 shows thus the signature and implementation of a simplified yet still generic function `unique_copy`. Later in this document we will consider an even more specific version of `unique_copy` that is implemented in C.

```
template<typename T>
size_type unique_copy(const T* a, size_type n, T* b)
{
  auto result = std::unique_copy(a, a + n, b);
  return result - b;
}
```

Listing 1.2: A simplified version of `unique_copy`

Besides assuming the input and output ranges to be (generic) arrays, this version of `unique_copy` returns the number of copied elements instead of an iterator that indicates the last copied element in the output range.

[1] We employ here and in the following the ACSL notation `a[0..n-1]` to denote an array of `n` elements.

Table 1.3 shows our interpretation of the requirements for `unique_copy` from the C++ standard[25.3.9][3] for the signature of Listing 1.2.

Requirement	Description
Unique Copy Size	The output range must be able to store the same number of elements as the input range.
Unique Copy Separation	The input range and the output range do not overlap.
Unique Copy Consecutive	Only the first element from every consecutive group of equal elements of the input range is copied into the output range.
Unique Copy Return	The algorithm returns the number of copied elements.
Unique Copy Complexity	At most $n-1$ comparisons of adjacent elements are performed.

Table 1.3.: Requirements of `unique_copy`

Requirement **Unique Copy Consecutive** captures the core functionality of the algorithms. The intention of this requirement, which is not explicitly mentioned in **Unique Copy Consecutive**, is that the copied elements do *not* contain adjacent equal elements. One goal of this report is to show how this requirement can be expressed in Frama-C's specification language ACSL.

Complexity requirements, as formulated in terms of the number of comparison operations in requirement **Unique Copy Complexity**, are essential for specifying efficient algorithms. However, since Frama-C does currently not provide sufficient support for specifying this kind of requirements, we will not consider them in the rest of this document.

1.2. Some examples for `unique_copy`

In this section we analyze the requirement **Unique Copy Consecutive** by looking at how `unique_copy` behaves on different inputs.

Figure 1.4 shows the result of `unique_copy` when applied to a short array of integers. The arrows indicate from which index in the array `a` the respective value in the array `b` originates. The gray portion in the target array indicates that in our example not all elements from `a` have been copied.

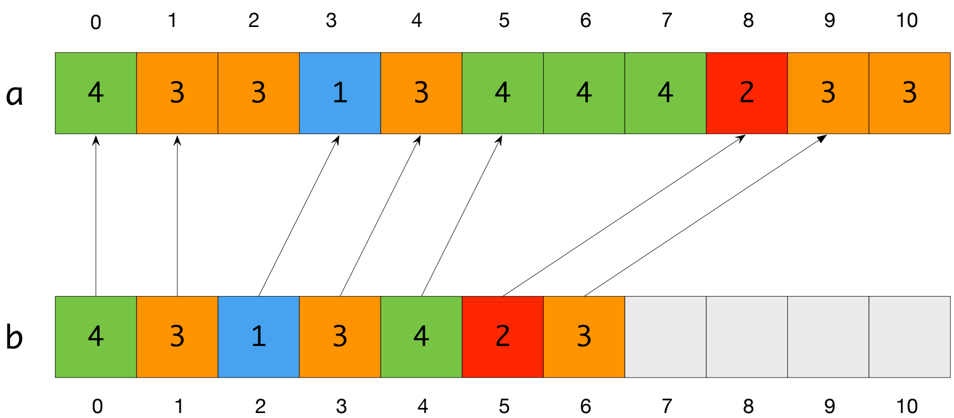

Figure 1.4.: Example of applying `unique_copy`

Requirement **Unique Copy Consecutive** also implies that if `unique_copy` is applied to sequence that contains no adjacent equal elements in the first place, then it behaves like an ordinary copy algorithms (Figure 1.5).

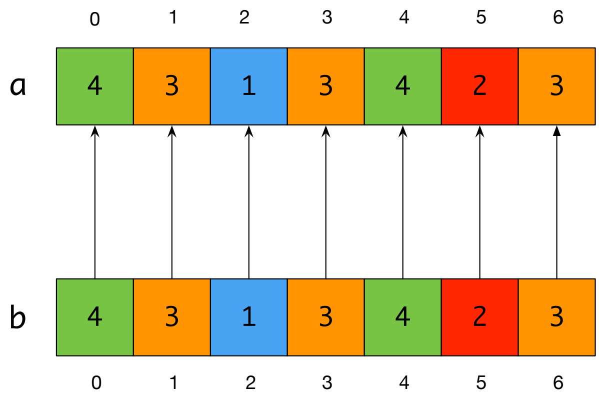

Figure 1.5.: Applying `unique_copy` to a sequence with no adjacent equal elements

Another, somewhat extreme, example is applying `unique_copy` to a sequence where all elements are equal to each other. In this case, the result of `unique_copy` will consists of a single value (Figure 1.6).

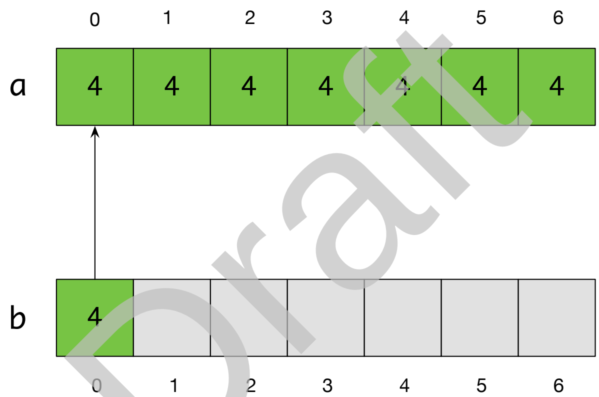

Figure 1.6.: Applying `unique_copy` to a sequence where all elements are equal

A typical use case of `unique_copy` is to apply it to a *sorted* sequence. In this case calling `unique_copy` ensures that each value of the input range occurs exactly once in the output range (Figure 1.7). This is of course known to `Unix` programmers who can use `sort FILE | uniq` to remove all duplicate lines from `FILE`.

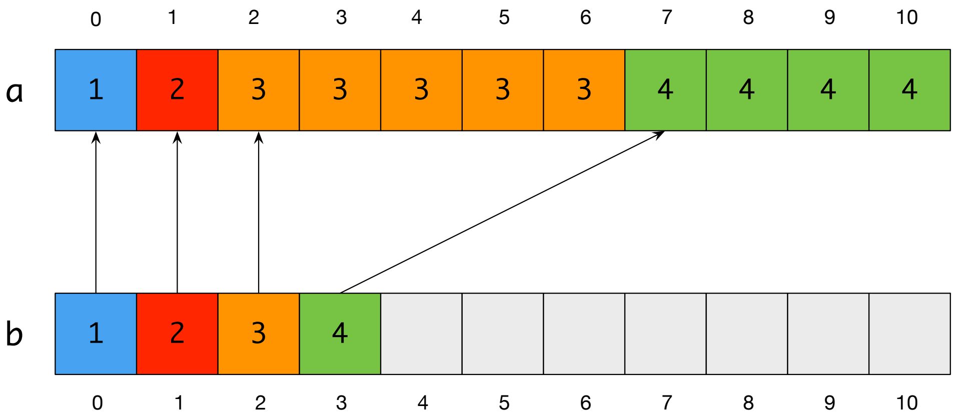

Figure 1.7.: Applying `unique_copy` to remove all duplicate elements from a sorted sequence

1.3. A first analysis of `unique_copy`

Figure 1.8 is a slight modification of Figure 1.4. We show here only the indices of the source array whose values are copied into the target array. In addition, we have added another (dashed) arrow to link the indices that correspond to the *one past the end* locations of the input and output ranges, respectively. We use this additional arrow in order to be able to describe all sub sequences of consecutive equal elements in the source array.

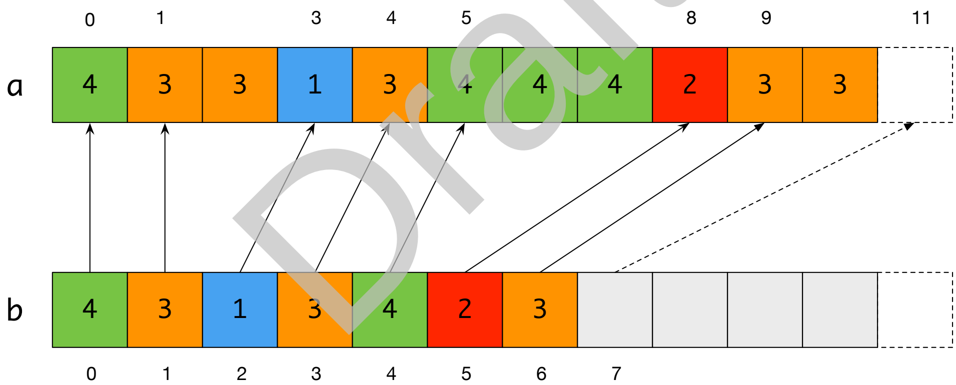

Figure 1.8.: Partitioning the input of `unique_copy`

These arrows between the indices of the array `b` and the array `a` define the following sequence p of eight indices where the index that points one past the end is underlined.

$p = (0, 1, 3, 4, 5, 8, 9, \underline{11})$ for Figure 1.8

For the other examples the corresponding index sequences are

$p = (0, 1, 2, 3, 4, 5, 6, \underline{7})$ for Figure 1.5
$p = (0, \underline{7})$ for Figure 1.6
$p = (0, 1, 2, 7, \underline{11})$ for Figure 1.7

Don't forget that the last three figures *do not show* the respective *one past the end* arrows.

More generally, we refer to the sequence p as *partitioning sequence* of `unique_copy` for the array `a[0..n-1]`. This sequence is characterized by the following properties: If $m + 1$ is the **length of a partitioning sequence**, then we observe that they are **strictly monotone increasing**

$$0 = p_0 < \ldots < p_{m+1} = n \tag{1.1}$$

and that the right-open index intervals

$$[p_i, p_{i+1}) \qquad \forall i : 0 \le i < m$$

mark **consecutive ranges** of equal elements in the source array, that is,

$$\mathsf{a}[p_i] = \mathsf{a}[k] \qquad \forall k : p_i \le k < p_{i+1} \tag{1.2}$$

We also have that the consecutive ranges are **maximal** in the following sense

$$\mathsf{a}[p_i] \neq \mathsf{a}[p_{i+1}] \qquad \forall i : 0 \le i < m - 1 \tag{1.3}$$

and last but not least for the **result of `unique_copy`** it must hold

$$\mathsf{b}[i] = \mathsf{a}[p_i] \qquad \forall i : 0 \le i < m \tag{1.4}$$

We have added an appendix A to this document where we have a more mathematical look on the existence and the properties of the partitoning sequence.

2. Unit tests

In this chapter we derive both *test data* and *test code* that shall establish that the implementation of `unique_copy` from Listing 1.2 satisfies the requirements listed in Table 1.3. We are mainly concerned with *unit tests* that capture the functionality of `unique_copy`. This is also the reason why we are not discussing the also important issue of *code coverage*.

We are dealing in this chapter with tests of a *generic* implementation, that is, with an implementation that is parameterized over the type of the elements stored in arrays. Our test code is therefore also as generic as suitable, see Section 2.1 for example. Of course, the actual test execution appears with both specific type parameters and concrete test data.

We start our presentation in Section 2.2 with a discussion of suitable test data for `unique_copy`. Thereby we also have a look at the test data for `unique_copy` in *libcxx* [4], an open source implementation of the C++ standard library.

Requirement **Unique Copy Consecutive** captures the core of `unique_copy`, and the design of our tests aims particularly at checking this requirement. This is, however, not an easy undertaking because it is not clear in the beginning how to convincingly demonstrate that the *first* (and only the first) element of a consecutive group of equal elements is copied. Our first tests `unique_copy` in Sections 2.3 and 2.4 (and the corresponding tests in *libcxx*) therefore only show that there are no adjacent equal elements in the output array.

There is, however, a not too complicated method to show the copying of the first element of the consecutive ranges. As we will show in Section 2.5.3 this method relies both on

1. our semi-formal analysis of the `unique_copy` in Section 1.3 and
2. the *generic* nature of the implementation of `unique_copy`.

The latter allows us to replace values of type `T` essentially by `std::pair<T,size_t>` where the `second` field of `std::pair` will hold the index of the copied element in the input sequence of `unique_copy`.

2.1. Preparation of test code

In order to facilitate the testing of `unique_copy` from Listing 1.2, we provide a wrapper implementation that uses the container `vector` from the C++ standard library.

```
template<typename T>
std::vector<T>
unique_copy(const std::vector<T>& a)
{
  std::vector<T> b(a.size());

  auto size = unique_copy(a.data(), a.size(), b.data());
  b.resize(size);

  return b;
}
```

Listing 2.1: `unique_copy` for `std::vector`

Using this generic auxiliary function from Listing 2.1 has several advantages for our tests.

- The `vector` container conveniently encapsulates both the memory and the number of elements of a C array.
- A `vector` hides many details of dynamic memory allocation from the user. Listing 1.2 shows that it is also easy to *resize* a `vector` object after it was created.
- The C++ standard ensures that different `vector` objects manage their own memory. Thus, using `vector`, it is easy to satisfy **Unique Copy Separation** which states that its two array arguments do not overlap.
- Also note that we initially declare the output `vector` to have the same size as the input `vector`. We are thus making sure that the requirement **Unique Copy Size** is satisfied.

2.2. Test data

Table 2.2 shows our initial test data for `unique_copy`. These are exactly the examples from Section 1.2 that have been used there to describe the behavior of `unique_copy`.

Input	Output	Reference
(4, 3, 3, 1, 3, 4, 4, 4, 2, 3, 3)	(4, 3, 1, 3, 4, 2, 3)	Figure 1.4
(4, 3, 1, 3, 4, 2, 3)	(4, 3, 1, 3, 4, 2, 3)	Figure 1.5
(4, 4, 4, 4, 4, 4, 4)	(4)	Figure 1.6
(1, 2, 3, 3, 3, 3, 3, 4, 4, 4)	(1, 2, 3, 4)	Figure 1.7

Table 2.2.: Initial test data

Boundary test data are data for extreme inputs of the specific algorithm. It can be argued that the example from Figure 1.6 where the elements of the input array equal *one* value represents boundary test data. Table 2.3 shows the expected behavior of `unique_copy` for input ranges of size 0 and size 1.

Input	Output	Reference
( )	( )	empty input range
(3)	(3)	one-element input range

Table 2.3.: Boundary test data

It is interesting to compare our test data with those from the functional tests for `unique_copy` in an open source implementation of the C++ standard library [4, `unique_copy.pass.cpp`]. We have listed these test data in Table 2.4.

Input	Output
(0, 1, 2, 2, 4)	(0, 1, 2, 4)
(0)	(0)
(0, 1)	(0, 1)
(0, 0)	(0)
(0, 0, 1)	(0, 1)
(0, 0, 1, 0)	(0, 1, 0)
(0, 0, 1, 1)	(0, 1)
(0, 0, 1)	(0, 1)
(0, 1, 1, 1, 2, 2, 2)	(0, 1, 2)

Table 2.4.: Test data for `unique_copy` from an open source test suite

Here the emphasis is on an arguably more systematic presentation of small input ranges of sizes. Interestingly, however, there is no test case for the empty range.

2.3. A basic test

When we present *test code* then we present code that shows that the function under test satisfies certain *properties* which in turn are justified by the requirements.

Listing 2.5, for example, contains code that tests that the elements copied by `unique_copy` contain no adjacent equal elements. This is, as we have explained in Section 1.2, a simple consequence of **Unique Copy Consecutive** from Table 1.3.

```
template<typename T>
std::vector<T>
unique_copy_basic_test(const std::vector<T>& input)
{
  auto result = unique_copy(input);
  assert(std::adjacent_find(result.begin(), result.end()) == result.end());

  //std::cout << "test " << __func__ << " succeeded " << std::endl;

  return result;
}
```

Listing 2.5: Testing the absence of adjacent equal elements

The key ingredient of the test in Listing 2.5 consists in calling the C++ standard library function `adjacent_find` which searches its input range for (the first) occurrence of two consecutive equal elements. If there is no such occurrence, `adjacent_find` returns the iterator that indicates the end of the range.

Listing 2.6 shows how the test from Listing 2.5 is executed.

```
int main(int argc, char** argv)
{
  assert(argc == 2);
  std::fstream file(argv[1]);
  std::vector<int> v;

  while (true) {
    file >> v;
    if (file) {
      // std::cout << v << std::endl;
      unique_copy_basic_test(v);
      v.clear();
    }
    else {
      break;
    }
  }

  std::cout << "\tsuccessful execution of " << argv[0] << "\n";

  return EXIT_SUCCESS;
}
```

Listing 2.6: Test execution code for the absence of adjacent equal elements

In our setting the test data step from the file in Listing 2.7 which contains the input data from Tables 2.2, 2.3, and 2.4.

```
(4, 3, 3, 1, 3, 4, 4, 4, 2, 3, 3)
(4, 3, 1, 3, 4, 2, 3)
(4, 4, 4, 4, 4, 4, 4)
(1, 2, 3, 3, 3, 3, 3, 4, 4, 4)

(3)
()

(0, 1, 2, 2, 4)
(0)
(0, 1)
(0, 0)
(0, 0, 1)
(0, 0, 1, 0)
(0, 0, 1, 1)
(0, 0, 1)
(0, 1, 1, 1, 2, 2, 2)
```

Listing 2.7: Test input data for `unique_copy`

2.4. Extending the basic test

This basic test can be extended to the slightly more elaborate test in Listing 2.8 that in addition to Listing 2.6 compares whether

1. the number of copied elements equals the size of the expected output range and
2. the copied elements actually equal the expected output range.

```
template<typename T>
void
unique_copy_compare_test(const std::vector<T>& input,
                         const std::vector<T>& expected)
{
  auto result = unique_copy_basic_test<T>(input);
  assert(result == expected);

  //std::cout << "test " << __func__ << " succeeded " << std::endl;
}
```

Listing 2.8: Comparing with expected result

Listing 2.9 shows how the test from Listing 2.8 is executed with the test data read from a file.

```
int main(int argc, char** argv)
{
  assert(argc == 2);
  std::fstream file(argv[1]);

  while (true) {
    std::vector<int> input, expected;
    file >> input;
    file >> expected;
    if (file) {
      // std::cout << input << "\t" << expected << std::endl;
      unique_copy_compare_test(input, expected);
    }
    else {
      break;
    }
  }

  std::cout << "\tsuccessful execution of " << argv[0] << "\n";

  return EXIT_SUCCESS;
}
```

Listing 2.9: Test execution code for comparing with expected result

In this setting, the test data step from the file in Listing 2.10 which contains the input data and the expected output data from Tables 2.2, 2.3, and 2.4.

```
(4, 3, 3, 1, 3, 4, 4, 4, 2, 3, 3)  (4, 3, 1, 3, 4, 2, 3)
(4, 3, 1, 3, 4, 2, 3)              (4, 3, 1, 3, 4, 2, 3)
(4, 4, 4, 4, 4, 4, 4)              (4)
(1, 2, 3, 3, 3, 3, 3, 4, 4, 4      (1, 2, 3, 4)

()                                 ()
(3)                                (3)

(0, 1, 2, 2, 4)                    (0, 1, 2, 4)
(0)                                (0)
(0, 1)                             (0, 1)
(0, 0)                             (0)
(0, 0, 1)                          (0, 1)
(0, 0, 1, 0)                       (0, 1, 0)
(0, 0, 1, 1)                       (0, 1)
(0, 0, 1)                          (0, 1)
(0, 1, 1, 1, 2, 2, 2)              (0, 1, 2)
```

Listing 2.10: Test input and expected output for `unique_copy`

Executing the test from Listing 2.8 with the test data from Section 2.2 allows us to establish whether `unique_copy` satisfies the Requirement **Unique Copy Size** and the above mentioned consequence of Requirement **Unique Copy Consecutive**. However, these tests cannot establish that the *first* (and only the first) element of each consecutive group of equal elements is copied. In this sense, our test is as expressive as the tests for `unique_copy` from [4, `unique_copy.pass.cpp`]

2.5. Partition testing

In Section 1.3 we had, informally, argued that for each sequence $a = (a_0, \ldots, a_{n-1})$ there is a *partitioning sequence* $p = (p_0, \ldots, p_m)$ that satisfies the conditions (1.1), (1.2), and (1.3) and which, moreover, relates the input and output of `unique_copy` according to Equation (1.4). The term *partition testing* refers here to using these properties in the design of our tests.

The problem is that the partitioning sequence does not *explicitly* occur in `unique_copy`. There is, however, a relatively simple and natural way to make the partitioning sequence explicitly.

We explain the basic idea at hand of the input sequence

$$a = (4, 3, 3, 1, 3, 4, 4, 4, 2, 3, 3)$$

from Figure 1.8. We begin with creating a sequence of pairs (a_i, i) consisting of the original value a_i and its index i, in other words, we *zip* the sequence a with the sequence of its indices. In order to facilitate the readability of lists of pairs we also write $\binom{a_i}{i}$ instead of (a_i, i).

For our example, the augmented sequence of pairs a' reads

$$a' = \left(\binom{4}{0}, \binom{3}{1}, \binom{3}{2}, \binom{1}{3}, \binom{3}{4}, \binom{4}{5}, \binom{4}{6}, \binom{4}{7}, \binom{2}{8}, \binom{3}{9}, \binom{3}{10}\right)$$

If we define the equality of two pairs (a_i, i) and (a_j, j) by the equality of its first components, then `unique_copy` *piggybacks* the original index of an element into the result. In other words, `unique_copy` applied to the sequence a' produces the following sequence

$$b' = \left(\binom{4}{0}, \binom{3}{1}, \binom{1}{3}, \binom{3}{4}, \binom{4}{5}, \binom{2}{8}, \binom{3}{9}\right)$$

where the second components indicate the index in the input sequence.

Finally, we extract the two lists of its first and second components from b' (that is we *unzip* b') and add a final element 11 (the number of elements of a) to the sequence of indices. We thus obtain the sequence

$$b = (4, 3, 1, 3, 4, 2, 3)$$

which is just the result of applying `unique_copy` to a, and the partitioning sequence

$$p = (0, 1, 3, 4, 5, 8, 9, 11)$$

Both sequences can, of course be found in Figure 1.8.

In the following subsections, we discuss the details of the implementation of our partitioning test.

2.5.1. Pairs of values and indices

Listing 2.11 shows our definition of a type for *indexed values* which consists of a pair of generic type `T` and `size_t` as index type. We implement this type with the generic class `std::pair` that has two public fields `first` and `second`, respectively.

```
template<typename T>
struct Indexed : public std::pair<T, size_t> {

  // inherit constructors of base class
  using std::pair<T, size_t>::pair;

};

template<typename T>
bool operator==(const Indexed<T>& a, const Indexed<T>& b)
{
  return a.first == b.first;
}
```

Listing 2.11: A type for indexed values

The equality operation (`operator==`) of this new type has been defined in such a way that only the first component of the underlying pair type is evaluated. Note that we only provide a special implementation for the equality operation but not for copying and assigning a value `p` of type `Indexed<T>`. This is essential for keeping the values `p.first` and `p.second` unchanged while they are processed by our generic `unique_copy`.

2.5.2. Creating the partition sequence

Listing 2.12 shows our implementation of computing the partitioning sequence of a given input sequence.

```
template<typename T>
std::pair<std::vector<T>, std::vector<size_t>>
    unique_copy_create_partition(const std::vector<T>& input)
{
  std::vector<Indexed<T>> zipped(input.size());
  for (size_t i = 0; i < input.size(); i++) {
    zipped[i] = Indexed<T>(input[i], i);
  }

  auto output = unique_copy(zipped);

  std::pair<std::vector<T>, std::vector<size_t>> unzipped;
  unzipped.first.resize(output.size());
  unzipped.second.resize(output.size() + 1);

  for (size_t i = 0; i < output.size(); ++i) {
    unzipped.first[i]  = output[i].first;
    unzipped.second[i] = output[i].second;
  }
  unzipped.second.back() = input.size();

  return unzipped;
}
```

Listing 2.12: Creating the partition underlying `unique_copy`

- As explained at the beginning of this section we start with creating the sequence of pairs of values and their respective indices.
- We then pass this sequence of pairs to our generic version of `unique_copy` for `vector` from Listing 2.1. The resulting sequence contains, thanks to our definition of equality of our pair type, the values with their indices in the original sequence.
- We then *unzip* the vector of pairs into a pair of vectors and add to the index vector the size of the input sequence as final element.

2.5.3. Testing the partition properties

Listing 2.13 shows our generic partition test. We have added comments to facilitate the tracing of our test code to the properties (1.1)–(1.4) from Section 1.3.

First unique_copy is called and the corresponding partition sequence is computed. Note that computing of this partition sequence naturally also leads to a second computation of the result of unique_copy. Thus, initially we check (for sanity) that both computations have the same result.

```
template<typename T>
void unique_copy_partition_test(const std::vector<T>& a)
{
  auto b = unique_copy(a);
  auto unzipped = unique_copy_create_partition(a);
  assert(unzipped.first == b);

  const std::vector<size_t>& p = unzipped.second;

  // partition sequence is one element longer than output array
  assert(p.size() == b.size() + 1);

  // monotonicity (first and last element only)
  assert(p.front() == 0);
  assert(p.back()  == a.size());

  for (size_t i = 0; i < b.size(); ++i) {
    // consider i-th segment of the partition
    auto begin = p[i];
    auto end   = p[i + 1];

    // monotonicity
    assert(begin < end);

    // consecutive range of equal elements
    for (size_t k = begin; k < end; ++k) {
      assert(a[begin] == a[k]);
    }

    // maximal consecutive range of equal elements
    if (i + 1 < b.size()) {
      assert(a[begin] != a[end]);
    }

    // result of unique_copy
    assert(b[i] == a[begin]);
  }
}
```

Listing 2.13: Partition testing of unique_copy

We then check whether the first element of partitioning sequence equals 0 and whether the last element equals the size of the input sequence. This is, of course, in accordance with the chain of (in)equalities from Relation 1.1. Finally, we check for each partition segment $[p_i, p_{i+1})$ the rest of the monotonicity conditions from Relation (1.1), and then proceed to verify the properties (1.2), (1.3), and (1.4).

Listing 2.14 shows the code for executing partition tests with test data read from a file. As test data we use again the inputs from Listing 2.7.

```
int main(int argc, char** argv)
{
  assert(argc == 2);
  std::fstream file(argv[1]);

  while (true) {
    std::vector<int> v;
    file >> v;
    if (file) {
      // std::cout << v << std::endl;
      unique_copy_partition_test(v);
    }
    else {
      break;
    }
  }

  std::cout << "\tsuccessful execution of " << argv[0] << "\n";

  return EXIT_SUCCESS;
}
```

Listing 2.14: Test execution code for partition tests

3. Formal verification with Frama-C/WP

In this chapter we discuss the formal verification of `unique_copy` with Frama-C/WP. The first issue is that while `std::unique_copy` is implemented in C++ and heavily relies on C++ templates, Frama-C/WP can only deal with C functions. For this reason we present in Section 3.1 an implementation of `unique_copy` in C. As the verification platform Frama-C comes with its own formal specification language ACSL [5], we briefly explain in Section 3.2 the main elements of an ACSL function contract.

We discuss our first and simplest version of an ACSL specification of `unique_copy` in Section 3.3. The main idea of a so-called *minimal contract* is that our formal specification is just strong enough to verify the absence of certain undefined behaviors, such as illegal memory accesses and integer overflows. More specifically this means that our minimal contract for `unique_copy` formalizes **Unique Copy Size** and **Unique Copy Separation** but only partially formalizes **Unique Copy Return**. The formalization of the core requirement **Unique Copy Consecutive** is not addressed at all. Still, the verification of a minimal contract is meaningful since the addressed undefined behaviors are often a cause for security vulnerabilities. The verification of the *absence* of these undesirable behaviors can play an important role for ensuring the robustness of software.

In Section 3.4, we extend the minimal contract from Section 3.3 by a postcondition that states that the output range of `unique_copy` will not contain any adjacent equal elements. In other words, the new contract also partially addresses **Unique Copy Consecutive**.

Finally, in Section 3.5 we present a further extension of our contract that also captures the missing aspects of **Unique Copy Return** and **Unique Copy Consecutive** in the specification of `unique_copy`. Here, we build on top of our analysis of the so-called *partitioning sequence* from Section 1.3.

3.1. Reformulation of the algorithms in C

Our reformulation of `unique_copy` in the C programming language has a signature that can be considered as a specialisation of our simplified generic implementation of Listing 1.2. Instead of the generic type parameter `T`, however, we employ the integer type alias `value_type`. We also replace the standard unsigned integer type `size_t` by the type alias `size_type`. These type aliases are defined in Listing 3.1.

```
typedef int bool;

typedef int value_type;

typedef unsigned int size_type;
```

Listing 3.1: Definition of some basic type aliases

The basic idea of our C-implementation of `unique_copy` in Listing 3.2 is to traverse the input array `a[0...n-1]` and copy an element `a[i]` to the output array `b[0..n-1]` whenever it has been detected that it is different from its predecessor `a[i-1]`. Assuming a non-empty array, the implementation starts with copying `a[0]` to `b[0]`. In order to detect whether the current value `a[i]` is different from its predecessor we compare it with the most recently copied value `b[k]`.

```
size_type unique_copy(const value_type* a, size_type n, value_type* b)
{
  if (n == 0u) {
    return n;
  }
  else {
    size_type k = 0u;
    b[k] = a[0];
    for (size_type i = 1u; i < n; ++i) {
      const value_type val = a[i];
      if (b[k] != val) {
        b[++k] = val;
      }
    }
    return ++k;
  }
}
```

Listing 3.2: Re-implementation of `unique_copy` in C

Note that the test code in Chapter 2 has be designed in such a way that it can be applied also to a function with the signature in Listing 3.2.

3.2. Elements of ACSL contracts

Listing 3.3 shows the main elements of an ACSL function contract.

```
/*@
  requires preconditions;

  assigns locations;

  ensures postconditions;
*/
result func(arguments);
```

Listing 3.3: Main elements of an ACSL function contract

The *requires* clauses state the preconditions that must be satisfied in the caller context in order to expect that the function properly works. The *assigns* clauses list memory locations that can be modified by the function. Assignment clauses are a key element to describe the *side effects* of a function. To specify that a function does not change any memory locations the empty memory location `\nothing` is used. Note that `assigns \nothing;` is *very* different from providing no assignment clause. The latter means that the content of *any* memory location can change when the function is called. Finally, the *ensures* clauses express which postconditions shall hold after the function has been called.

3.3. A minimal contract for `unique_copy`

When we talk about a *minimal contract* of a function we mean a small contract that covers only basic properties. One might, for example, only be interested that during the execution of a function no runtime errors such as arithmetic overflows or invalid pointer accesses occur. Since many software security problems are caused by undetected runtime errors, minimal contracts can help to achieve a higher degree of quality assurance.

More specifically this means that our minimal contract for `unique_copy` formalizes the requirements **Unique Copy Size** and **Unique Copy Separation** but only partially formalizes **Unique Copy Return**. The formalization of the core requirement **Unique Copy Consecutive** is not addressed at all.

3.3.1. Formal specification

Listing 3.4 shows the specification of our minimal contract. We have *labeled* the various preconditions and postconditions of our contract by names, e.g., we use the label `sep` in order to refer to our formal specification of **Unique Copy Separation**. Using these user-supplied labels simplifies the documentation of contracts and can also be helpful during the process of formal verification. In the following we often refer to the various formal properties in a contract by their labels.

```
/*@
  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 result:     0 <= \result <= n;
  ensures unchanged: Unchanged{Old, Here}(b, \result, n);
*/
size_type
unique_copy(const value_type* a, size_type n, value_type* b);
```


Listing 3.4: A “minimal“ contract for `unique_copy`

Using the built-in predicates `\valid` and `\valid_read`, the preconditions `valid` state that

1. the elements of the input range `a[0..n-1]` can be safely accessed for reading,
2. whereas the elements of the output range `b[0..n-1]` can be safely accessed both for reading and writing.

Note that in accordance with **Unique Copy Size** both ranges have the same size. The informal specification also states in **Unique Copy Separation** that the input and output ranges do not overlap. This precondition is expressed by our property `sep` which in turn uses the built-in predicate `\separated`.

The `assigns` clause of our minimal contract states that `unique_copy` can only modify the array `b[0..n-1]`. Note that this requirement on the side effects of `unique_copy` cannot be found in the requirements in Table 1.3. This can be attributed to the fact that little is known about internal side effects of the generic type parameter `T`. In our more specific situation with the concrete type `value_type` we can use the means of ACSL to restrict the side effects allowed by our contract.

The postcondition `result` describes the numerical range for the return value of `unique_copy`. In other words, our minimal contract only provides a rather coarse estimation of **Unique Copy Return** for the number of elements copied by `unique_copy`. Note the use the ACSL keyword `\result` in this postcondition to refer to the return value of function. Also note that we have employed a *chained inequality* instead of writing

```
0 <= \result && \result <= n
```

This is a nice, little feature that helps writing compact contracts. There is a second postcondition `unchanged` that is formulated using the *user defined* ACSL predicate `Unchanged` [6, §6.1] that we show here in Listing 3.5. This predicate comes in the form of two overloaded versions. The first one is defined for an array section wheres the second one only requires the length of the array. The arguments `K` and `L` of the predicate are *labels* that represent *program states*. The predicate `Unchanged` says the respective elements of the array have the same value in state `K` and state `L`.

```
/*@
  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);
*/
```

Listing 3.5: Definition of the predicate `Unchanged`

We use the predicate `Unchanged` in order to make the assigns clause a bit more precise. Using the predefined labels `Old`, which refers to the pre-state of the contract, and `Post`, which refers to the post-state of the contract, the postcondition `unchanged` says that element of the range `b[\result..n-1]` are the same before and after `unique_copy` has been called. All this would easier if we could use just the assigns clause

```
assigns b[0..\result-1];
```

since this would imply our postcondition

```
ensures unchanged: Unchanged{Old, Here}(b, \result, n);
```

We remark here that the ACSL documentation does not forbid the use of `\result` outside of `ensures` clauses [5, p. 30]. While Frama-C/WP does not reject it either, the corresponding proof obligations are, in any case, not verified.

3.3.2. Graphical presentation of the minimal contract

Figure 3.6 is an attempt to graphically represent the minimal contract from Listing 3.4. In contrast to Figure 1.4, it is not indicated which elements of the input array have been copied to which elements of the output array. This is because, it has not been specified, whether any element at all has been copied. On the other hand, the minimal contract ensures that the part of the output array, that is not needed to hold the result, is kept unchanged.

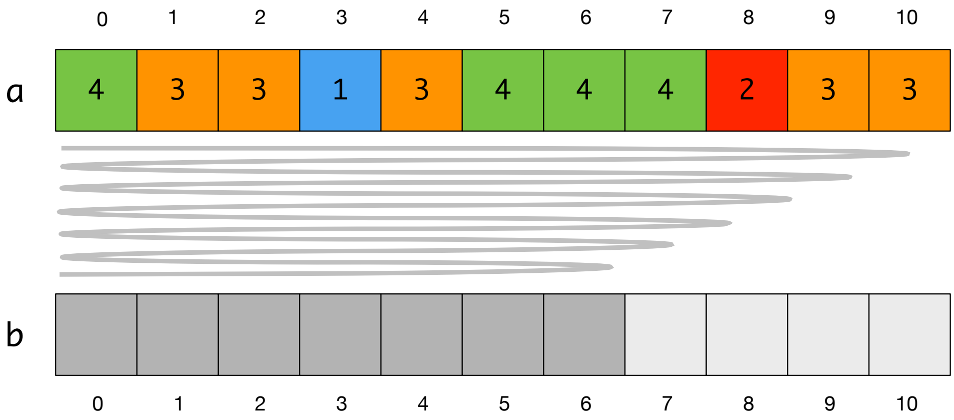

Figure 3.6.: Representation of a minimal contract for `unique_copy`

3.3.3. Annotating the implementation

The verification of the contract in Listing 3.4 requires that we add appropriate *loop annotations* to the implementation in Listing 3.7. Among these annotations are so-called *loop invariants*, which are formulas that must hold at the beginning of each loop iteration.

```
size_type unique_copy(const value_type* a, size_type n, value_type* b)
{

  if (n == 0u) {
    return n;
  }
  else {
    size_type k = 0u;
    b[k] = a[0];

    /*@
      loop invariant bound:     0 <= k < i <= n;
      loop invariant unchanged: Unchanged{Pre, Here}(b, k+1, n);
      loop assigns i, k, b[0..n-1];
      loop variant n-i;
    */
    for (size_type i = 1u; i < n; ++i) {
      const value_type val = a[i];
      if (b[k] != val) {
        b[++k] = val;
      }
    }

    return ++k;
  }

}
```

Listing 3.7: Annotations for the minimal contract

We now have a closer look at the loop annotations.

- The loop invariant `bound` states that the index `k` is always less than `i` that both are limited from below and above by `0` and the array length `n`, respectively.
- Similar to the postcondition `unchanged` in Listing 3.4 the loop invariant `unchanged` states that in each iteration of the loop the values in the range `b[k+1..n]` will be the same as in the pre-state of `unique_copy`. Note the use of the predefined label `Pre` to denote the program state before the function was called.
- We also need a *loop assigns clause* which lists all memory locations that can be changed during the execution of the loop. These memory locations comprise not only the array `b[0..n-1]` but also the local variables `i` and `k`.
- Finally, we have a *loop variant* which must contain a positive value that is decreased in each loop iteration. Loop variants serve to verify the *termination* of loops.

3.3.4. Verifying the absence of undefined behavior

The WP plugin[7] of Frama-C (in short also Frama-C/WP) is activated by using the option `-wp`. However, before Frama-C/WP starts generating and discharging proof obligations, the Frama-C kernel produces a *normalized version* of the source code. Most Frama-C plugins, including Frama-C/WP, use this semantically equivalent presentation to conduct their respective analyses.

Listing 3.8 shows the normalized version of the formal specification from Listing 3.4 and the annotated implementation from Listing 3.7. In order to extract the normalized version, which is also shown in Frama-C GUI, one can use the option `-print`. One of the most visible differences between the original and the normalized form is that *for loops* are represented as *while loops*. For more details on the normalization process we refer to the respective section of the Frama-C manual [8, §5.3].

```
/*@ 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));
    ensures result: 0 <= \result <= \old(n);
    ensures unchanged: Unchanged{Old, Here}(\old(b), \result, \old(n));
    assigns *(b + (0 .. n - 1));
 */
size_type unique_copy(value_type const *a, size_type n, value_type *b)
{
  size_type __retres;
  if (n == 0u) {
    __retres = n;
    goto return_label;
  }
  else {
    size_type k = 0u;
    *(b + k) = *(a + 0);
    {
      size_type i = 1u;
      /*@ loop invariant bound: 0 <= k < i <= n;
          loop invariant unchanged: Unchanged{Pre, Here}(b, k + 1, n);
          loop assigns i, k, *(b + (0 .. n - 1));
          loop variant n - i;
      */
      while (i < n) {
        {
          value_type const val = *(a + i);
          if (*(b + k) != val) {
            k ++;
            *(b + k) = val;
          }
        }
        i ++;
      }
    }
    k ++;
    __retres = k;
    goto return_label;
  }
return_label:
  return __retres;
}
```


Listing 3.8: Normalized presentation of the minimal contract

The Frama-C/WP plugin allows to generate additional assertions that are placed as guards before potentially dangerous C constructs, such as pointer dereferencing of integer operations that might overflow. Listing 3.9 shows these additional assertions in the normalized version of `unique_copy` when using the options

```
-wp-rte -warn-unsigned-overflow -warn-unsigned-downcast
```

For more details how to customize the generation of RTE (runtime error) guards we refer to the respective manuals [7, 9].

```
/*@ 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));
    ensures result: 0 <= \result <= \old(n);
    ensures unchanged: Unchanged{Old, Here}(\old(b), \result, \old(n));
    assigns *(b + (0 .. n - 1));
 */
size_type unique_copy(value_type const *a, size_type n, value_type *b)
{
  size_type __retres;
  if (n == 0u) {
    __retres = n;
    goto return_label;
  }
  else {
    size_type k = 0u;
    /*@ assert rte: mem_access: \valid(b + k); */
    /*@ assert rte: mem_access: \valid_read(a + 0); */
    *(b + k) = *(a + 0);
    {
      size_type i = 1u;
      /*@ loop invariant bound: 0 <= k < i <= n;
          loop invariant unchanged: Unchanged{Pre, Here}(b, k + 1, n);
          loop assigns i, k, *(b + (0 .. n - 1));
          loop variant n - i;
      */
      while (i < n) {
        {
          /*@ assert rte: mem_access: \valid_read(a + i); */
          value_type const val = *(a + i);
          /*@ assert rte: mem_access: \valid_read(b + k); */
          if (*(b + k) != val) {
            /*@ assert rte: unsigned_overflow: k + 1 <= 4294967295; */
            k ++;
            /*@ assert rte: mem_access: \valid(b + k); */
            *(b + k) = val;
          }
        }
        /*@ assert rte: unsigned_overflow: i + 1 <= 4294967295; */
        i ++;
      }
    }
    /*@ assert rte: unsigned_overflow: k + 1 <= 4294967295; */
    k ++;
    __retres = k;
    goto return_label;
  }
return_label:
  return __retres;
}
```

Listing 3.9: Normalized presentation of the minimal contract with RTE assertions

Verifying the minimal contract with the additional run time error assertions essentially shows that a large class of undefined behaviors cannot occur *if* the preconditions of the contract are satisfied. Since undefined behaviors often represent security vulnerabilities, even the verification of the minimal contract can, thus, provide significant evidence that the execution of a function such as `unique_copy` cannot cause security weaknesses.

3.4. A more elaborate contract for `unique_copy`

After using the minimal contract to prove the absence of undefined behavior we now show that there are no equal neighbors in the output array `b[0..\result-1]`. This property reflects an important consequence of **Unique Copy Consecutive**. It does, however, neither express the fact that only the first element of every consecutive range within the input array is copied nor does it state that the elements in the output range are at all related to those from the input range.

In order to formalize this new property we use the new predicate `HasEqualNeighbors` [6, §3.4.1] which we show here in Listing 3.10. The predicate states that there exists an element in the range `a[0..n-1]` which is equal to its direct successor.

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

Listing 3.10: The predicate `HasEqualNeighbors`

3.4.1. Formal specification

Listing 3.11 is an extension of our minimal contract. We keep all properties but also add the new postcondition `unique` to our contract.

```
/*@
  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 result:    0 <= \result <= n;
  ensures unique:    !HasEqualNeighbors (b, \result);
  ensures unchanged: Unchanged{Old, Here}(b, \result, n);
 */
size_type
unique_copy(const value_type* a, size_type n, value_type* b);
```

Listing 3.11: A more elaborate contract for `unique_copy`

In order to formally express the new postcondition we use the negation of `HasEqualNeighbors` from Listing 3.10.

3.4.2. Graphical presentation of the more elaborate contract

Figure 3.12 is an attempt to graphically represent the more elaborate specification from Listing 3.11. Compared to Figure 3.6 our new figure highlights that neighbouring elements of the output array are not equal. Our figure, however, still not indicates which elements of the input array have been copied to which elements of the output array.

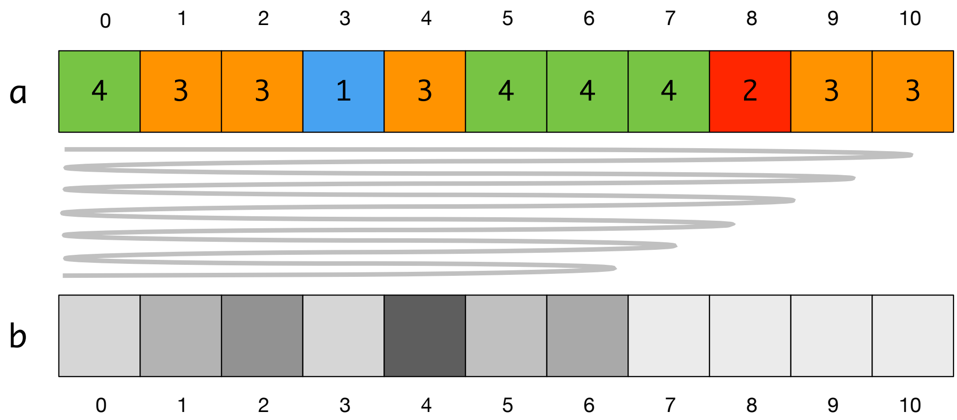

Figure 3.12.: Representation of a more elaborate contract for `unique_copy`

3.4.3. Annotating the implementation

In order to support the verification of the postcondition `unique` from Listing 3.11 we add a loop invariant that is also labeled as `unique` to our implementation 3.13. This loop invariant states that in each loop iteration there are no adjacent equal elements in the range `b[0..k]` of already copied elements. Not surprisingly, we are using the predicate `HasEqualNeighbors` to formally describe this property.

```
size_type
unique_copy(const value_type* a, size_type n, value_type* b)
{
  if (n == 0u) {
    return n;
  }
  else {
    size_type k = 0u;

    b[k] = a[0];

    /*@
      loop invariant bound:     0 <= k < i <= n;
      loop invariant unique:    !HasEqualNeighbors(b, k+1);
      loop invariant unchanged: Unchanged{Pre, Here}(b, k+1, n);
      loop assigns i, k, b[0..n-1];
      loop variant n-i;
    */
    for (size_type i = 1; i < n; ++i) {
      const value_type val = a[i];
      if (b[k] != val) {
        b[++k] = val;
      }
    }
    return ++k;
  }

}
```

Listing 3.13: Annotations for a more elaborate contract of `unique_copy`

3.5. A complete contract for `unique_copy`

In this section we finally tackle the issue of formalizing the requirements of **Unique Copy Consecutive** in ACSL. The main idea is that we

1. capture in ACSL the properties of the partitioning sequence from Section 1.3
2. use these properties to specify and verify `unique_copy`

3.5.1. Formalizing the number of elements copied by `unique_copy`

The logic function `UniqueSize` in Listing 3.14 computes the number of elements that are to be copied by `unique_copy` from an array `a[0..n-1]` In other words, `UniqueSize` represents the number *m* in the inequalities (1.1) of consecutive sub-ranges of equal elements. Our axioms in Listing 3.14 essentially give a *recursive* definition of `UniqueSize`.

```
/*@
  axiomatic UniqueSizeAxiomatic
  {
    logic integer UniqueSize(value_type* a, integer n) reads a[0..n-1];

    axiom UniqueSizeEmpty:
      \forall value_type *a, integer n;
        n <= 0  ==>  UniqueSize(a, n) == 0;

    axiom UniqueSizeOne:
      \forall value_type *a, integer n;
        n == 1  ==>  UniqueSize(a, n) == 1;

    axiom UniqueSizeEqual:
      \forall value_type *a, integer n;
        0 < n ==> a[n-1] == a[n] ==> UniqueSize(a, n+1) == UniqueSize(a, n);

    axiom UniqueSizeDiffer:
      \forall value_type *a, integer n;
        0 < n ==> a[n-1] != a[n] ==>  UniqueSize(a, n+1) == UniqueSize(a, n) + 1;

    axiom UniqueSizeRead{K,L}:
      \forall value_type *a, integer n, i;
        Unchanged{K,L}(a, n) ==> UniqueSize{K}(a, n) == UniqueSize{L}(a, n);
  }
*/
```

Listing 3.14: Axiomatic description of the function `UniqueSize`

Note that the axioms of `UniqueSize` cover also negative array size, e.g. Axiom `UniqueSizeEmpty`, and in general ignore whether the involved pointers can be dereferenced. The issue here is that the function `UniqueSize` must be defined as a *total function* regardless of the fact whether the involved function arguments make sense in C-code. For more details we refer to the rules for logic definitions in the description of ACSL [5, §2.2.2].

The following ACSL lemmas `UniqueSizeBound` (Listing 3.15) formulates some simple bounds on the number of copied elements. As trivial as these inequalities might look like, their not too complicated proofs rely on mathematical induction. Since automatic theorem provers are often not capable of performing induction proofs, we have proven this lemma with the interactive theorem prover Coq.

```
/*@
  lemma UniqueSizeBound:
    \forall value_type *a, integer n;
      0 <= n  ==>  0 <= UniqueSize(a, n) <= n;
*/
```

Listing 3.15: Lemma `UniqueSizeBound`

3.5.2. Formalizing the properties of the partitions of `unique_copy`

The function `UniquePartition`, whose axiomatic definition is given in Listing 3.16, defines the partitioning sequence *p* from Section 1.3. Before we begin to relate the axioms from Listing 3.16 to the formulas from Section 1.3 we want to remind the reader that logic functions (and predicates) must be total that is they must be defined for all possible argument values.

```
/*@
  axiomatic UniquePartitionAxiomatic
  {
    logic integer
      UniquePartition(value_type* a, integer n, integer i) reads a[0..n-1];

    axiom UniquePartitionEmpty:
      \forall value_type *a, integer n, i;
        n <= 0  ==>  UniquePartition(a, n, i) == 0;

    axiom UniquePartitionLeft:
      \forall value_type *a, integer n, i;
        0 < n  ==>  i <= 0  ==>  UniquePartition(a, n, i) == 0;

    axiom UniquePartitionRight:
      \forall value_type *a, integer n, i;
        0 < n  ==>  UniqueSize(a, n) <= i  ==>  UniquePartition(a, n, i) == n;

    axiom UniquePartitionMonotone:
      \forall value_type *a, integer n, i, j;
        0 <= i < j <= UniqueSize(a, n)  ==>
        UniquePartition(a, n, i) < UniquePartition(a, n, j);

    axiom UniquePartitionSegment:
      \forall value_type *a, integer n, i, k;
        0 <= i < UniqueSize(a, n) ==>  \let pi = UniquePartition(a, n, i);
        pi <= k < UniquePartition(a, n, i+1) ==> a[pi] == a[k];

    axiom UniquePartitionMaximal:
      \forall value_type *a, integer n, i;
        0 <= i < UniqueSize(a, n) - 1  ==>
        a[UniquePartition(a, n, i)] != a[UniquePartition(a, n, i+1)];

    axiom UniquePartitionEqual:
      \forall value_type *a, integer n, m, i;
        n < m  ==>  0 <= i < UniqueSize(a, n)  ==>
        UniquePartition(a, n, i) == UniquePartition(a, m, i);

    axiom UniquePartitionRead{K,L}:
      \forall value_type *a, integer n, i;
        Unchanged{K,L}(a, n) ==>
          UniquePartition{K}(a, n, i) == UniquePartition{L}(a, n, i);
  }
*/
```

Listing 3.16: Axiomatic description of the function `UniquePartition`

- The monotonicity conditions (1.1) are described by the axioms UniquePartitionEmpty, UniquePartitionLeft, UniquePartitionRight and UniquePartitionMonotone.
- Equation (1.2) is represented by the axiom UniquePartitionSegment.
- Inequality (1.3) is described by axiom UniquePartitionMaximal.
- Axiom UniquePartitionEqual expresses that the value of UniquePartition(a, n, i) does not depend on the size of the array.
- Axiom UniquePartitionRead, finally states that UniquePartition is independent from the particular programme state in which it is used—as long as the respective array elements are equal in both states.

With the definitions of the logic functions UniqueSize and UniquePartition we can now formulate the ACSL predicate Unique from Listing 3.17. This predicate reflects Equation (1.4) and therefore will serve a prominent role in our complete contract of unique_copy.

```
/*@
  predicate
    Unique(value_type* a, integer n, value_type* b) =
      \forall integer k; 0 <= k < UniqueSize(a, n) ==>
        b[k] == a[UniquePartition(a, n, k)];
*/
```

Listing 3.17: The predicate Unique

Before we turn, however, our attention to the contract of unique_copy we show in Listing 3.18 a couple of simple ACSL lemmas that will be helpful in verifying the new contract.

```
/*@
  lemma UniquePartitionZero:
     \forall value_type *a, integer n;
       UniquePartition(a, n, 0) == 0;

  lemma UniquePartitionLowerBound:
     \forall value_type *a, integer n, i;
       0 < n ==>
       0 <= i < UniqueSize(a, n) ==>
       0 <= UniquePartition(a, n, i);

  lemma UniquePartitionUpperBound:
     \forall value_type *a, integer n, i;
       0 < n ==>
       0 <= i < UniqueSize(a, n) ==>
       UniquePartition(a, n, i) < n;
*/
```

Listing 3.18: Some lemmas regarding UniquePartition

3.5.3. Formal specification

Listing 3.19 shows how we use the predicate Unique in the postcondition unique in order to formally specify **Unique Copy Consecutive** for unique_copy.

```
/*@
  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 bound:     0 <= \result <= n;
  ensures size:      \result == UniqueSize(a, n);
  ensures unique:    Unique(a, n, b);
  ensures unchanged: Unchanged{Old, Here}(a, n);
  ensures unchanged: Unchanged{Old, Here}(b, \result, n);
 */
size_type
unique_copy(const value_type* a, size_type n, value_type* b);
```

Listing 3.19: A complete contract for unique_copy

A natural question is whether our postcondition unique is a generalization of the postcondition with the same name from the contract 3.11. Fortunately, this question can be answered in the affirmative. In fact, Lemma UniqueImpliesNoEqualNeighbors from Listing 3.20 states exactly the desired implication.

```
/*@
  lemma UniqueImpliesNoEqualNeighbors:
    \forall value_type *a, *b, integer n;
       Unique(a, n, b)  ==>  !HasEqualNeighbors(b, UniqueSize(a, n));
*/
```

Listing 3.20: The predicate UniqueImpliesNoEqualNeighbors

3.5.4. Annotating the implementation

Listing 3.22 shows that we need considerably more annotation in order to verify the contract from Listing 3.19. We also rely on Lemma UnchangedSection [6, §6.1] that we show here in Listing 3.21.

```
/*@
  lemma
    UnchangedSection{K,L}:
      \forall value_type *a, integer m, n, p, q;
         0 <= m <= p <= q <= n     ==>
         Unchanged{K,L}(a, m, n)   ==>
         Unchanged{K,L}(a, p, q);
*/
```

Listing 3.21: Lemma UnchangedSection

```
size_type
unique_copy(const value_type* a, size_type n, value_type* b)
{
  if (n == 0u) {
    return n;
  }
  else {
    size_type k = 0u;
    b[k] = a[0];
    //@ assert mapping:   0 == UniquePartition(a, n, k);

    /*@
      loop invariant bound:      0 <= k < i <= n;
      loop invariant size:       k+1 == UniqueSize(a, i);
      loop invariant copy:       b[k] == a[i-1];
      loop invariant mapping:    UniquePartition(a, n, k) < i;
      loop invariant mapping:    i <= UniquePartition(a, n, k+1);
      loop invariant unique:     Unique(a, i, b);
      loop invariant unchanged: Unchanged{Pre, Here}(b, k+1, n);
      loop assigns i, k, b[0..n-1];
      loop variant n-i;
    */
    for (size_type i = 1u; i < n; ++i) {
      const value_type val = a[i];
      if (b[k] != val) {
        //@ assert distinct:  a[i-1] != a[i];
        //@ ghost Before:
        b[++k] = val;
        //@ assert unchanged: Unchanged{Before,Here}(b, k);
        //@ assert unchanged: Unchanged{Before,Here}(a, n);
        //@ assert mapping:   i == UniquePartition(a, n, k);
        //@ assert size:      k == UniqueSize(a, i);
        //@ assert unique:    Unique(a, i, b);
      }
    }

    return ++k;
  }
}
```

Listing 3.22: Annotations for the complete contract of `unique_copy`

The annotations in Listing 3.22 come not only in the form of loop invariants or ACSL assertions. We also employ so-called *ghost code* whose purpose we will explain now. As explained in the ACSL documentations [5, §2.12], variables and statements that appear in comments marked as

```
/*@ ghost ... */
```

or

```
//@ ghost ...
```

are treated as C variables and statements, however, they are visible only in the specifications. In Listing 3.22 we declare the label `Before` as ghost. We could also have resorted to ACSL *statement contracts* [5, §2.4.4] but opted here for using ghost code.

3.6. Results of formal verification

This section gives all settings that depend on the software release of Frama-C, Why3, or one of the employed provers. For our experiments we used the WP plug-in of Frama-C [1, version 17.1] together with the Why3 [10, version 0.88.3] verification platform.

Here are the most important options of Frama-C that we used in our experiments.

```
-pp-annot -no-unicode
-wp -wp-rte -wp-model Typed+ref
-warn-unsigned-overflow -warn-unsigned-downcast
-wp-timeout 10 -wp-steps 1000 -wp-coq-timeout 10
```

Table 3.23 lists the various provers that we used to discharge the proof obligations.

Prover	Type	Version	Reference
Alt-Ergo	automatic	2.2.0	[11]
CVC4	automatic	1.6	[12]
CVC3	automatic	2.4.1	[13]
Z3	automatic	4.8.1	[14]
Coq	interactive	8.7.2	[15]

Table 3.23.: Provers used in during verification

Table 3.24 shows some statistics on how the proof obligattions were discharged by the provers. There are two things to note here.

1. The table also contains a column for the built-in simplifier Qed of Frama-C/WP.
2. For each proof obligation, the simplifier/provers are executed by Frama-C/WP in the order

   Qed $\mapsto$ Alt-Ergo $\mapsto$ CVC4 $\mapsto$ CVC3 $\mapsto$ Z3 $\mapsto$ Coq

   until the proof obligation has been discharged.

Algorithm	Verification Conditions		Individual Provers					
			Qed	Alt-Ergo	CVC4	CVC3	Z3	Coq
§3.3	23 / 23	100	8	15				
§3.4	26 / 26	100	8	18				
§3.5	50 / 50	100	9	29	5	1	2	4

Table 3.24.: Some statistics on the used provers

4. Conclusions — Frama-C and Testing

We have investigated in this report at some lengths various aspects of testing and formal verification of `unique_copy` — a not too complicated algorithm from the C++ standard library. Testing and formal verification are sufficiently different techniques to assure the quality of software. Unsurprisingly, however, they both rely on an in-depth analysis of the (informal) requirements. We have conducted such an analysis in Chapter 1.

This analysis allowed us in Chapter 2 to derive both test code and test data that capture the core aspects of the algorithm under investigation. Similarly, we have shown in Chapter 3 that, depending on the properties that one wishes to verify, there are various ways to come up with a formal specification.

Ideally, formal verification and testing should go hand in hand. In particular, the process of finding the necessary code annotations involves a lot of guessing whose results are best checked with some test data before one ventures to prove them. The Frama-C verification platform provides the E-ACSL plug-in [16] that shows how this goal can be achieved.

Unfortunately, E-ACSL does not yet provide sufficient support to straightforwardly apply it in this case study. However, there is no principal gap that would hinder its use in a synthesis of *dynamic* and *formal* analyses. Applying such a synthesis to more algorithms and data structures from the C++ standard library would of course also require proper support from Frama-C for C++ code. As of now, the Frama-Clang plug-in [17] serves as a prototype that can parse (annotated) C++ code and conduct simple analyses.

A. Mathematical definition of `unique_copy`

We have here a more mathematical look at the properties of partitioning sequences from Section 1.3. We show, in particular, that for each nonempty array there exists a uniquely determined partitioning sequence.

Lemma 1 (partitioning sequence) *Let X be a nonempty set and $a = (a_0, \ldots, a_{n-1})$ a sequence of nonzero length n in X. There exists a uniquely determined sequence $p = (p_0 \ldots, p_m)$ of $m + 1$ indices, with $0 \le m \le n$, which has the following properties.*

The sequence p is strictly increasing

$$0 = p_0 < \ldots < p_m = n \tag{A.1}$$

The elements of p partition *the sequence a into segments of equal elements*

$$a_k = a_{p_i} \qquad \forall i, k : p_i \le k < p_{i+1} \land 0 \le i < m \tag{A.2}$$

These segments are maximal *in the following sense*

$$a_{p_i} \ne a_{p_{i+1}} \qquad \forall i : 0 \le i < m - 1 \tag{A.3}$$

Proof

We start with showing by mathematical induction the existence of a sequence $p = (p_0 \ldots, p_m)$ that satisfies the relations (A.1), (A.2), and (A.3).

1. If $a = (a_0)$ is a sequence of length 1, then we define the sequence p as $p = (0, 1)$.
2. Let us assume that for $a = (a_0, \ldots, a_{n-1})$ a sequence $p = (p_0, \ldots, p_m)$ with the desired properties exists. We consider now the longer sequence $(a_0, \ldots, a_{n-1}, x)$.
   a) If $x = a_{n-1}$, then we define the sequence $p' = (p'_0, \ldots, p'_m)$ by simply requiring $p'_i = p_i$ for $0 \le i < m$ and $p'_m = n + 1$.
   b) If, on the other hand, we have $x \ne a_{n-1}$, then we define the sequence $p' = (p'_0, \ldots, p'_m, p'_{m+1})$ by requiring $p'_i = p_i$ for $0 \le i \le m$ and $p'_{m+1} = n + 1$.

   In both cases the relations (A.1), (A.2), and (A.3) also hold for the sequence p'.

We now prove the uniqueness of the sequence p. Let $p = (p_0, \ldots, p_m)$ and $q = (q_0, \ldots, q_r)$ be two sequences that satisfy the relations (A.1), (A.2), and (A.3). We will show by an indirect proof that $p = q$ holds.

1. We assume at first that both sequences have the same length, that is, $m = r$. Let k be the smallest index with $p_k \ne q_k$. It follows then from relation (A.1) that $0 < k < m$ holds. Let without loss of generality be $q_{k-1} < p_k < q_k$. From Inequality (A.3) we have $a_{p_k} \ne a_{p_{k-1}} = a_{q_{k-1}}$. On the other hand we have according to Equation (A.2) the relation $a_{p_k} = a_{q_{k-1}}$. This contradiction shows that not just the lengths of p and q are equal but the sequences themselves.

2. We assume now, without loss of generality, that $m < r$ holds. We obtain from the Inequalities (A.1), on the one hand,

$$q_m < q_r = n = p_m$$

or in short

$$q_m < p_m$$

There is therefore a least index k with $0 < k \leq m$ such that

$$p_k \neq q_k$$

holds and we can apply the same steps as in the first case to reach a contradiction. ■

Based on this lemma we can for each sequence *mathematically* define the effect of `unique_copy`.

Definition 1 (`unique_copy`)

1. Let $a = (a_0, \dots, a_{n-1})$ be a nonempty sequence of length m and $p = (p_0, \dots, p_m)$ its *partitioning sequence*.

   The result of applying `unique_copy` to a is the sequence $b = (b_0, \dots, b_{m-1})$ which is defined as

   $$b_i = a_{p_i} \qquad \text{for } 0 \leq i < m \tag{A.4}$$

   In other words, b_i equal the first element of the i-th segment of equal elements of a.

2. Applying `unique_copy` to the empty sequence () is defined as the empty sequence. □

A simple consequence of Equations (A.3) and (A.4) is

$$b_i \neq b_{i+1} \qquad \text{for } 0 \leq i < m - 1$$

which expresses the fact that the result of `unique_copy` does not contain adjacent equal elements.

Bibliography

[1] Frama-C Software Analyzers. http://frama-c.com, 2018.

[2] ANSI/ISO C Specification Language. http://frama-c.com/acsl.html.

[3] Lawrence Crowl and Thorsten Ottosen. Working Draft, Standard for Programming Language C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf, 2013. publicly available draft of C++ 14 standard.

[4] Repository of "libc++" C++ Standard Library. https://llvm.org/svn/llvm-project/libcxx/trunk, 2018. Revision 345375: /libcxx/trunk.

[5] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL 1.13 Implementation in Chlorine-20180501. http://frama-c.com/download/acsl-implementation-Chlorine-20180501.pdf, May 2018.

[6] Jens Gerlach *et al.* ACSL by Example. Technical Report Version 17.2.0, Fraunhofer FOKUS, Jul 2018. https://github.com/fraunhoferfokus/acsl-by-example.

[7] Patrick Baudin, François Bobot, Loïc Correnson, and Zaynah Dargaye. WP Plug-in Manual — Frama-C Chlorine-20180501. http://frama-c.com/download/rte-manual-Chlorine-20180501.pdf, 2018.

[8] Loïc Correnson, Pascal Cuoq, Florent Kirchner, André Maroneze, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakobowski. Frama-C User Manual, Release Chlorine-20180501. http://frama-c.com/download/user-manual-Chlorine-20180501.pdf, 2018.

[9] Philippe Herrmann and Julien Signoles. Frama-C's annotation generator plug-in for Frama-C Chlorine-20180501. http://frama-c.com/download/frama-c-wp-manual.pdf, 2018.

[10] Why3 — Where Programs Meet Provers. http://why3.lri.fr, 2018.

[11] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. The Alt-Ergo SMT Solver. http://alt-ergo.lri.fr, 2018.

[12] Clark Barrett and Cesare Tinelli. Homepage of CVC4. http://cvc4.cs.stanford.edu/web/, 2018.

[13] Clark Barrett and Cesare Tinelli. Homepage of CVC3. http://www.cs.nyu.edu/acsys/cvc3/, 2010.

[14] Microsoft Research. The Z3 Theorem Prover. https://github.com/Z3Prover/z3, 2018.

[15] The Coq Consortium. The Coq Proof Assistant. https://coq.inria.fr, 2018.

[16] E-ACSL plug-in. http://frama-c.com/eacsl.html, 2018.

[17] The Frama-Clang plugin. http://frama-c.com/frama-clang.html, 2018.