ACSL By Example

Towards a Verified C Standard Library

Version 9.3.0
for
Frama-C Fluorine
December 2013

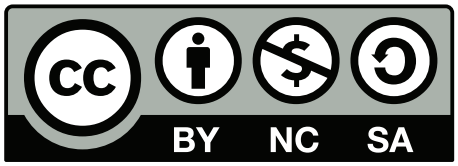
Jochen Burghardt
Andreas Carben
Jens Gerlach
Hans Pohl
Kim Völlinger

Former Authors

Liangliang Gu
Kerstin Hartig
Juan Soto

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no 317753.[1]

This body of work was completed within the DEVICE-SOFT project, which was supported by the Programme Inter Carnot Fraunhofer from BMBF (Grant 01SF0804) and ANR.[2]



Except where otherwise noted, this work is licensed under
http://creativecommons.org/licenses/by-nc-sa/3.0/

[1] project duration: 2012–2015
[2] project duration: 2009–2012

Foreword

This report provides various examples for the formal specification, implementation, and deductive verification of C programs using the ANSI/ISO-C Specification Language (ACSL [1]) and the WP plug-in [2] of Frama-C [3] (Framework for Modular Analysis of C programs). The report at hand has been revised and refers to the *Fluorine* release from June 2013 of Frama-C.

We have chosen our examples from the C++ Standard Library whose initial version is still known as the *Standard Template Library* (STL).[3] The STL contains a broad collection of *generic* algorithms that work not only on C arrays but also on more elaborate containers, i.e., data structures. For the purposes of this document we have selected representative algorithms, and converted their implementation from C++ function templates to C functions that work on arrays of type `int`.[4]

We will continue to extend and refine this report by describing additional STL algorithms and data structures. Thus, step by step, this document will evolve from an ACSL tutorial to a report on a formally specified and deductively verified standard library for ANSI/ISO-C. Moreover, should ACSL be extended to a C++ specification language, our work may be extended to a deductively verified C++ Standard Library.

You may email comments, suggestions or errors to

jens.gerlach@fokus.fraunhofer.de

In particular, we encourage you to check vigilantly whether our formal specifications capture the essence of the informal description of the STL algorithms.

We appreciate your feedback and hope that this document helps foster the adoption of deductive verification techniques.

[3] See http://www.sgi.com/tech/stl/

[4] We are not directly using `int` in the source code but rather `value_type` which is a `typedef` for `int`.

Contents

1. Introduction **9**
- 1.1. Structure of this Document . . . 10
- 1.2. Types, Arrays, Ranges and Valid Indices . . . 10

2. The Hoare Calculus **13**
- 2.1. The Assignment Rule . . . 15
- 2.2. The Sequence Rule . . . 17
- 2.3. The Implication Rule . . . 17
- 2.4. The Choice Rule . . . 18
- 2.5. The Loop Rule . . . 19
- 2.6. Derived Rules . . . 21

3. Non-mutating Algorithms **23**
- 3.1. The `equal` Algorithm . . . 24
- 3.2. The `mismatch` Algorithm . . . 28
- 3.3. The `find` Algorithm . . . 30
- 3.4. The `find` Algorithm Reconsidered . . . 32
- 3.5. The `find_first_of` Algorithm . . . 34
- 3.6. The `adjacent_find` Algorithm . . . 36
- 3.7. The `search` Algorithm . . . 38
- 3.8. The `count` Algorithm . . . 42

4. Maximum and Minimum Algorithms **45**
- 4.1. A Note on Relational Operators . . . 46
- 4.2. The `max_element` Algorithm . . . 48
- 4.3. The `max_element` Algorithm with Predicates . . . 50
- 4.4. The `max_seq` Algorithm . . . 52
- 4.5. The `min_element` Algorithm . . . 54

5. Binary Search Algorithms **57**
- 5.1. The `lower_bound` Algorithm . . . 58
- 5.2. The `upper_bound` Algorithm . . . 60
- 5.3. The `equal_range` Algorithm . . . 62
- 5.4. The `binary_search` Algorithm . . . 64

6. Mutating Algorithms **67**
- 6.1. The `fill` Algorithm . . . 68
- 6.2. The `swap` Algorithm . . . 70
- 6.3. The `swap_ranges` Algorithm . . . 72

6.4. The `copy` Algorithm . . . 76
6.5. The `reverse_copy` Algorithm . . . 78
6.6. The `reverse` Algorithm . . . 80
6.7. The `rotate_copy` Algorithm . . . 82
6.8. The `replace_copy` Algorithm . . . 84
6.9. The `remove_copy` Algorithm . . . 86
6.10. The `unique_copy` Algorithm . . . 90
6.11. The `iota` Algorithm . . . 94

7. The `Stack` Data Type **97**
7.1. Methodology Overview . . . 98
7.2. Stack Axioms . . . 99
7.3. The Structure `Stack` and its Associated Functions . . . 101
7.4. Stack Invariants . . . 102
7.5. Equality of Stacks . . . 104
7.6. Runtime Equality of Stacks . . . 106
7.7. Verification of Stack Functions . . . 107
7.8. Verification of Stack Axioms . . . 120

8. Formal Verification **125**

A. Changes **129**
A.1. New in Version 9.3.0 (December 2013) . . . 129
A.2. New in Version 8.1.0 (not published) . . . 129
A.3. New in Version 7.1.1 (August 2012) . . . 130
A.4. New in Version 7.1.0 (December 2011) . . . 130
A.5. New in Version 6.1.0 (not published) . . . 130
A.6. New in Version 5.1.1 (February 2011) . . . 130
A.7. New in Version 5.1.0 (May 2010) . . . 131
A.8. New in Version 4.2.2 (May 2010) . . . 131
A.9. New in Version 4.2.1 (April 2010) . . . 131
A.10. New in Version 4.2.0 (January 2010) . . . 132
A.11. Unofficial Release as part of the ES_PASS project (March 2009) . . . 132

Bibliography **133**

List of Logic Specifications

1.1. The Predicate `IsValidRange` 12

3.2. The Predicate `IsEqual` 25
3.11. The Predicate `HasValue` 32
3.14. The Predicate `HasValueOf` 34
3.17. The Predicate `HasEqualNeighbors` 36
3.20. The Predicate `HasSubRange` 38
3.24. Axiomatic Definition of `Count` 42

4.1. Requirements for a partial order on **`value_type`** 46
4.2. Semantics of Derived Comparison Operators 47
4.5. The `IsMaximum` and `IsFirstMaximum` Predicates 50

5.1. The Predicate `IsSorted` 57

6.20. The Logic Function `RemoveMap` 87
6.23. The Logic Function `UniqueWhither` 90
6.25. The Predicate `UniqueCopy` 92

7.5. The Logical Functions `CapacityOfStack`, `SizeOfStack` and `TopOfStack` 103
7.6. Predicates for Empty an Full Stacks 103
7.7. The Predicate `IsValidStack` 103
7.8. Equality of Stacks 104
7.10. Equality of Stacks is an Equivalence Relation 105

List of Figures

2.2. Example Source-Code Fragments and Annotations 13
2.3. Loop Source Code Fragments and Annotations 14
2.4. The Assignment Rule 15
2.5. An Assignment Rule Example Instance 15
2.6. Simpler, but *Faulty* Assignment Rule 16
2.7. An Example Instance of the Faulty Rule from Figure 2.6 16
2.8. An Example Instance of the Faulty Rule from Figure 2.6 16
2.9. An Example Instance of the Faulty Rule from Figure 2.6 17
2.10. The Sequence Rule 17
2.11. The Implication Rule 17
2.12. The Choice Rule 18
2.13. An Example Application of the Choice Rule 19
2.14. The Loop Rule 19
2.15. An Abstract Ring Buffer Loop 20

3.22. Explanation for Line 13 39

6.7. Loop Invariant Check Points 75
6.18. Stability of `remove_copy` 86
6.19. Stability of `remove_copy` with respect to indices 87
6.24. An Example of the Logic Function `UniqueWhither` 91

7.1. Push and Pop on a Stack 97
7.2. Methodology Overview 98
7.4. Interpretation of `Stack` 101
7.9. Example of Two Equal Stacks 104
7.13. Methodology for the verification of well-definition 107

List of Tables

2.1. Some ACSL Formula Syntax 13

8.1. Results for Non-mutating Algorithms 126
8.2. Results for Maximum and Minimum Algorithms 126
8.3. Results for Binary Search Algorithms 127
8.4. Results for Mutating Algorithms 127
8.5. Results for `Stack` functions 128
8.6. Results for the well-definition of the `Stack` functions 128
8.7. Results for Stack Axioms 128

1. Introduction

The Framework for Modular Analyses of C, Frama-C [3], is a suite of software tools dedicated to the analysis of C source code. Its development efforts are conducted and coordinated at two French public institutions: CEA LIST [4], a laboratory of applied research on software-intensive technologies, and INRIA Saclay[5], the French National Institute for Research in Computer Science and Control in collaboration with LRI [6], the Laboratory for Computer Science at Université Paris-Sud.

ACSL (ANSI/ISO-C Specification Language) [1] is a formal language to express behavioral properties of C programs. This language can specify a wide range of functional properties by adding annotations to the code. It allows to create function contracts containing preconditions and post-conditions. It is possible to define type and global invariants as well as logic specifications, such as predicates, lemmas, axioms or logic functions. Furthermore, ACSL allows statement annotations such as assertions or loop annotations.

Within Frama-C, the WP plug-in [2] enables deductive verification of C programs that have been annotated with ACSL. The WP plug-in uses Hoare-style weakest precondition computations to formally prove ACSL properties of C code. Verification conditions are generated and submitted to external automatic theorem provers or interactive proof assistants.

The Verification Group at Fraunhofer FOKUS[7] see the great potential for deductive verification using ACSL. However, we recognize that for a novice there are challenges to overcome in order to effectively use the WP plug-in for deductive verification. In order to help users gain confidence, we have written this tutorial that demonstrates how to write annotations for existing C programs. This document provides several examples featuring a variety of annotated functions using ACSL. For an in-depth understanding of ACSL, we strongly recommend users to read the official Frama-C introductory tutorial [8] first. The principles presented in this paper are also documented in the ACSL reference document [9].

Acknowledgement

Many members from the Frama-C community provided valuable input and comments during the course of the development of this document. In particular, we wish to thank our project partners from the DEVICE-SOFT[5] project, Virgile Prevosto, Pascal Cuoq, Armand Puccetti, Florent Kirchner and Benjamin Monate from CEA LIST, as well as the WP developers Loïc Correnson and Zaynah Dargaye (also CEA LIST).

We also like to express our gratitude to Claude Marché (LRI/INRIA) and Yannick Moy (AdaCore) for their helpful comments and detailed suggestions for improvement.

[5] See `http://www.fokus.fraunhofer.de/en/sqc/forschungsthemen/verification/index.html`

1.1. Structure of this Document

The functions presented in this document were selected from the C++ Standard Template Library (STL) [10]. The original C++ implementation was stripped from its generic implementation and mapped to C arrays of type **value_type**.

Chapter 2 provides a short introduction into the Hoare Calculus.

We have grouped various STL algorithms in chapters as follows:

- non-mutating algorithms (Chapter 3)
- maximum/minimum algorithms (Chapter 4)
- binary search algorithms (Chapter 5)
- mutating algorithms (Chapter 6)

The order of these chapters reflects their increasing complexity.

Using the example of a stack, we tackle in Chapter 7 the problem of how a data type and its associated C functions can be specified with ACSL and automatically verified with Frama-C.

1.2. Types, Arrays, Ranges and Valid Indices

This section describe several general conventions and basic definitions we use throughout this document.

1.2.1. Types

In order to keep algorithms and specifications as general as possible, we use abstract type names on almost all occasions. We currently defined the following types:

```
typedef int value_type;

typedef unsigned int size_type;

typedef int bool;
```

Programmers who know the types associated with STL containers will not be surprised that **value_type** refers to the type of values in an array whereas **size_type** will be used for the indices of an array.

This approach allows one to modify e.g. an algorithm working on an **int** array to work on a **char** array by changing only one line of code, viz. the **typedef** of **value_type**. Moreover, we believe in better readability as it becomes clear whether a variable is used as an index or as a memory for a copy of an array element, just by looking at its type.

The latter reason also applies to the use of **bool**. To denote values of that type, we `#defined` the identifiers `false` and `true` to be `0` and `1`, respectively. While any non-zero value is accepted

to denote `true` in ACSL like in C the algorithms shown in this tutorial will always produce `1` for `true`. We omit clauses like `\result == false || \result == true` from our function contracts for sake of brevity only. They still appear at places where they are needed for the provers, e.g. Listing 7.19. Due to the above definitions, the ACSL truth-value constant `\false` and `\true` can be used interchangeably with our `false` and `true`, respectively, in ACSL clauses, but not in C code.

1.2.2. Array and Ranges

The C Standard describes an array as a "contiguously allocated nonempty set of objects" [11, §6.2.5.20]. If `n` is a constant integer expression with a value greater than zero, then

```
int a[n];
```

describes an array of type **`int`**. In particular, for each `i` that is greater than or equal to 0 and less than `n`, we can dereference the pointer `a+i`.

Let the following prototype represent a function, whose first argument is the address to a range and whose second argument is the length of this range.

```
void example(value_type* a, size_type n);
```

To be very precise, we have to use the term *range* instead of *array*. This is due to the fact, that functions may be called with empty ranges, i.e., with `n == 0`. Empty arrays, however, are not permitted according to the definition stated above. Nevertheless, we often use the term *array* and *range* interchangeably.

1.2.3. Specification of Valid Ranges in ACSL

The following ACSL fragment expresses the precondition that the function `example` expects that for each `i`, such that `0 <= i < n`, the pointer `a+i` may be safely dereferenced.

```
/*@
    requires 0 <= n;
    requires \valid(a+(0.. n-1));
*/
void example(value_type* a, size_type n);
```

In this case we refer to each index `i` with `0 <= i < n` as a *valid index* of `a`.

ACSL's built-in predicate `\valid(a+(0.. n))` refers to all addresses `a+i` where `0 <= i <= n`. However, the array notation **`int`** `a[n]` of the C programming language refers only to the elements `a+i` where `i` satisfies `0 <= i < n`. Users of ACSL must therefore specify `\valid(a+(0.. n-1))` in order to express a valid array of length `n`.

In order to avoid this repeated writing of `n-1` we introduce the predicate `IsValidRange`.

```
/*@
  predicate IsValidRange(value_type* a, integer n) =
    (0 <= n) && \valid(a+(0.. n-1));
*/
```

Listing 1.1: The Predicate `IsValidRange`

Using `IsValidRange` allows us to shorten the specification of the function `example` to

```
/*@
    requires IsValidRange(a, n);
*/
void example(value_type* a, size_type n);
```

The predicate `IsValidRange` will be frequently referenced in this document.

2. The Hoare Calculus

In 1969, C.A.R. Hoare introduced a calculus for formal reasoning about properties of imperative programs [12], which became known as “Hoare Calculus”.

The basic notion is

```
//@ assert P;
Q;
//@ assert R;
```

where `P` and `R` denote logical expressions and `Q` denotes a source-code fragment. Informally, this means “If `P` holds before the execution of `Q`, then `R` will hold after the execution”. Usually, `P` and `R` are called “precondition” and “postcondition” of `Q`, respectively. The syntax for logical expressions is described in [9, Section 2.2] in full detail. For the purposes of this tutorial, the notions shown in Table 2.1 are sufficient. Note that they closely resemble the logical and relational operators in C.

`!P`	negation	“`P` is not true”
`P && Q`	conjunction	“`P` is true and `Q` is true”
`P || Q`	disjunction	“`P` is true or `Q` is true”
`P ==> Q`	implication	“if `P` is true, then `Q` is true”
`P <==> Q`	equivalence	“if, and only if, `P` is true, then `Q` is true”
`x < y == z`	relation chain	“`x` is less than `y` and `y` is equal to `z`”
`\forall int x; P(x)`	universal quantifier	“`P(x)` is true for every `int` value of `x`”
`\exists int x; P(x)`	existential quantifier	“`P(x)` is true for some `int` value of `x`”

Table 2.1.: Some ACSL Formula Syntax

The Listings 2.2 and 2.3 shows three example source-code fragments and annotations.

```
//@ assert x % 2 == 1;
++x;
//@ assert x % 2 == 0;
```

(a)

```
//@ assert 0 <= x <= y;
++x;
//@ assert 0 <= x <= y + 1;
```

(b)

Figure 2.2.: Example Source-Code Fragments and Annotations

```
//@ assert true;
while (--x != 0)
    sum += a[x];
//@ assert x == 0;
```

Figure 2.3.: Loop Source Code Fragments and Annotations

Their informal meanings are as follows:

Figure 2.2 (a) "If `x` has an odd value before execution of the code `++x` then `x` has an even value thereafter."

Figure 2.2 (b) "If the value of `x` is in the range $\{0, \ldots, y\}$ before execution of the same code, then `x`'s value is in the range $\{0, \ldots, y + 1\}$ after execution."

Figure 2.3 "Under any circumstances, the value of `x` is zero after execution of the loop code."

Any C programmer will confirm that these properties are valid.[6] The examples were chosen to demonstrate also the following issues:

- For a given code fragment, there does not exist one fixed pre- or postcondition. Rather, the choice of formulas depends on the actual property to be verified, which comes from the application context. The two examples in Figure 2.2 share the same code fragment, but have different pre- and postconditions.
- The postcondition need not be the most restricting possible formula that can be derived. In Figure 2.3, it is not an error that we stated only that `0 <= x` although we know that even `1 <= x`.
- In particular, pre- and postconditions need not contain all variables appearing in the code fragment. Neither `sum` nor `a[]` is referenced in the formulas of Figure 2.3.
- We can use the predicate **`true`** to denote the absence of a properly restricting precondition, as we did in Figure 2.3.
- It is not possible to express by pre- and postconditions that a given piece of code will always terminate. Figure 2.3 only states that *if* the loop terminates, then `x == 0` will hold. In fact, if `x` has a negative value on entry, the loop will run forever. However, if the loop terminates, `x == 0` will hold, and that is what Figure 2.3 claims.

  Usually, termination issues are dealt with separately from correctness issues. Termination proofs may, however, refer to properties stated (and verified) using the Hoare Calculus.

Hoare provided the rules shown in Figures 2.4 to 2.14 in order to reason about programs. We will comment on them in the following sections.

[6] We leave the important issues of overflow aside for a moment.

2.1. The Assignment Rule

We start with the rule that is probably the least intuitive of all Hoare-Calculus rules, viz. the assignment rule. It is depicted in Figure 2.4, where "`P {x↦e}`" denotes the result of substituting each occurrence of `x` in `P` by `e`.

```
//@ assert P {x |--> e};
x = e;
//@ assert P;
```

Figure 2.4.: The Assignment Rule

For example,

if	`P`	is	`x > 0 && a[2* x ] == 0` ,
then	`P {x ↦ y+1}`	is	`y+1 > 0 && a[2*(y+1)] == 0` .

Hence, we get Figure 2.5 as an example instance of the assignment rule. Note that parentheses are required in the index expression to get the correct `2*(y+1)` rather than the faulty `2*y+1`.

```
//@ assert y+1 > 0 && a[2*(y+1)] == 0;
x = y+1;
//@ assert x > 0 && a[2*x] == 0;
```

Figure 2.5.: An Assignment Rule Example Instance

Note that several different expressions `P` may result in the same expression `P {x |--> e}`. For example, all four expressions

```
          x  > 0 && a[2*  x  ] == 0  ,
          x  > 0 && a[2*(y+1)] == 0  ,
        y+1 > 0 && a[2*  x  ] == 0  ,
and     y+1 > 0 && a[2*(y+1)] == 0  ,
```

result in `y+1 > 0 && a[2*(y+1)] == 0`
after applying `{x |--> y+1}` .

For this reason, the same precondition and statement may result in several different postconditions (All four above expressions are valid postconditions in Figure 2.5, for example). However, given a postcondition and a statement, there is only one precondition that corresponds.

When first confronted with Hoare's assignment rule, most people are tempted to think of a simpler and more intuitive alternative, shown in Figure 2.6.

```
//@ assert P;
x = e;
//@assert P && x == e;
```

Figure 2.6.: Simpler, but *Faulty* Assignment Rule

Figures 2.7–2.9 show some example instances of this faulty rule.

```
//@ assert y > 0;
x = y+1;
//@ assert y > 0 && x == y+1;
```

Figure 2.7.: An Example Instance of the Faulty Rule from Figure 2.6

While Figure 2.7 happens to be ok, Figures 2.8 and 2.9 lead to postconditions that are obviously nonsensical formulas.

```
//@ assert true;
x = x+1;
//@assert x == x+1;
```

Figure 2.8.: An Example Instance of the Faulty Rule from Figure 2.6

The reason is that in the assignment in Figure 2.8 the left-hand side variable `x` also appears in the right-hand side expression `e`, while the assignment in Figure 2.9 just destroys the property from its precondition.

```
//@ assert x < 0;
x = 5;
//@ assert x < 0 && x == 5;
```

Figure 2.9.: An Example Instance of the Faulty Rule from Figure 2.6

Note that the correct example Figure 2.7 can as well be obtained as an instance of the correct rule from Figure 2.4, since replacing `x` by `y+1` in its postcondition yields `y > 0 && y+1 == y+1` as precondition, which is logically equivalent to just `y > 0`.

2.2. The Sequence Rule

The sequence rule, shown in Figure 2.10, combines two code fragments `Q` and `S` into a single one `Q ; S`. Note that the postcondition for `Q` must be identical to the precondition of `S`. This just reflects the sequential execution ("first do `Q`, then do `S`") on a formal level. Thanks to this rule, we may "annotate" a program with interspersed formulas, as it is done in Frama-C.

```
//@ assert P;
Q;
//@ assert R;
```

and

```
//@ assert R;
S;
//@ assert T;
```

$\rightsquigarrow$

```
//@ assert P;
Q ; S;
//@ assert T;
```

Figure 2.10.: The Sequence Rule

2.3. The Implication Rule

The implication rule, shown in Figure 2.11, allows us at any time to weaken a postcondition and to sharpen a precondition. We will provide application examples together with the next rule.

```
//@ assert P;
Q;
//@ assert R;
```

$\rightsquigarrow$

```
//@ assert P`;
Q;
//@ assert R`;
```

if `P` ==> P`
and `R` ==> R`

Figure 2.11.: The Implication Rule

2.4. The Choice Rule

The choice rule, depicted in Figure 2.12, is needed to verify `if(...){...} else{...}` statements. Both branches must establish the same postcondition, viz. `S`. The implication rule can be used to weaken differing postconditions `S1` of a `then` branch and `S2` of an **`else`** branch into a unified postcondition `S1||S2`, if necessary. In each branch, we may use what we know about the condition `B`, e.g. in the **`else`** branch, that it is false. If the **`else`** branch is missing, it can be considered as consisting of an empty sequence, having the postcondition `P && !B`.

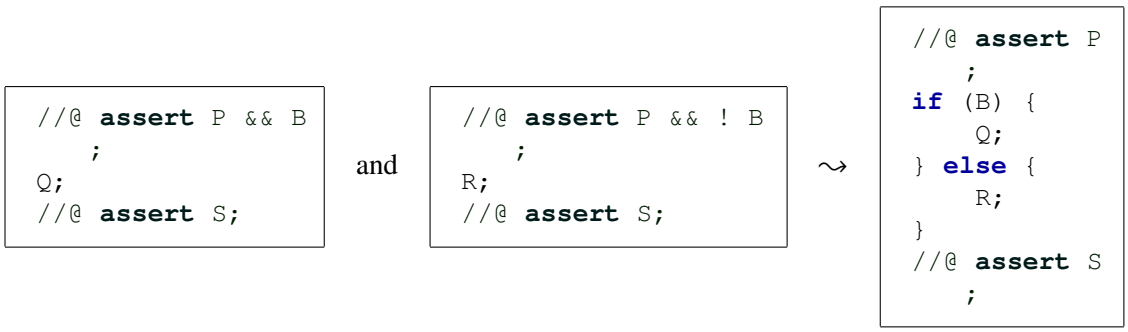


Figure 2.12.: The Choice Rule

Figure 2.13 shows an example application of the choice rule. The variable `i` may be used as an index into a ring buffer **`int`** `a[n]`. The shown code fragment just advances the index `i` appropriately. We verified that `i` remains a valid index into `a[]` provided it was valid before. Note the use of the implication rule to establish preconditions for the assignment rule as needed, and to unify the postconditions of the `then` and **`else`** branch, as required by the choice rule.

`//@assert 0 <= i < n`	given precondition
`if (i < n-1){`	
`  //@assert 0 <= i < n - 1;`	using that the condition `i < n-1;` holds in the
`  //@assert 1 <= i+1 < n;`	`then` part by the implication rule
`  i = i+1;`	
`  //@assert 1 <= i < n;`	by the assignment rule
`  //@assert 0 <= i < n;`	weakened by the implication rule
`} else {`	
`  //@assert 0 <= i == n-1 < n;`	using that then condition `i < n-1` fails in the
`  //@assert 0 == 0 && 0 < n;`	**`else`** part weakened by the implication rule
`  i = 0;`	
`  //@assert i == 0 && 0 < n;`	by the assignment rule
`  //@assert 0 <= i < n;`	weakened by the implication rule
`}`	
`//@assert 0 <= i < n;`	by the choice rule from the `then` and the **`else`** part

Figure 2.13.: An Example Application of the Choice Rule

2.5. The Loop Rule

The loop rule, shown in Figure 2.14, is used to verify a **`while`** loop. This requires to find an appropriate formula, `P`, which is preserved by each execution of the loop body. `P` is also called a loop invariant.

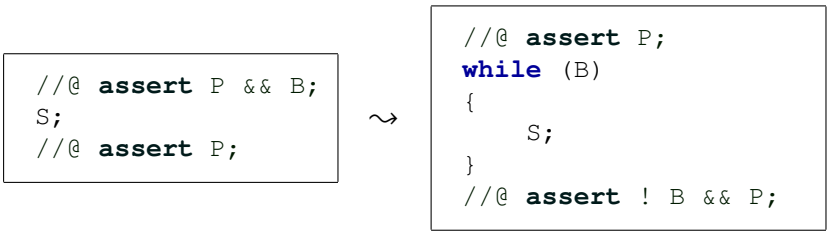


Figure 2.14.: The Loop Rule

To find it requires some intuition in many cases; for this reason, automatic theorem provers usually have problems with this task.

As said above, the loop rule does not guarantee that the loop will always eventually terminate. It merely assures us that, if the loop has terminated, the postcondition holds. To emphasis this, the properties verifiable with the Hoare Calculus are usually called “partial correctness” properties, while properties that include program termination are called “total correctness” properties.

As an example application, let us consider an abstract ring-buffer loop as shown in Figure 2.15.

```
//@assert 0 < n;                          given precondition
//@assert 0 <= 0 < n;                     follows trivially
int i = 0;
//@assert 0 <= i < n;                     by the assignment rule
while(!done) {
  //@assert 0 <= i < n && !done;          may be assumed by the loop rule
  a[i] = getchar();
  //@assert 0 <= i < n && !done;          required property of getchar
  //@assert 0 <= i < n;                   weakened by the implication rule
  if (i < n-1)i++; else i = 0;
  //@assert 0 <= i < n;                   as seen above (Figure 2.13)
  process(a, i, &done);
  //@assert 0 <= i < n;                   required property of process
}
//@assert 0 <= i < n;                     by the loop rule
```

Figure 2.15.: An Abstract Ring Buffer Loop

Figure 2.15 shows a verification proof for the index `i` lying always within the valid range `[0..n-1]` during, and after, the loop. It uses the proof from Figure 2.13 as a sub-part. Note the following issues:

- To reuse the proof from Figure 2.13, we had to drop the conjunct `!done`, since we didn't consider it in Figure 2.13. In general, we may *not* infer

  ```
  //@ assert P && S;
  Q;
  //@ assert R && S;
  ```
  from
  ```
  //@ assert P;
  Q;
  //@ assert R;
  ```

  since the code fragment `Q` may just destroy the property `S`. This is obvious for `Q` being the fragment from Figure 2.13, and `S` being e.g. `i != 0`.

  Suppose for a moment that `process` had been implemented in a way such that it refuses to set `done` to **`true`** unless it is **`false`** at entry. In this case, we would really need that `!done` still holds after execution of Figure 2.13. We would have to do the proof again, looping-through an additional conjunct `!done`.

- We have similar problems to carry the property `0 <= i < n && !done` and `0 <= i < n` over the statement `a[i] = getchar()` and `process(a, i, &done)`, respectively. We need to specify that neither `getchar` nor `process` is allowed to alter the value of `i` or `n`. In ACSL, there is a particular language construct `assigns` for that purpose, which is introduced in Section 6.2 on Page 70.

- In our example, the loop invariant can be established between any two statements of the loop body. However, this need not be the case in general. The loop rule only requires the invariant holds before the loop and at the end of the loop body. For example, `process`

could well change the value of `i`[7] and even `n` intermediately, as long as it re-establishes the property `0 <= i < n` immediately prior to returning.

- The loop invariant, `0 <= i < n`, is established by the proof in Figure 2.13 also after termination of the loop. Thus, e.g., a final `a[i] = '\0'` after the loop would be guaranteed not to lead to a bounds violation.
- Even if we would need the property `0 <= i < n` to hold only immediately before the assignment `a[i] = getchar()`, since, e.g., `process`'s body didn't use `a` or `i`, we would still have to establish `0 <= i < n` as a loop invariant by the loop rule, since there is no other way to obtain any property inside a loop body. Apart from this formal reason it is obvious that `0 <= i < n` wouldn't hold during the second loop iteration unless we re-established it at the end of the first one, and that is just what the while rule requires.

2.6. Derived Rules

The above rules don't cover all kinds of statements allowed in C. However, missing C-statements can be rewritten into a form that is semantically equivalent and covered by the Hoare rules.

For example,

```
switch (E) {
    case E1: Q1; break; ...
    case En: Qn; break;
    default: Q0; break;
}
```

is semantically equivalent to

```
if (E == E1) {
    Q1;
} else ... if (E == En) {
    Qn;
} else {
    Q0;
}
```

if `E` doesn't have side-effects. While the **`if-else`** form is usually slower in terms of execution speed on a real computer, this doesn't matter for verification purposes, which are separate from execution issues.

Similarly, **`for`** `(P; Q; R) {S}` can be re-expressed as `P;` **`while`** `(Q) {S ; R}`, and so on.

It is then possible to derive a Hoare rule for each kind of statement not previously discussed, by applying the classical rules to the corresponding re-expressed code fragment. However, we do not present these derived rules here.

Although procedures cannot be re-expressed in the above way if they are (directly or mutually) recursive, it is still possible to derive Hoare rules for them. This requires the finding of appropriate

[7] We would have to change the call to `process(a, &i, &done)` and the implementation of `process` appropriately. In this case we couldn't rely on the above-mentioned `assigns` clause for `process`.

"procedure invariants" similar to loop invariants. Non-recursive procedures can, of course, just be inlined to make the classical Hoare rules applicable.

Note that `goto` cannot be rewritten in the above way; in fact, programs containing `goto` statements cannot be verified with the Hoare Calculus. See [13] for a similar calculus that can deal with arbitrary flowcharts, and hence arbitrary jumps. In fact, Hoare's work was based on that calculus. Later calculi inspired from Hoare's work have been designed to re-integrate support for arbitrary jumps. However, in this tutorial, we will not discuss example programs containing a `goto`.

3. Non-mutating Algorithms

In this chapter, we consider *non-mutating* algorithms, i.e., algorithms that neither change their arguments nor any objects outside their scope. This requirement can be formally expressed with the following *assigns clause*:

```
assigns \nothing;
```

Each algorithm in this chapter therefore uses this assigns clause in its specification.

The specifications of these algorithms are not very complex. Nevertheless, we have tried to arrange them so that the earlier examples are simpler than the later ones. All algorithms work on one-dimensional arrays ("ranges").

`equal` (Section 3.1 on Page 24) compares two ranges element-by-element.

`mismatch` (Section 3.2 on Page 28) returns the smallest index where two ranges differ. An implementation of `equal` using `mismatch` is introduced.

`find` (Section 3.3 on Page 30) provides *sequential* or *linear search* and returns the smallest index at which a given value occurs in a range. In Section 3.4, on Page 32, a predicate is introduced in order to simplify the specification.

`find_first_of` (Section 3.5, on Page 34) provides similar to `find` a *sequential search*, but unlike `find` it does not search for a particular value, but for the least index of a given range which occurs in another range.

`adjacent_find` (Section 3.6 on Page 36) can be used to find equal neighbors in an array.

`search` (Section 3.7, on Page 38) finds a subsequence that is identical to a given sequence when compared element-by-element and returns the position of the first occurrence.

`count` (Section 3.8, on Page 42) returns the number of occurrences of a given value in a range. Here we will employ some user-defined axioms to formally specify `count`.

3.1. The `equal` Algorithm

The `equal` algorithm in the C++ Standard Library compares two generic sequences. For our purposes we have modified the generic implementation[8] to that of an array of type **`value_type`**. The signature now reads:

```
bool equal(const value_type* a, size_type n, const value_type* b);
```

The function returns `true` if `a[i] == b[i]` holds for each `0 <= i < n`. Otherwise, `equal` returns `false`.

3.1.1. Formal Specification of `equal`

The ACSL specification of `equal` is shown in Listing 3.1. We discuss the specification now line by line.

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

  assigns \nothing;

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

  behavior some_not_equal:
    assumes \exists integer i; 0 <= i < n && a[i] != b[i];
    ensures !\result;

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

Listing 3.1: Formal specification of `equal`

The first part of our specification are the preconditions, which must be existent before the algorithm is executed. Those requirements can be specified with the **`requires`**-clause in ACSL. In case of the `equal` algorithm it is needed that `n` is non-negative and that the pointers `a` and `b` point to *n* contiguously allocated objects of type **`value_type`** (see also Section 1.2).

In the second part of our specification we make a statement about objects and arguments that changed by the function. Since `equal` is a non-mutating algorithm and does not modify any memory location outside its scope we just define `assigns \nothing` (see Page 23).

Finally, we must define the postconditions, which must satisfy after the `equal` algorithm is finished. Therefore we have two behaviors:

[8] See `http://www.sgi.com/tech/stl/equal.html`.

The behavior `all_equal` applies if an element-wise comparison of the two ranges yields that they In this case the function `equal` is expected to return `true`. Finally, the behavior `some_not_equal` applies if there is at least one valid index `i` where the elements `a[i]` and `b[i]` differ. In this case the function `equal` is expected to return `false`.

The negation of the formula

```
\forall integer i; 0 <= i < n ==> a[i] == b[i];
```

in behavior `all_equal` is just the formula

```
\exists integer i; 0 <= i < n && a[i] != b[i];
```

in behavior `some_not_equal`. Therefore, these two behaviors complement each other.

The **`complete behaviors`**-clause in Listing 3.1 expresses the fact that for all ranges `a` and `b` that satisfy the preconditions of the contract *at least one* of the specified named behaviors, in this case `all_equal` and `some_not_equal`, applies.

The **`disjoint behaviors`**-clause in Listing 3.1 formalizes the fact that for all ranges `a` and `b` that satisfy the preconditions of the contract *at most one* of the specified named behaviors, in this case `all_equal` and `some_not_equal`, applies.

3.1.2. The `IsEqual` Predicate

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 behavior.

The motto *don't repeat yourself* is not just good programming practice.[9] It is also true for concise and easy to understand specifications. We will therefore introduce specification elements that we can apply to the `equal` algorithm as well as to other specifications and implementations with the described behavior.

In Listing 3.2 we introduce the predicate `IsEqual`.

```
/*@
  predicate
    IsEqual{A,B}(value_type* a, integer n, value_type* b) =
      \forall integer i; 0 <= i < n ==>
        \at(a[i], A) == \at(b[i], B);
*/
```

Listing 3.2: The Predicate `IsEqual`

This predicate formalizes the fact that the arrays `a[0],...,a[n-1]` and `b[0],...,b[n-1]` are equal when compared element by element. The letters `A` and `B` are *labels*[10] that must be supplied when using the predicate `IsEqual`. We use labels in the definition of `IsEqual` to

[9] Compare `http://en.wikipedia.org/wiki/Don't_repeat_yourself`.
[10] Labels are used in C to name the target of the *goto* jump statement.

extend its applicability. The expression `\at(a[i], A)` means that `a[i]` is evaluated at the label `A`. Frama-C defines several standard labels, e.g. `Old` and `Post`, a programmer can use to refer to the pre-state or post-state, respectively, of a function. For more details on labels we refer to the ACSL specification [9, p. 39].

Using this predicate we can reformulate the specification of `equal` in Listing 3.1 as shown in Listing 3.3. Here we use the predefined label `Here`. When used in an **`ensures`** clause the label `Here` refers to the pre-state of a function.

```
/*@
  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);
```

Listing 3.3: Formal specification of `equal` using the `IsEqual` predicate

Note that the equivalence is needed in the ensures clause. Putting an equality instead is not legal in ACSL, because `IsEqual` is a predicate.

3.1.3. Implementation of `equal`

Listing 3.4 shows one way to implement the function `equal`. In our description, we concentrate on the *loop annotations*.

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

  return true;
}
```

Listing 3.4: Implementation of `equal`

The first loop *invariant* is needed to prove that all accesses to `a` and `b` occur with valid indices. However, we may *not* require simply

```
loop invariant 0 <= i < n;
```

since the very last loop iteration would violate this formula. Therefore, we have to weaken the formula to that shown in the implementation of Listing 3.4, which is preserved by *all* iterations of the loop. Note that `0 <= i < n` is still valid immediately before the array accesses in, since we may assume there in addition that the loop condition `i < n` holds. However, `0 <= i < n` is invalid after completion of the loop, while the loop invariant is guaranteed to hold there, too, cf. the loop rule in Figure 2.14 on Page 19.

Most important is the last loop *invariant*. It complies with the postcondition of the specification in Listing 3.3 and is needed to prove that for each iteration all elements of `a` and `b` up to that iteration step are equal.

The loop assigns-clause in Listing 3.4 expresses that only the loop index is modified in any iteration. This is in accordance with the fact that `equal` is a *non-mutating* algorithm.

The loop *variant* is needed to generate correct verification conditions for the termination of the `for`-loop. In order to prove the termination of the loop, Frama-C needs to know an expression whose value is decreased by each and every loop cycle and is always positive[11][9, Subsections 2.4.2, 2.5.1]. For a `for` loop as simple as that the expression `n-i` is sufficient for that purpose.

Again, we can use the predicate `IsEqual` in order to simplify the second loop invariant, which complies our postcondition. Listing 3.5 shows the modified implementation using the predicate `IsEqual`.

```
bool equal(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 false;
  }

  return true;
}
```

Listing 3.5: Implementation of `equal` using the `IsEqual` predicate

[11] Except for possibly the very last iteration.

3.2. The `mismatch` Algorithm

The `mismatch` algorithm is closely related to the negation of `equal` from Section 3.1. Its signature reads

```
size_type mismatch(const value_type* a, int n,
                   const value_type* b);
```

The function `mismatch` returns the smallest index where the two ranges `a` and `b` differ. If no such index exists, that is, if both ranges are equal then `mismatch` returns the length `n` of the two ranges.[12]

3.2.1. Formal Specification of `mismatch`

We use the `IsEqual` predicate defined in Listing 3.2 also for the formal specification of `mismatch` that is shown in Listing 3.6.

Note in particular the use of `IsEqual` in the specification shown in Listing 3.6 in order to express that `mismatch` returns the *smallest* index where the two arrays differ. Note also that the completeness and disjointedness of the behaviors `all_equal` and `some_not_equal` has now become immediately obvious, since their **`assumes`** clauses are just literal negations of each other.

```
/*@
  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);
```

Listing 3.6: Formal specification of `mismatch`

[12] See also `http://www.sgi.com/tech/stl/mismatch.html`.

3.2.2. Implementation of `mismatch`

Listing 3.7 shows an implementation of `mismatch` that we have enriched with some loop annotations to support the deductive verification.

```
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;
}
```

Listing 3.7: Implementation of `mismatch`

We use the predicate `IsEqual` as shown in Listing 3.7 in order to express that all indices `k` that are less than the current index `i` satisfy the condition `a[k] == b[k]`. This is necessary to prove that `mismatch` indeed returns the smallest index where the two ranges differ.

3.2.3. Implementation of `equal` by calling `mismatch`

Listing 3.8 shows an implementation of the `equal` algorithm by a simple call of `mismatch`.[13]

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

Listing 3.8: Implementation of `equal` with `mismatch`

[13] See also the note on the relationship of `equal` and `mismatch` on `http://www.sgi.com/tech/stl/equal.html`.

3.3. The `find` Algorithm

The `find` algorithm in the C++ standard library implements *sequential search* for general sequences.[14] We have modified the generic implementation, which relies heavily on C++ templates, to that of a range of type **`value_type`**. The signature now reads:

```
size_type find(const value_type* a, size_type n, value_type val);
```

The function `find` returns the least *valid* index `i` of `a` where the condition `a[i] == val` holds. If no such index exists then `find` returns the length `n` of the array.

3.3.1. Formal Specification of `find`

The formal specification of `find` in ACSL is shown in Listing 3.9.

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

  assigns  \nothing;

  behavior some:
    assumes  \exists integer i; 0 <= i < n && a[i] == val;
    ensures  0 <= \result < n;
    ensures  a[\result] == val;
    ensures  \forall integer i; 0 <= i < \result ==> a[i] != val;

  behavior none:
    assumes  \forall integer i; 0 <= i < n ==> a[i] != val;
    ensures  \result == n;

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

Listing 3.9: Formal specification of `find`

The **`requires`**-clause indicates that `n` is non-negative and that the pointer `a` points to n contiguously allocated objects of type **`value_type`** (see Section 1.2).

The `assigns`-clause indicates that `find` (as a non-mutating algorithm), does not modify any memory location outside its scope (see Page 23).

We have subdivided the specification of `find` into two behaviors (named `some` and `none`). The behavior `some` applies if the sought-after value is contained in the array. We express this condition by using the **`assumes`**-clause. The next line expresses that if the assumptions of the behavior are satisfied then `find` will return a valid index. The algorithm also ensures that the returned (valid) index `i`, `a[i] == val` holds. Therefore we define this property in the second postcondition of

[14] See `http://www.sgi.com/tech/stl/find.html`.

behavior `some`. Finally, it is important to express that `find` return the smallest index `i` for which `a[i] == val` holds (see last postcondition of behavior `some`).

The behavior `none` covers the case that the sought-after value is *not* contained in the array (see **`assumes`**-clause of behavior `none` in Listing 3.9). In this case, `find` must return the length `n` of the range `a`.

Note that the formula in the **`assumes`**-clause of the behavior `some` is the negation of the **`assumes`**-clause of the behavior `none`. Therefore, we can express that these two behaviors are *complete* and *disjoint*.

3.3.2. Implementation of `find`

Listing 3.10 shows a straightforward implementation of `find`. The only noteworthy elements of this implementation are the *loop annotations*.

```
size_type find(const 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 i;
    loop   variant n-i;
   */
  for (size_type i = 0; i < n; i++)
    if (a[i] == val)
      return i;

  return n;
}
```

Listing 3.10: Implementation of `find`

The first loop *invariant* is needed to prove that accesses to `a` only occur with valid indices. With the second loop *invariant* is needed for the proof of the postconditions of the behavior `some` (see Listing 3.9). It expresses that for each iteration the sought-after value is not yet found up to that iteration step.

Finally, the loop *variant* `n-i` is needed to generate correct verification conditions for the termination of the loop.

3.4. The `find` Algorithm Reconsidered

In this section we specify the `find` algorithm in a slightly different way when compared to Section 3.3. Our approach is motivated by a considerable number of closely related formulas. We have in Listings 3.9 and 3.10 the following formulas

```
\exists integer i; 0 <= i < n        &&  a[i] == val;
\forall integer i; 0 <= i < \result ==> a[i] != val;
\forall integer i; 0 <= i < n        ==> a[i] != val;
\forall integer k; 0 <= k < i        ==> a[k] != val;
```

Note that the first formula is the negation of the third one.

In order to be more explicit about the commonalities of these formulas we define a predicate, called `HasValue` (see Listing 3.11), which describes the situation that there is a valid index `i` such that

```
a[i] == val
```

holds.

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

Listing 3.11: The Predicate `HasValue`

Note that we needed to provide a label, viz. `A`, to the predicate, since its evaluation depends on a memory state, viz. then contents of `a[]`. ACSL allows to abbreviate `\at(a[i],A)` by `a[i]` if, as in our predicate body, `A` is the only available label.

With this predicate we can encapsulate all uses of the ∀ and ∃ quantifiers in both the specification of the function contract of `find` and in the loop annotations. The result is shown in Listings 3.12 and 3.13.

3.4.1. Formal Specification of `find`

This approach leads to a specification of `find` that is more readable than the one described in Section 3.3.

In particular, it can be seen immediately that the conditions in the assumes clauses of the two behaviors `some` and `none` are mutually exclusive since one is the literal negation of the other. Moreover, the requirement that `find` returns the smallest index can also be expressed using the `HasValue` predicate, as depicted with the last postcondition of behavior `some` as shown in Listing 3.12.

```
/*@
  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);
```

Listing 3.12: Formal Specification of `find` Using the `HasValue` Predicate

3.4.2. Implementation of `find`

The predicate `HasValue` is also used in the loop annotation inside the implementation of `find`.

```
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;
}
```

Listing 3.13: Implementation of `find` with loop annotations based on `HasValue`

3.5. The `find_first_of` Algorithm

The `find_first_of` algorithm[15] is closely related to `find` (see Sections 3.3 and 3.4).

```
size_type find_first_of(const value_type* a, size_type m,
                        const value_type* b, size_type n);
```

As in `find` it performs a sequential search. However, whereas `find` searches for a particular value, `find_first_of` returns the least index `i` such that `a[i]` is equal to one of the values `b[0]`, ..., `b[n-1]`.

3.5.1. Formal Specification of `find_first_of`

Similar to our approach in Section 3.4, we define a predicate `HasValueOf` that formalizes the fact that there are valid indices `i` for `a` and `j` of `b` such that `a[i] == b[j]` hold. One way to achieve this would be to define `HasValueOf` as follows:

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

However, we have chosen to reuse the predicate `HasValue` (Listing 3.11) to define the `HasValueOf` predicate. The result is shown in Listing 3.14.

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

Listing 3.14: The Predicate `HasValueOf`

Both the predicates `HasValueOf` and `HasValue` occur in the formal specification of `find_first_of` (see Listing 3.15). Note how similar the specification of `find_first_of` becomes to that of `find` (Listing 3.12) when using these predicates.

3.5.2. Implementation of `find_first_of`

Our implementation of `find_first_of` is shown in Listing 3.16.

[15] See `http://www.sgi.com/tech/stl/find_first_of.html`.

```
/*@
  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);
```

Listing 3.15: Formal specification of `find_first_of`

```
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 \forall integer k; 0 <= k < i ==> !HasValue(b, n,
       a[k]);
    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;
}
```

Listing 3.16: Implementation of `find_first_of`

Note the call of the `find` function shown in the Listing above. In the original STL implementation[16], `find_first_of` does not call `find` but rather inlines it. The reason for this were probably efficiency considerations. We opted for an implementation of `find_first_of` that emphasizes reuse. Besides, leading to a more concise implementation, we also have to write less loop annotations.

[16] See `http://www.sgi.com/tech/stl/stl_algo.h`

3.6. The `adjacent_find` Algorithm

The `adjacent_find` algorithm[17]

```
size_type adjacent_find(const value_type* a, size_type n);
```

returns the smallest valid index i, such that i+1 is also a valid index and such that

```
a[i] == a[i+1]
```

holds. The `adjacent_find` algorithm returns `n` if no such index exists.

3.6.1. Formal Specification of `adjacent_find`

As in the case of other search algorithms, we first define a predicate `HasEqualNeighbors` (see Listing 3.17) that captures the essence of finding two adjacent indices at which the array holds equal values.

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

Listing 3.17: The Predicate `HasEqualNeighbors`

```
/*@
  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);
```

Listing 3.18: Formal specification of `adjacent_find`

[17] See `http://www.sgi.com/tech/stl/adjacent_find.html`

We use the predicate `HasEqualNeighbors` to define the formal specification of `adjacent_find` (see Listing 3.18).

3.6.2. Implementation of `adjacent_find`

The implementation of `adjacent_find`, including loop (in)variants is shown in Listing 3.19. Please note the use of the predicate `HasEqualNeighbors` in the loop invariant to match the similar postcondition of behavior `some`.

```
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;
}
```

Listing 3.19: Implementation of `adjacent_find`

3.7. The `search` Algorithm

The `search` algorithm in the C++ standard library finds a subsequence that is identical to a given sequence when compared element-by-element. For our purposes we have modified the generic implementation [18] to that of an array of type **`value_type`**. The signature now reads:

```
size_type search(const value_type* a, size_type m,
                 const value_type* b, size_type n)
```

The function `search` returns the first index `i` of the array `a` where the condition `a[i+k]==b[k]` for each `0 <= k < n` holds. If no such index exists then `search` returns the length `m` of the array `a`.

3.7.1. Formal Specification of `search`

Our specification of `search` starts with introducing the predicate `HasSubRange`. This predicate formalizes the fact that the sequence `a` contains a subsequence which is identical to the sequence `b`.

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

Listing 3.20: The Predicate `HasSubRange`

The ACSL specification of `search` is shown in Listing 3.21.

The behavior `has_match` applies if the sequence `a` contains a subsequence, which is identical to the sequence `b`. We express this condition with **`assumes`** by using the predicate `HasSubRange` as shown in Listing 3.21 which is mentioned above.

The first **`ensures`**-clause of behavior `has_match` indicates that the return value must be in the range `[0,m-n]`. The second one expresses that `search` returns the smallest index where `b` can be found in `a`. Finally, in the last line under behavior `has_match` we indicate that the sequence `a` contains a subsequence (from the position `\result`), which is identical to the sequence `b`.

Figure 3.22 explains, that if `\result` is the smallest index where `b[0,...,n)` can be found in `a`, then `b` is not a subrange of `a[0,\result+n-1)`.

The behavior `no_match` covers the case that there is no such subsequence in sequence `a`, which equals to the sequence `b`. In this case, `search` must return the length `m` of the range `a`. In any case, if the ranges `a` or `b` are empty, then the return value will be `0`. We express this fact with the following line:

```
ensures (n == 0 || m == 0) ==> \result == 0;
```

[18] See `http://www.sgi.com/tech/stl/search.html`.

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

  assigns \nothing;

  ensures (n == 0 || m == 0) ==> \result == 0;

  behavior has_match:
    assumes HasSubRange(a, m, b, n);
    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);
```

Listing 3.21: Formal specification of `search`

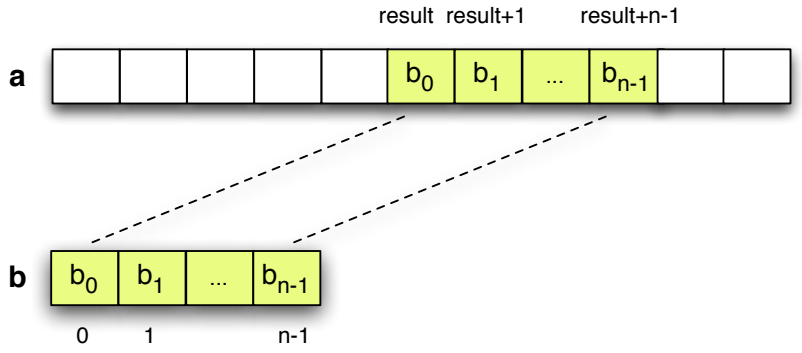


Figure 3.22.: Explanation for Line 13

The formula in the assumes clause of the behavior `has_match` is the negation of the assumes clause of the behavior `no_match`. Therefore, we can express that these two behaviors are *complete* and *disjoint*.

3.7.2. Implementation of `search`

Our implementation of `search` is shown in Listing 3.23. It follows the C++ standard library implementation in being easy to understand, but needing an order of magnitude of `m*n` operations. In contrast, the sophisticated algorithm from [14] needs only `m+n` operations.

```
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;
}
```

Listing 3.23: Implementation of `search`

The second loop *invariant* is needed for the proof of the postconditions of the behavior `has_match` (see Listing 3.21). It expresses that for each iteration the subsequence, which equals to the sequence `b`, is not yet found up to that iteration step.

3.8. The `count` Algorithm

The `count` algorithm in the C++ standard library counts the frequency of occurrences for a particular element in a sequence. For our purposes we have modified the generic implementation[19] to that of arrays of type **`value_type`**. The signature now reads:

```
size_type count(const value_type* a, size_type n, value_type val);
```

Informally, the function returns the number of occurrences of `val` in the array `a`.

3.8.1. Formal Specification of `count`

The specification of `count` will be fairly short because it employs the *logic function* `Count` whose considerably longer definition is given in Listing 3.24. This definition of `Count` is a generalization of the *logic function* `nb_occ` of the ACSL specification [9, p.50].

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

    axiom Count0:
      \forall value_type *a, v, integer i, j;
        i >= j ==> Count(a, v, i, j) == 0;

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

    axiom Count2_1:
      \forall value_type *a, v, integer i;
        (a[i] != v ==> Count(a, v, i, i+1) == 0);

    axiom Count2_2:
      \forall value_type *a, v, integer i;
        (a[i] == v ==> Count(a, v, i, i+1) == 1);
  }
*/
```

Listing 3.24: Axiomatic Definition of `Count`

The ACSL keyword **`axiomatic`** is used to gather the logic function `Count` and its defining axioms. The logic function `Count` determines the number of occurrences of a value `x` in the index range `[i,...,j-1]` of an array of type **`value_type`**. Note that the bounding indices `i`

[19] See `http://www.sgi.com/tech/stl/count.html`.

and `j` and the return value for `Count` are of type **`integer`**. The **`reads`** clause specifies the set of memory locations which `Count` depends on[20] (see [9, §2.6.10] for more details).

The axioms `Count0` and `Count2` cover the base cases, i.e., when the considered index range is empty or of length 1. Axiom `Count1`, on the other hand, deals with the composition of adjacent ranges `[i,...,j-1]` and `[j,...,k-1]`. The Lemma `CountLemma` is a consequence of axiom `Count1`.

Listing 3.25 shows our approach for a formalization of `count` in ACSL.

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

  assigns \nothing;

  ensures \result == Count(a, val, 0, n);
*/
size_type count(const value_type* a, size_type n, value_type val);
```

Listing 3.25: Formal Specification of `count`

3.8.2. Implementation of `count`

Listing 3.26 shows a possible implementation of `count` and the corresponding loop (in)variants. Note the reference to `Count` in the last loop invariant. The second loop invariant is necessary to prove that the local variable `cnt` does not exceed the range of **`size_type`**.

```
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;
}
```

Listing 3.26: Implementation of `count`

[20] WP does not support arbitrary reads definitions like our suggested `reads a[i..j-1]`. This issue will be solved in a future WP release. However, reformulating the reads clause to `reads a[i], a[j-1]` is still sufficient using the memory model Store.

4. Maximum and Minimum Algorithms

In this chapter we discuss the formal specification of algorithms that compute the maximum or minimum values of their arguments. As the algorithms in Chapter 3, they also do not modify any memory locations outside their scope. The most important new feature of the algorithms in this chapter is that they compare values using binary operators such as <.

We consider in this chapter the following algorithms.

`max_element` (Section 4.2, on Page 48) returns an index to a maximum element in range. Similar to `find` it also returns the smallest of all possible indices. An alternative specification which relies on user-defined predicates will be introduced in Section 4.3, on Page 50).

`max_seq` (Section 4.4, on Page 52) is very similar to `max_element` and will serve as an example of *modular verification*. It returns the maximum value itself rather than an index to it.

`min_element` which can be used to find the smallest element in an array (Section 4.5).

First, however, we discuss in Section 4.1 general properties that must be satisfied by the relational operators.

4.1. A Note on Relational Operators

Note that in order to compare values, STL algorithms usually rely solely on the *less than* operator $<$ or special function objects.[21] To be precises, the operator $<$ must be a *partial order*,[22] which means that the following rules hold.

irreflexivity	$\forall x : \neg(x < x)$
asymmetry	$\forall x, y : x < y \implies \neg(y < x)$
transitivity	$\forall x, y, z : x < y \wedge y < z \implies x < z$

If you wish check that the operator $<$ of our **value_type**[23] satisfies this properties you can formulate lemmas in ACSL and verify them with Frama-C. (see Listing 4.1).

```
/*@
  lemma LessIrreflexivity:
    \forall value_type a; !(a < a);

  lemma LessAntisymmetry:
    \forall value_type a, b; (a < b) ==> !(b < a);

  lemma LessTransitivity:
    \forall value_type a, b, c; (a < b) && (b < c) ==> (a < c);
*/
```

Listing 4.1: Requirements for a partial order on **value_type**

[21] See http://www.sgi.com/tech/stl/LessThanComparable.html.
[22] See http://en.wikipedia.org/wiki/Partially_ordered_set
[23] See Section 1.2

It is of course possible to specify and implement the algorithms of this chapter by only using operator <. For example, `a <= b` can be written as `a < b || a == b`, or, for our particular ordering on **`value_type`**, as `!(b < a)`. However, for the purpose of this introductory document we have opted for a more user friendly representation. Listing 4.2 formulates condition on the semantics of the derived operator >, <= and >=.

```
/*@
  lemma Greater:
    \forall value_type a, b; (a > b) <==> (b < a);

  lemma LessOrEqual:
    \forall value_type a, b; (a <= b) <==> !(b < a);

  lemma GreaterOrEqual:
    \forall value_type a, b; (a >= b) <==> !(a < b);
*/
```

Listing 4.2: Semantics of Derived Comparison Operators

4.2. The `max_element` Algorithm

The `max_element` algorithm in the C++ Standard Template Library[24] searches the maximum of a general sequence. The signature of our version of `max_element` reads:

```
size_type max_element(const value_type* a, size_type n);
```

The function finds the largest element in the range `a[0, n)`. More precisely, it returns the unique valid index `i` such that

1. for each index `k` with `0 <= k < n` the condition `a[k] <= a[i]` holds and
2. for each index `k` with `0 <= k < i` the condition `a[k] < a[i]` holds.

The return value of `max_element` is `n` if and only if there is no maximum, which can only occur if `n == 0`.

4.2.1. Formal Specification of `max_element`

A formal specification of `max_element` in ACSL is shown in Listing 4.3.

```
/*@
  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);
```

Listing 4.3: Formal Specification of `max_element`

We have subdivided the specification of `max_element` into two behaviors (named `empty` and `not_empty`). The behavior `empty` contains the specification for the case that the range contains no elements. The behavior `not_empty` applies if the range has a positive length.

[24] See `http://www.sgi.com/tech/stl/max_element.html`

The second **ensures** clause of behavior `not_empty` indicates that the returned valid index `k` refers to a maximum value of the array. The third one expresses that `k` is indeed the *first* occurrence of a maximum value in the array.

Given the requirement `0 <= n` which is a part of `IsValidRange(a, n)`, the formula in the assumes clause of the behavior `empty` is the negation of the assumes clause of the behavior `not_empty`. Therefore, we can express that these two behaviors are *complete* and *disjoint*.

4.2.2. Implementation of `max_element`

Listing 4.4 shows an implementation of `max_element`. In our description, we concentrate on the *loop annotations*.

```
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 = 1; i < n; i++)
    if (a[max] < a[i])
      max = i;

  return max;
}
```

Listing 4.4: Implementation of `max_element`

The second loop *invariant* is needed to prove the first postcondition of behavior `not_empty` in Listing 4.3, and in order to prove that all pointer accesses occur with valid indices. Using the next loop *invariant* we prove the second postcondition of behavior `not_empty` in Listing 4.3. Finally, the last postcondition of this behavior can be proved with the endmost loop *invariant*.

4.3. The `max_element` Algorithm with Predicates

In this section we present another specification of the `max_element` algorithm. The main difference is that we employ two user defined predicate `IsMaximum` and `IsFirstMaximum`.

```
/*@
  predicate IsMaximum{L}(value_type* a, integer n, integer max) =
     \forall 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];
*/
```

Listing 4.5: The `IsMaximum` and `IsFirstMaximum` Predicates

The new formal specification of `max_element` in ACSL is shown in Listing 4.6.

```
/*@
  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);
```

Listing 4.6: Formal Specification of `max_element`

Listing 4.7 shows implementation of `max_element` with rewritten loop invariants.

```
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 max, i;
    loop   variant n-i;
  */
  for (size_type i = 0; i < n; i++)
    if (a[max] < a[i])
      max = i;

  return max;
}
```

Listing 4.7: Implementation of `max_element`

4.4. The `max_seq` Algorithm

In this section we consider the function `max_seq` (see Chapter 3, [8]) that is very similar to the `max_element` function of Section 4.2. The main difference between `max_seq` and `max_element` is that `max_seq` returns the maximum value (not just the index of it). Therefore, it requires a *non-empty* range as an argument.

Of course, `max_seq` can easily be implemented using `max_element` (see Listing 4.9). Moreover, using only the formal specification of `max_element` in Listing 4.3 we are also able to deductively verify the correctness of this implementation. Thus, we have a simple example of *modular verification* in the following sense:

> Any implementation of `max_element` that is separately proven to implement the contract in Listing 4.3 makes `max_seq` behave correctly. Once the contracts have been defined, the function `max_element` could be implemented in parallel, or just after `max_seq`, without affecting the verification of `max_seq`.

4.4.1. Formal Specification of `max_seq`

A formal specification of `max_seq` in ACSL is shown in Listing 4.8.

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

  assigns  \nothing;

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

Listing 4.8: Formal Specification of `max_seq`

Using the first **`requires`**-clause we express that `max_seq` needs a *non-empty* range as input. Additionally, the second **`requires`** is needed to state that the block pointed to by `p` must contain at least `n` elements. In other words, `p[0]`, `p[1]`, ..,`p[n-1]` must all be valid memory accesses. Also possible would be the already-known notation `IsValidRange(p, n)`.

By using the **`ensures`**-clause we express our postconditions. They formalize that `max_seq` indeed returns the maximum value of the range.

4.4.2. Implementation of `max_seq`

Listing 4.9 shows the trivial implementation of `max_seq` using `max_element`. Since `max_seq` requires a non-empty range the call of `max_element` returns an index to a maximum value in the range. The fact that `max_element` returns the smallest index is of no importance in this context.

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

Listing 4.9: Implementation of `max_seq`

4.5. The `min_element` Algorithm

The `min_element` algorithm in the C++ standard library[25] searches the minimum in a general sequence. The signature of our version of `min_element` reads:

```
size_type min_element(const value_type* a, size_type n);
```

The function `min_element` finds the smallest element in the range `a[0,n)`. More precisely, it returns the unique valid index `i` such that

1. for each index `k` with `0 <= k < n` the condition `a[k] >= a[i]` holds and
2. for each index `k` with `0 <= k < i` the condition `a[k] > a[i]` holds.

The return value of `min_element` is `n` if and only if `n == 0`.

4.5.1. Formal Specification of `min_element`

The ACSL specification of `min_element` is shown in Listing 4.10. Given the great similarity to the `max_element` specification (see Section 4.2) we do not provide a detailed explanation here.

```
/*@
  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);
```

Listing 4.10: Formal Specification of `min_element`

[25] See `http://www.sgi.com/tech/stl/min_element.html`.

4.5.2. Implementation of `min_element`

Listing 4.11 shows the implementation of `min_element`. Given the great similarity with respect to the `max_element` implementation (see Section 4.2) we do not provide a detailed explanation here.

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

  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 min, i;
    loop   variant n-i;
  */
  for (size_type i = 0; i < n; i++)
    if (a[i] < a[min])
      min = i;

  return min;
}
```

Listing 4.11: Implementation of `min_element`

5. Binary Search Algorithms

In this chapter, we consider the four *binary search* algorithms of the STL, namely

- `lower_bound` in Section 5.1
- `upper_bound` in Section 5.2
- `equal_range` in Section 5.3
- `binary_search` in Section 5.4.

All binary search algorithms require that their input array is sorted in ascending order. The predicate `IsSorted` in Listing 5.1 formalizes these requirements.

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

Listing 5.1: The Predicate `IsSorted`

As in the case of the of maximum/minimum algorithms from Chapter 4 the binary search algorithms only use the less-than operator < (and the derived operators <=, > and >=) to determine whether a particular value is contained in a sorted range. Thus, different to the `find` algorithm in Section 3.3, the equality operator == does not occur directly in the specification of binary search.

5.1. The `lower_bound` Algorithm

The `lower_bound` algorithm is one of the four binary search algorithms of the STL. For our purposes we have modified the generic implementation[26] to that of an array of type **`value_type`**. The signature now reads:

```
size_type lower_bound(const value_type* a, size_type n,
                      value_type val);
```

As with the other binary search algorithms `lower_bound` requires that its input array is sorted in ascending order. Specifically, `lower_bound` will return the *largest* index `i` with `0 <= i <= n` such that for each index `k` with `0 <= k < i` the condition `a[k] < val` holds. This specification makes `lower_bound` a bit tricky to use as a search algorithm:

- If `lower_bound` returns `n` then for each index `0 <= i < n` holds `a[i] < val`. Thus, `val` is not contained in `a`.
- If, however, `lower_bound` returns an index `0 <= i < n` then we can only be sure that `val <= a[i]` holds.

5.1.1. Formal Specification of `lower_bound`

The ACSL specification of `lower_bound` is shown in Listing 5.2.

```
/*@
  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);
```

Listing 5.2: Formal specification of `lower_bound`

- The preconditions express, by using the predicate `IsSorted`, that the values in the (valid) array need to be sorted in ascending order.
- The postconditions formalize the central properties, mentioned above, of the return value of `lower_bound`.

[26] See `http://www.sgi.com/tech/stl/lower_bound.html`.

5.1.2. Implementation of `lower_bound`

Our implementation of `lower_bound` is shown in Listing 5.3. Each iteration step narrows down the range that contains the sought-after result. The loop invariants express that in each iteration step all indices less than the temporary left bound `left` contain values smaller than `val` and all indices not less than the temporary right bound `right` contain values not smaller than `val`.

```
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 assigns middle, left, right;

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

    if (a[middle] < val)
    {
      //@ assert \forall integer i; 0 <= i < middle+1 ==> a[i] < val;
      left = middle + 1;
    }
    else
      right = middle;
  }

  return left;
}
```

Listing 5.3: Implementation of `lower_bound`

5.2. The upper_bound Algorithm

The upper_bound[27] algorithm is a version of the binary_search algorithm closely related to lower_bound of Section 5.1.

The signature reads:

```
size_type upper_bound(const value_type* a, size_type n,
                      value_type val)
```

In contrast to the lower_bound algorithm the upper_bound algorithm locates the *largest* index i with 0 <= i <= n such that for each index k with 0 <= k < i the condition a[k] <= val holds. That means:

- If upper_bound returns an index 0 <= i < n then we can be sure that val < a[i] holds. Thus, if upper_bound returns 0 then we know that val is not contained in a.
- If upper_bound returns n then we can only be sure that for each index 0 <= i < n holds val <= a[i].

5.2.1. Formal Specification of upper_bound

The ACSL specification of upper_bound is shown in Listing 5.4.

```
/*@
  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);
```

Listing 5.4: Formal specification of upper_bound

The specification is quite similar to the specification of lower_bound (see Listing 5.2). The difference can be seen in the postconditions. As we are searching for the upper bound this time, upper_bound has to ensures that

- all indices less than the returned one contain values not greater than val
- all indices not less than the returned one contain values greater than val.

[27] See http://www.sgi.com/tech/stl/upper_bound.html.

5.2.2. Implementation of `upper_bound`

Our implementation of `upper_bound` is shown in Listing 5.5.

The loop invariants express that for each iteration step all indices less than the temporary left bound `left` contain values not greater than `val` and all indices not less than the temporary right bound `right` contain values greater than `val`.

```
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 assigns middle, left, right;

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

    if (a[middle] <= val)
    {
      //@ assert \forall integer i; 0 <= i < middle+1 ==> a[i] <= val;
      left = middle + 1;
    }
    else
      right = middle;
  }

  return right;
}
```

Listing 5.5: Implementation of `upper_bound`

5.3. The `equal_range` Algorithm

The `equal_range` algorithm is one of the four binary search algorithms of the STL. For our purposes we have modified the generic implementation[28] to that of an array of type `value_type`. Moreover, instead of a pair of iterators, our version of `equal_range` returns a pair of indices. To be more precisely, the return type of `equal_range` is

```
struct size_type_pair
{
  size_type first;
  size_type second;
};
```

Thus, the signature of `equal_range` now reads:

```
size_type_pair equal_range(const value_type* a, size_type n,
                           value_type val);
```

As with the other binary search algorithms `equal_range` requires that its input array is sorted in ascending order.

Informally, the specification of `equal_range` states that it *combines* the results of the algorithms `lower_bound` (Section 5.1) and `upper_bound` (Section 5.2). Thus, if `pair` is the return value of

```
equal_range(a, n, val)
```

then `pair.first` shall equal

```
lower_bound(a, n, val)
```

whereas `pair.second` shall equal

```
upper_bound(a, n, val)
```

5.3.1. Formal Specification of `equal_range`

The ACSL specification of `equal_range` is shown in Listing 5.6.

The preconditions express that the values in the (valid) array need to be sorted in ascending order.

The postconditions express that the pair of indices (f, s) returned by `equal_range` satisfy the following properties:

- $0 \leq f \leq s \leq n$
- the set of indices $[f, s) = \{i \mid f \leq i < s\}$ is the *largest* set for which $a[i] = \text{val}$ holds

[28] See `http://www.sgi.com/tech/stl/equal_range.html`.

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

  assigns \nothing;

  ensures 0 <= \result.first <= \result.second <= n;

  ensures \forall integer i; 0 <= i < \result.first  ==> a[i] <  val;
  ensures \forall integer i;
              \result.first  <= i < \result.second  ==> a[i] == val;
  ensures \forall integer i; \result.second <= i < n ==> a[i]  > val;
 */
size_type_pair equal_range(const value_type* a, size_type   n,
                           value_type val);
```

Listing 5.6: Formal specification of `equal_range`

5.3.2. Implementation of `equal_range`

Our implementation of `equal_range` is shown in Listing 5.7.

```
size_type_pair equal_range(const value_type *a, size_type n,
                           value_type val)
{
  size_type first  = lower_bound(a, n, val);
  size_type second = upper_bound(a, n, val);
  //@ assert first <= second;

  size_type_pair pair = { first, second };
  return pair;
}
```

Listing 5.7: Implementation of `equal_range`

We call the two functions `lower_bound` and `upper_bound` and return their respective results as a pair. However, instead of doing this straightforward, we use the temporary indices `first` and `second` and formulate the assertion `first` $\leq$ `second`. Using this assertion enables us to prove the first postcondition in Listing 5.6.

That this assertion is correct can be shown with an indirect proof. If indeed `first` were larger then `second` then the postcondition of `lower_bound` (Listing 5.2) would ensure that `a[second]` < `val` holds, whereas `upper_bound` (Listing 5.4) would guarantee that `a[second]` > `val` holds. This contradiction leads to `first` $\leq$ `second`.

Since the prover Alt-Ergo could not find a proof for the assertion, we have conducted the indirect proof with the interactive theorem prover Coq.

5.4. The `binary_search` Algorithm

The `binary_search` algorithm is one of the four binary search algorithms of the STL. For our purposes we have modified the generic implementation[29] to that of an array of type **`value_type`**. The signature now reads:

```
bool binary_search(const value_type* a, size_type n,
                   value_type  val);
```

Again, `binary_search` requires that its input array is sorted in ascending order. It will return `true` if there exists an index `i` in `a` such that `a[i] == val` holds.[30]

5.4.1. Formal Specification of `binary_search`

The ACSL specification of `binary_search` is shown in Listing 5.8.

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

  assigns \nothing;

  ensures \result <==> \exists integer i; 0 <= i < n && a[i] == val;
 */
bool binary_search(const value_type* a, size_type n, value_type val);
```

Listing 5.8: Formal specification of `binary_search`

Note that we can use our previously introduced predicate `HasValue` (see Page 32) in Listing 5.9. It is interesting to compare this specification with that of `find` in Listing 3.12. Both `find` and `binary_search` allow to determine whether a value is contained in an array. The fact that the C++ standard library requires that `find` has *linear* complexity whereas `binary_search` must have a *logarithmic* complexity can currently not be expressed with ACSL.

```
/*@
  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);
```

Listing 5.9: Formal specification of `binary_search` using the `HasValue` predicate

[29] See `http://www.sgi.com/tech/stl/binary_search.html`.

[30] To be more precise: The C++ standard library requires that `(a[i] <= val) && (val <= a[i])` holds. For our definition of **`value_type`** (see Section 1.2) this means that `val` equals `a[i]`.

5.4.2. Implementation of `binary_search`

Our implementation of `binary_search` is shown in Listing 5.10.

```
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;
}
```

Listing 5.10: Implementation of `binary_search`

`binary_search` first calls the `lower_bound` function from Section 5.1. Remember, if `lower_bound` returns an index `0 <= lwb < n` then we can be sure that `val <= a[lwb]` holds.

6. Mutating Algorithms

Let us now turn our attention to another class of algorithms, viz. *mutating* algorithms, i.e., algorithms that change one or more ranges. In Frama-C, you can explicitly specify that, e.g., entries in an array `a` may be modified by a function `f`, by including the following *assigns clause* into the `f`'s specification:

```
assigns a[0..length-1];
```

The expression `length-1` refers to the value of `length` when `f` is entered, see [9, Subsection 2.3.2]. Below are the seven example algorithms we will discuss next.

`fill` (Section 6.1 on Page 68) initializes each element of an array by a given fixed value.

`swap` (Section 6.2 on Page 70) exchanges two values.

`swap_ranges` (Section 6.3 on Page 72) exchanges the contents of the arrays of equal length, element by element. We use this example to present "modular verification", as `swap_ranges` reuses the verified properties of `swap`.

`copy` (Section 6.4 on Page 76) copies a source array to a destination array.

`reverse_copy` and `reverse` (Sections 6.5 and 6.6 on Pages 78 and 80, respectively) reverse an array. Whereas `reverse_copy` copies the result to a separate destination array, the `reverse` algorithm works in place.

`rotate_copy` (Section 6.7 on Page 82) rotates a source array by `m` positions and copies the results to a destination array.

`replace_copy` (Section 6.8 on Page 84) copies a source array to a destination array, but substitutes each occurrence of a given `old` value by a given `new` value.

`remove_copy` (Section 6.9 on Page 86) copies a source array to a destination array, but omits each occurrence of a value.

`unique_copy` (Section 6.10 on Page 90) copies a source array to a destination array, but in a consecutive group of duplicate elements only the first one is copied.

`iota` (Section 6.11 on Page 94) writes consecutive integers into an array. Here we have do deal with possible integer overflows when `iota` is executed.

6.1. The `fill` Algorithm

The `fill` algorithm in the C++ Standard Library initializes general sequences with a particular value. For our purposes we have modified the generic implementation[31] to that of an array of type `value_type`. The signature now reads:

```
void fill(value_type* a, size_type n, value_type val);
```

6.1.1. Formal Specification of `fill`

Listing 6.1 shows the formal specification of `fill` in ACSL.

```
/*@
  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);
```

Listing 6.1: Formal Specification of `fill`

The algorithm `iota` requires as a precondition that the array `a` is a valid range. Furthermore, in the `assigns` clause we expresses that `fill` (as a mutating algorithm), may only modify the corresponding elements of `a`. Finally, in the following postcondition we state that for each element of `a` holds `a[i] == val`.

[31] See `http://www.sgi.com/tech/stl/fill.html`.

6.1.2. Implementation of `fill`

Listing 6.2 shows an implementation of `fill`. The only noteworthy elements of this implementation are the *loop annotations*.

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

    loop assigns i, a[0..n-1];
    loop variant n-i;
  */
  while(i < n)
  {
    a[i] = val;
    ++i;
    //@ assert 0 < i <= n;
  }
}
```

Listing 6.2: Implementation of `fill`

The first loop invariant is needed to prove that each access to the range `a` occurs with valid indices. The second one expresses that for each iteration the array is filled with the value of `val` up to the index `i` of the iteration. This loop invariant corresponds to the postcondition in Listing 6.1.

In order to prove the `assigns`-clause of the specification in Listing 6.1 the loop `assigns`-clause is needed. This clause indicates that loop only modifies the index variable `i` and range `a` between the indices `0` and `n-1`. Finally, to demonstrate that the loop terminates we need the loop **`variant`** `n-i` and the assertion within the loop body.

6.2. The `swap` Algorithm

The `swap` algorithm[32] in the C++ STL exchanges the contents of two variables. Similarly, the `iter_swap` algorithm[33] exchanges the contents referenced by two pointers. Since C and hence ACSL, does not support an `&` type constructor ("declarator"), we will present an algorithm that processes pointers and refer to it as `swap`.

Our version of the original signature now reads:

```
void swap(value_type* p, value_type* q);
```

6.2.1. Formal Specification of `swap`

The ACSL specification for the `swap` function is shown in Listing 6.3.

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

  assigns *p;
  assigns *q;

  ensures \valid(p);
  ensures \valid(q);

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

Listing 6.3: Formal Specification of `swap`

The preconditions which formalize by the **`requires`**-clause states that both argument pointers to the `swap` function must be dereferenceable.

The `assigns`-clauses formalize that the `swap` algorithm modifies only the entries referenced by the pointers `p` and `q`. Nothing else may be altered. Equivalently, we could have used a clause

```
assigns *p, *q;
```

instead. In general, when more than one *assigns clause* appears in a function's specification, it permitted to modify any of the referenced locations. However, if no *assigns clause* appears at all, the function is free to modify any memory location, see [9, Subsection 2.3.2]. To forbid a function to do any modifications outside its scope, a clause

```
assigns \nothing;
```

[32] See `http://www.sgi.com/tech/stl/swap.html`.
[33] See `http://www.sgi.com/tech/stl/iter_swap.html`.

must be used, as we practised in the example specifications in Chapter 3.

Upon termination of `swap` the entries must be mutually exchanged. We can express those post-conditions by using the **`ensures`**-clause. The expression `\old(*p)` refers to the pre-state of the function contract, whereas by default, a postcondition refers the values after the functions has been terminated.

6.2.2. Implementation of `swap`

Listing 6.4 shows the usual straight-forward implementation of `swap`. No interspersed ACSL is needed to get it verified by Frama-C.

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

Listing 6.4: Implementation of `swap`

6.3. The `swap_ranges` Algorithm

The `swap_ranges` algorithm[34] in the C++ STL exchanges the contents of two expressed ranges element-wise. After translating C++ reference types and iterators to C, our version of the original signature reads:

```
void swap_ranges(value_type* a, size_type n, value_type* b);
```

We do not return a value since it would equal `n`, anyway.

This function refers to the previously discussed algorithm `swap`. Thus, `swap_ranges` serves as another example for "modular verification". The specification of `swap` will be automatically integrated into the proof of `swap_ranges`.

6.3.1. Formal Specification of `swap_ranges`

Listing 6.5 shows an ACSL specification for the `swap_ranges` algorithm.

```
/*@
  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);
```

Listing 6.5: Formal Specification of `swap_ranges`

The `swap_ranges` algorithm works correctly only if `a` and `b` do not overlap. For example,

```
value_type buf[3] = { 10, 11, 12 };
value_type* const a = &buf[0];
value_type* const b = &buf[1];
swap_ranges(a, 2, b);
printf("a = { %d, %d }\n",a[0],a[1]);
printf("b = { %d, %d }\n",b[0],b[1]);
```

results in the output

```
a = { 11, 12 }
b = { 12, 10 }
```

where `b` is different from the old `a`. Because of that fact we use the `separated`-clause to tell Frama-C that `a` and `b` must not overlap.

[34] See `http://www.sgi.com/tech/stl/swap_ranges.html`.

With the `assigns`-clause we postulate that the `swap_ranges` algorithm alters the elements contained in two distinct ranges, modifying the corresponding elements and nothing else.

The postconditions of `swap_ranges` specify that the content of each element in its post-state must equal the pre-state of its counterpart. We can use the predicate `IsEqual` (see Listing 3.2) together with the label `Old` and `Here` to express the postcondition of `swap_ranges`. In our specification in Listing 6.5, for example, we specify that the array `a` in the memory state that corresponds to the label `Here` is equal to the array `b` at the label `Old`. Since we are specifying a postcondition `Here` refers to the post-state of `swap_ranges` whereas `Old` refers to the pre-state.

6.3.2. Implementation of `swap_ranges`

Listing 6.6 shows an implementation of `swap_ranges` together with the necessary loop annotations.

```
void swap_ranges(value_type* a, size_type n, value_type* b)
{
    size_type i = 0;
  /*@
    loop invariant 0 <= i <= n;
    loop invariant IsEqual{Pre,Here}(a, i, b);
    loop invariant IsEqual{Here,Pre}(a, i, b);
    loop assigns a[0..n-1];
    loop assigns b[0..n-1], i;
    loop   variant n-i;
  */
  while (i < n)
  {
    swap(&a[i], &b[i]);
    ++i;
    //@ assert 0 < i <= n;
  }
}
```

Listing 6.6: Implementation of `swap_ranges`

For the postcondition of the specification in Listing 6.5 to hold, our loop invariants must ensure that at each iteration all of the corresponding elements that have been visited are swapped. Note that the loop invariant has to hold after incrementing `i` and before the loop test `i < n`. Figure 6.7 shows an example run for `n = 2`, `a=[10,11]`, and `b=[20,21]`. The loop invariant is checked for the memory states highlighted in green, and is valid there.

Again, we can use the predicate `IsEqual` to express the fact that in iteration `i` the elements `a[0],...,a[i-1]` are equal to the corresponding elements `b[0],...,b[i-1]` *before* the loop was entered. This is expressed in second loop invariant by binding `a` and `b` to the labels `Here` and `Pre`, respectively. The last loop invariant expresses the converse property.

The `assigns` clause declares that nothing except the ranges `a[0..n-1]` and `b[0..n-1]` are modified [9, Subsection 2.4.2]. which would be sufficient only for the current loop cycle. Finally, we prove the total correctness with the loop variant `n-i` and cover it with an additional assertion.

	n	i	a		b	
			0	1	0	1
	2		**10**	**11**	**20**	**21**
i = 0	2	**0**	10	11	20	21
i < n	2	0	10	11	20	21
swap	2	0	**20**	11	**10**	21
i++	2	**1**	20	11	10	21
i < n	2	1	20	11	10	21
swap	2	1	20	**21**	10	**11**
i++	2	**2**	20	21	10	11
i < n	2	2	20	21	10	11

Figure 6.7.: Loop Invariant Check Points

6.4. The copy Algorithm

The copy algorithm in the C++ Standard Library implements a duplication algorithm for general sequences. For our purposes we have modified the generic implementation[35] to that of a range of type value_type. The signature now reads:

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

Informally, the function copies every element from the source range a to the destination range b.

The verification of copy is very similar to that of swap_ranges, cf. Section 6.3.

6.4.1. Formal Specification of copy

The ACSL specification of copy is shown in Listing 6.8.

```
/*@
  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, const size_type n, value_type* b);
```

Listing 6.8: Formal Specification of copy

The copy algorithm expects that the ranges a and b are valid. Also important is that the ranges do not overlap, this property is expressed with the separated-clause in our specification.

Furthermore the function copy assigns the elements from the source range a to the destination range b, modifying the memory of the elements pointed to by b. Nothing else must be altered. To state this we use the assigns-clause.

In the final part of the specification we postulate the postcondition of copy, which ensures that the contents of a were actually copied to b. Again, we can use the IsEqual predicate from Section 3.1 to express that the array a equals b after copy has been called.

[35] See http://www.sgi.com/tech/stl/copy.html.

6.4.2. Implementation of `copy`

Listing 6.9 shows an implementation of the `copy` function.

```
void copy(const value_type* a, size_type n, value_type* b)
{
  size_type i = 0;
  /*@
    loop invariant 0 <= i <= n;
    loop invariant IsEqual{Here,Here}(a, i, b);
    loop assigns   b[0..n-1], i;
    loop   variant n-i;
  */
  while ( i < n)
  {
    b[i] = a[i];
    ++i;
    //@ assert 0 < i <= n;
  }
}
```

Listing 6.9: Implementation of `copy`

Here are some remarks on its loop invariants.

For the postcondition to be true, we must ensure that for every element `i`, the comparison `a[i] == b[i]` is `true`. This can be expressed by using the `IsEqual` predicate.

The `assigns` clause ensures that nothing but the range `b[0..i-1]` and the loop variable `i` is modified. In order to prove the termination of the loop for every possible `n` we use the loop variant `n-i` and cover it with an assertion.

6.5. The `reverse_copy` Algorithm

The `reverse_copy`[36] algorithm of the C++ Standard Library invert the order of elements in a sequence. `reverse_copy` does not change the input sequence and copies its result to the output sequence.

For our purposes we have modified the generic implementations to that of a range of type **`value_type`**. The signature now reads:

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

6.5.1. Formal Specification of `reverse_copy`

Informally, `reverse_copy` copies the elements from the array `a` into array `b` such that the copy is a reverse of the original array. Thus, after calling `reverse_copy` the following conditions shall be satisfied.

```
b[0]   ==  a[n-1]
b[1]   ==  a[n-2]
  ⋮     ⋮     ⋮
b[n-1] ==  a[ 0 ]
```

The ACSL specification of `reverse_copy` is shown in Listing 6.10.

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

Listing 6.10: Formal Specification of `reverse_copy`

The postcondition ensures that the contents of `a` were actually copied reversely to `b`.

[36] See `http://www.sgi.com/tech/stl/reverse_copy.html`.

6.5.2. Implementation of `reverse_copy`

Listing 6.11 shows an implementation of the `reverse_copy` function.

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

}
```

Listing 6.11: Implementation of `reverse_copy`

For the postcondition to be true, we must ensure that for every element `i`, the comparison `b[i] == a[n-1-i]` is `true`. This is formalized by the loop invariants. You can see that it is nearly similar to the postcondition from our specification in Listing 6.10.

6.6. The `reverse` Algorithm

The `reverse`[37] algorithm of the C++ Standard Library invert the order of elements in a sequence. The `reverse` algorithm works in place, meaning, that it modifies its input sequence.

For our purposes we have modified the generic implementations to that of a range of type **`value_type`**. The signature now reads:

```
void reverse(value_type* a, size_type n);
```

6.6.1. Formal Specification of `reverse`

The ACSL specification for the `reverse` function is shown in listing 6.12.

```
/*@
  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);
```

Listing 6.12: Formal Specification of `reverse`

The postcondition ensures that the contents of `a` is actually reversed.

[37] See `http://www.sgi.com/tech/stl/reverse.html`.

6.6.2. Implementation of `reverse`

Listing 6.13 shows an implementation of the `reverse` function.

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

  /*@
    loop invariant 0 <= first <= n;
    loop invariant 0 <= last <= n;

    loop invariant last == n-first;

    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..(n-1)], first, last;

    loop   variant last;
  */
  while (first < last)
  {
    swap(&a[first++], &a[--last]);
  }
}
```

Listing 6.13: Implementation of `reverse`

Here are some remarks on its loop invariants.

- The indices `first` and `last` are used to point at the elements which will be swapped in the current loop cycle. Therefore it is important that decreasing of `last` and increasing of `first` happen at the same time. Thus, the sum of the two indices is constant. By that and the initialization of both indices we can ensure that the right elements will be swapped.
- There are two loop invariants that express which array elements have already been reversed. One loop invariant expresses that the sequence from position `0` up to but not including `first` have already been reversed. The other loop invariant formalize that the elements which go from the position after `last` until `n-1` have already been reversed.

6.7. The `rotate_copy` Algorithm

The `rotate_copy` algorithm in the C++ Standard Library rotates a sequence by `m` positions and copies the results to another same sized sequence. For our purposes we have modified the generic implementation[38] to that of a range of type **`value_type`**. The signature now reads:

```
void rotate_copy(const value_type* a, size_type m,
                 size_type n, value_type* b);
```

Informally, the function copies the elements from the array `a` into array `b` such that the copy is a rotated version of the original array. In other words: After calling `rotate_copy` the following conditions shall be satisfied.

$$
\begin{array}{rcl}
\texttt{b[0]} & == & \texttt{a[m]} \\
\texttt{b[1]} & == & \texttt{a[m+1]} \\
\vdots & \vdots & \vdots \\
\texttt{b[n-m-1]} & == & \texttt{a[n-1]}
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{b[n-m]} & == & \texttt{a[0]} \\
\texttt{b[n-m+1]} & == & \texttt{a[1]} \\
\vdots & \vdots & \vdots \\
\texttt{b[n-1]} & == & \texttt{a[m-1]}
\end{array}
$$

This function refers to the previously discussed algorithm `copy`. Thus, `rotate_copy` serves as another example for "modular verification". The specification of `copy` will be integrated into the proof of `rotate_copy`.

[38] See `http://www.sgi.com/tech/stl/rotate_copy.html`.

6.7.1. Formal Specification of `rotate_copy`

The ACSL specification of `rotate_copy` is shown in Listing 6.14.

```
/*@
  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);
```

Listing 6.14: Formal Specification of `rotate_copy`

Here are some remarks on the specification

- The `separated`-clause tells WP that `a` and `b` must not overlap.
- The `assigns`-clause expresses that `rotate_copy` modifies only the elements in the range `b`.

6.7.2. Implementation of `rotate_copy`

Listing 6.15 shows an implementation of the `rotate_copy` function. In the `rotate_copy` function, we simply call the function `copy` twice.

```
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);
}
```

Listing 6.15: Implementation of `rotate_copy`

6.8. The `replace_copy` Algorithm

The `replace_copy` algorithm of the C++ Standard Library substitutes specific elements from general sequences. Here, the general implementation[39] has been altered to process **`value_type`** ranges. The new signature reads:

```
size_type replace_copy(const value_type* a, size_type n,
                       value_type* b,
                       value_type old_val, value_type new_val);
```

The `replace_copy` algorithm copies the elements from the range `a[0..n]` to range `b[0..n]`, substituting every occurrence of `old_val` by `new_val`. The return value is the length of the range. As the length of the range is already a parameter of the function this return value does not contain new information. However, the length returned is analogous to the implementation of the C++ Standard Library.

6.8.1. Formal Specification of `replace_copy`

The ACSL specification of `replace_copy` is shown in Listing 6.16.

```
/*@
  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);

  ensures \forall integer j; 0 <= j < n ==>
              (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);
```

Listing 6.16: Formal Specification of the `replace_copy` Function

In particular, the specification requires that the arrays `a` and `b` are non-overlapping. The core functionality of `replace_copy` is specified as follows: For every element `a[j]` of `a`, we have two possibilities. Either it equals `old_val` or it is different from `old_val`. In the former case, we specify that the corresponding element `b[j]` has to be substituted with `new_val`. In the latter case, we specify that `b[j]` must be `a[j]`.

[39] See `http://www.sgi.com/tech/stl/replace_copy.html`.

6.8.2. Implementation of `replace_copy`

An implementation (including loop annotations) of `replace_copy` is shown in Listing 6.17. Note how the structure of the loop annotations resembles the specification of Listing 6.16.

```
size_type replace_copy(const value_type* a, size_type n,
                       value_type* b,
                       value_type  old_val, value_type new_val)
{
    size_type i = 0;
  /*@
    loop invariant 0 <= i <= n;
    loop invariant \forall integer j; 0 <= j < i ==>
              (a[j] == old_val ==> b[j] == new_val);

    loop invariant \forall integer j; 0 <= j < i ==>
              (a[j] != old_val ==> b[j] == a[j]);

    loop assigns b[0..n-1], i;
    loop   variant n-i;
  */
  while (i < n)
  {
    b[i] = (a[i] == old_val ? new_val : a[i]);
    i++;
    //@ assert 0 < i <= n;
  }

  return n;
}
```

Listing 6.17: Implementation of the `replace_copy` algorithm

6.9. The `remove_copy` Algorithm

The `remove_copy` algorithm of the C++ Standard Library copies all elements of a sequence other than a given value. Here, the general implementation has been altered to process **`value_type`** ranges.[40] The new signature reads:

```
size_type remove_copy(const value_type* a, size_type n,
                      value_type* b, value_type v);
```

The most important facts of this algorithms are

1. The `remove_copy` algorithm copies elements that are not equal to `v` from range `a[0, n-1]` to a range beginning at `b[0]`.
2. The return value is the length of the resulting range.
3. The algorithm is stable, that is, the relative order of the elements in `b` is the same as in `a`.

6.9.1. Capturing the Stability of `remove_copy`

A particular challenge in the specification of `remove_copy` is how to express the stability of the removal. Figure 6.18 shows how `remove_copy` is supposed to copy elements that differ from `v` from one range to the other.

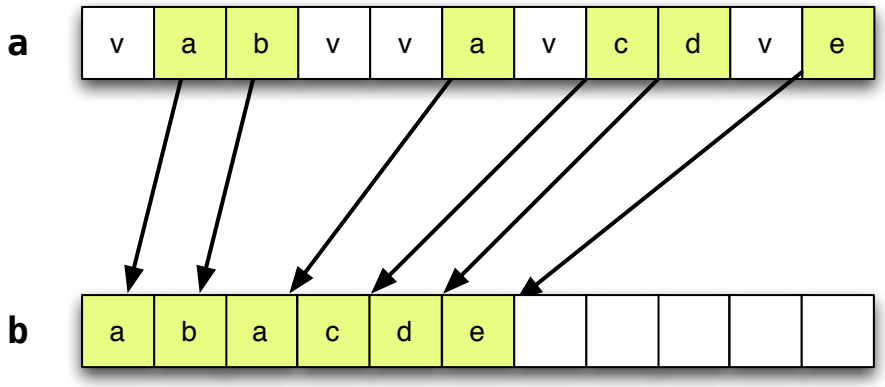


Figure 6.18.: Stability of `remove_copy`

[40] See `http://www.sgi.com/tech/stl/remove_copy.html`.

Figure 6.19 shows, with respect to array indices, how the elements different from `v` "slide" to positions with smaller indices. The main observation here is that *an element slides as many positions down as there are elements in front of it that equal* `v`.

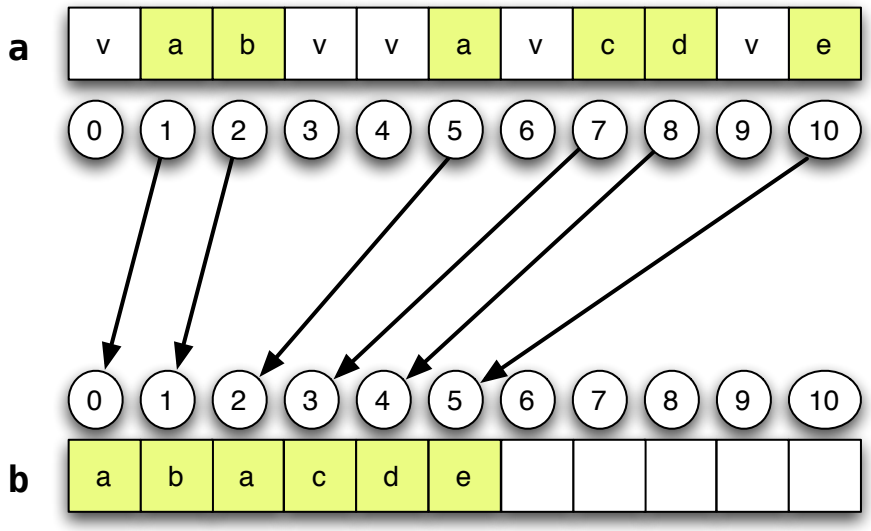


Figure 6.19.: Stability of `remove_copy` with respect to indices

As it turns out, it is quite easy[41] to express this property using the previously introduced logic function `Count` (see Listing 3.24 on Page 42). We simply define in Listing 6.20 a logic function `RemoveMap` which subtracts from every position `i` the number of occurrences of `v` that come before `i`.

```
/*@
  logic
    integer RemoveMap{L}(value_type* a, value_type v, integer i) =
      i - Count{L}(a, v, 0, i);
*/
```

Listing 6.20: The Logic Function `RemoveMap`

Note that `RemoveMap` is defined for all integers, including those indices `i` where `a[i]` equals `v`. In the specification of `remove_copy`, however, we will only use `RemoveMap` for indices where `a[i]` is different from `v`.

[41] To tell the truth, it took us quite some time to really understand how easy it is!

6.9.2. Formal Specification of `remove_copy`

We specify the function contract of `remove_copy` as shown in Listing 6.21.

```
/*@
  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 \forall integer i; 0 <= i < n ==>
              (a[i] != v ==> b[RemoveMap(a, v, i)] == a[i]);
  ensures \result == RemoveMap(a, v, n);
*/
size_type remove_copy(const value_type* a, size_type  n,
                      value_type* b, value_type v);
```

Listing 6.21: Formal Specification of the `remove_copy` Function

The algorithm `remove_copy` requires as a precondition that its two array elements do not overlap.

In the assigns close we would like to specify that `remove_copy` modifies the destination range only as much as there are copied elements. The most convenient way to specify this would be:

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

However, the `assigns` clause refers to the pre-state of the function, where the identifier `\result` cannot be used. We therefore add a line to express that the range `b[\result..(n-1)]` is left unchanged. Since we need to refer the function's post-state, we state this property by using a postcondition.

In the following postcondition we state an element `a[i]` that is different from `v` is copied to the position `RemoveMap(a, v, i)` of the range `b`. Note that `RemoveMap(a, v, n)` equals `n` decreased by the number of elements in the range `a[0]...a[n-1]` that equal v. This is indeed the expected result for the return value of `remove_copy`.

6.9.3. Implementation of `remove_copy`

An implementation of `remove_copy` is shown in Listing 6.22.

```
size_type remove_copy(const value_type* a, size_type n,
                      value_type* b, value_type v)
{
  size_type i = 0;
  size_type j = 0;

  /*@
    loop invariant 0 <= j <= i <= n;
    loop invariant j == RemoveMap(a, v, i);
    loop invariant \forall integer k; 0 <= k < i ==>
           (a[k] != v ==> b[RemoveMap(a, v, k)] == a[k]);
    loop assigns b[0..(n-1)], i, j;
    loop variant n - i;
  */
  while( i < n)
  {
    if (a[i] != v)
      b[j++] = a[i];
    ++i;
    //@ assert 0 < i <= n;
  }

  return j;
}
```

Listing 6.22: Implementation of the `remove_copy` Function

Almost each assertion of the loop invariant is motivated by a corresponding assertion of the specification. The former usually tells the provers how the latter is to be achieved during the loop at each step.

The first invariant indices `i` and `j` must stay within the range bounds. Moreover, we always have `j <= i`; the latter is needed to prove that `j <= n` holds. The next one corresponds to the two postconditions of the specification in Listing 6.21. And the last loop invariant is needed to prove the endmost postcondition of the specification.

The **`loop`** `assigns` clause ensures that nothing but `b[0..n-1]` and the local variables `i, j` are modified. It is necessary to prove the `assigns`-clause of the specification.

The loop variant `n-i` is always positive, and is decreased in every loop cycle, thus ensuring loop termination. To verify it successfully it is covered by an additional assertion within the loop body.

6.10. The `unique_copy` Algorithm

The `unique_copy` algorithm of the C++ Standard Library allows to remove duplicate elements from general sequences. Here, the general implementation[42] has been altered to process `value_type` ranges. The new signature reads:

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

The `unique_copy` algorithm copies elements in the range `a[0..n-1]` to a range beginning at `b[0]`, but of a consecutive group of equal elements only the first one is copied. The return value is the length of the resulting range.

6.10.1. A Predicate for `unique_copy`

The ACSL specification of `unique_copy` refers to the logic function `UniqueWhither` defined by axioms shown in Listing 6.23.

```
/*@
  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;

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

    lemma unique_5: \forall value_type *a, integer i, j;
      i < j ==> WhitherUnique(a, i) <= WhitherUnique(a, j);
  }
*/
```

Listing 6.23: The Logic Function `UniqueWhither`

The predicate `UniqueWhither(a, i)` helps to locate the positions of the elements in the result sequence. If an element is in position `i` in the source sequence, then it will appear in position `UniqueWhither(a, i)` in the result sequence, which means `a[i]=b[UniqueWhither(a, i)]`.

[42] See `http://www.sgi.com/tech/stl/unique_copy.html`.

Therefore `b[0..UniqueWhither(a, i)]` is a filtered copy of `a[0..i]`; in a consecutive group of equal elements only the first one is copied.[43] An example is shown in Figure 6.24, solid arrows indicating an actual copy, and dashed arrows indicating a redundant, and hence omitted, copy.

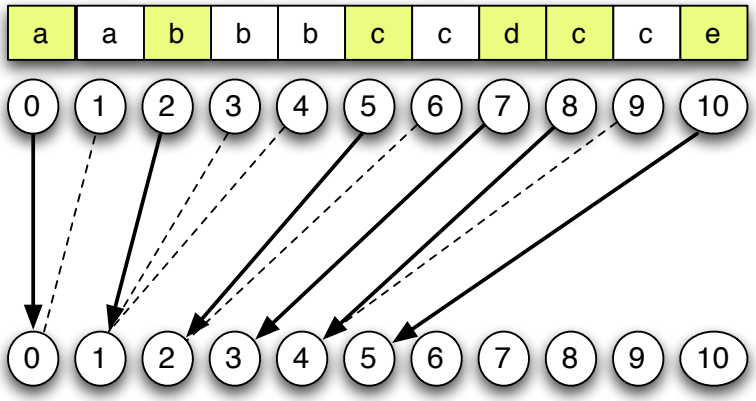


Figure 6.24.: An Example of the Logic Function `UniqueWhither`

We give some explanations about the axiomatic definition from Listing 6.23.

The first statement in the **`axiomatic`**-environment defines the signature of the predicate `UniqueWhither`.

Axiom unique_1 states the situation for the first element of `a`. It is simply copied to the beginning of the result sequence.

Axiom unique_2 states that the result sequence for `a[0..i]` has `UniqueWhither(a, i)` elements and that `a[i]` equals `a[i+1]`, i.e., the latter should be ignored. It states that under these circumstances, the result sequence for `a[0..i+1]` has also `UniqueWhither(a, i)` elements.

Axiom unique_3 states again that the result for `a[0..i]` has `UniqueWhither(a, i)` elements, but `a[i]` is now different from `a[i+1]`. It states that if in this situation `a[i+1]` is copied to the result sequence, then the result sequence for `a[i+1]` has `UniqueWhither(a, i)+1` elements.

Axiom unique_lemma_4 and 5 state the monotonicity of the logic function `UniqueWhither`, which will be used in the proof of the **`loop invariant`** in Listing 6.27. Although they both are consequences of axioms `unique_1..3`, we didn't prove, but simply claimed them here. These lemmas tell the provers how the latter is to be achieved, which means without the lemmas the provers will not know how to prove `unique_copy`.

[43] See 20 for more details on the reads clause.

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

Listing 6.25: The Predicate `UniqueCopy`

The ACSL specification of `unique_copy` refers to a predicate `UniqueCopy` shown in Listing 6.25. The predicate `UniqueCopy(a, n, b, m)` formalizes two facts. The first is that when the sequence `a` has at least one element, then `m-1` equals `UniqueWhither(a, n-1)`, which means the result sequence for `a[0..n-1]` has `m-1` elements. The second is that `a[i]` equals `b[UniqueWhither(a, i)]` for all `0 <= i < n`. Hence `b[0..m-1]` is a filtered copy of `a[0..n-1]` after removing duplicate elements.

6.10.2. Formal Specification of `unique_copy`

Using the predicate `UniqueCopy`, we can specify the function `unique_copy` in an elegant way.

```
/*@
  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);
```

Listing 6.26: Formal Specification of the `unique_copy` Function

For the first two preconditions we refer to Section 1.2 for how valid ranges are specified. The third one states that the source and destination range do not overlap.

The `assigns`-clause indicates that the algorithm modifies nothing but the destination range.

The postconditions for `unique_copy` must be defined with the **`ensures`**-clause. At first we state that the range of the result sequence must be equal to or smaller than that of the original sequence. The second postcondition expresses that `unique_copy` should behave according to `UniqueCopy`.

6.10.3. Implementation of `unique_copy`

An implementation (including loop annotations) of `unique_copy` is shown in Listing 6.27.

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

  size_type i = 1;
  size_type j = 0;
  b[j++] = a[0];

  /*@
    loop invariant 1 <= j <= i <= n;
    loop invariant UniqueCopy(a, i, b, j);
    loop assigns   b[1..n-1], i, j;
    loop variant   n - i;
  */
  while (i < n)
  {
    if (a[i] != a[i - 1])
      b[j++] = a[i];
    ++i;
    //@ assert 0 < i <= n;
  }

  return j;
}
```

Listing 6.27: Implementation of the `unique_copy` Algorithm

Almost each loop invariant is motivated by a corresponding assertion of the specification (Listing 6.26). The former usually tells the provers how the latter is to be achieved during the loop at each step. The first loop invariant covers the fact that the indices `i` and `j` must stay within the range bounds. With the next one we simply refer to the predicate `UniqueCopy` to specify the most essential part of the loop invariant. This is needed to prove our second postcondition of the specification in Listing 6.26.

The loop `assigns` clause ensures that nothing but `b[0..(n-1)], i, j` is modified. It is necessary to prove the corresponding `assigns`-clause of the specification in Listing 6.26.

The loop variant `n-i` is always positive, and is decreased in every loop cycle, thus ensuring loop termination for any input value of `n`.

6.11. The `iota` Algorithm

The `iota` algorithm in the C++ STL assigns sequentially increasing values to a range, where the start value is user defined. Our version of the original signature[44] reads:

```
void iota(value_type* a, size_type n, value_type val);
```

Starting at `val`, the function assigns consecutive integers to the range `a`. When specifying `iota` we must be careful to deal with possible overflows.

6.11.1. Formal Specification of `iota`

The ACSL specification of `iota` is shown in Listing 6.28.

Note that the specification of `iota` refers to `INT_MAX` which is defined in `limits.h`.

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

Listing 6.28: Formal Specification of `iota`

At first, the algorithm requires that `a` is a valid range. We refer to Section 1.2 for a discussion of how valid range are specified. Additional it is necessary to avoid integer overflow hence the length `n` of the array and the sum `val+n` must not greater than the constant `INT_MAX`.

The `iota` algorithm initializes and modifies the elements in range `a` our specification must contain the line `assigns a[0..n-1]`. Elements outside the range are not altered.

Upon termination, each element of `a` contains the sum of its index within `a` and the argument `val`.

[44] See `http://www.sgi.com/tech/stl/iota.html`.

6.11.2. Implementation of `iota`

Listing 6.29 shows an implementation of the `iota` function.

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

Listing 6.29: Implementation of `iota`

The first loop invariant indicates that the loop index must stay within bounds. The second one describes that in each iteration of the loop the current value `val` is equal to the sum of the value `val` in state of function entry and the loop index `i` (note the use of the `\at` clause here). This invariant is essential to prove the last invariant which represents the postcondition from our specification (see subsection 6.11.1 in Listing 6.28).

The `assigns` clause ensures that nothing but `a[0..(i-1)]`, `i` and `val` is modified.

The loop variant provides a positive expression which is decreased in every loop iteration to prove loop termination. Finally, we need one assertion in the loop body to prove the loop variant successfully.

7. The `Stack` Data Type

Originally, ACSL is tailored to the task of specifying and verifying one single C function at a time. However, in practice we are also faced with the task to implement a family of functions, usually around some sophisticated data structure, which have to obey certain rules of interdependence. In this kind of task, we are not interested in the properties of a single function (usually called "*implementation details*"), but in properties describing how several function play together (usually called "*abstract interface description*", or "*abstract data type properties*").

This chapter introduces a methodology to formally denote and verify the latter property sets using ACSL. For a more detailed discussion of our approach to the formal verification of `Stack` we refer to this thesis [15].

A *stack* is a data type that can hold objects and has the property that, if an object *a* is *pushed* on a stack *before* object *b*, then *a* can only be removed (*popped*) after *b*. A stack is, in other words, a *first-in, last-out* data type (see Figure 7.1). The *top* function of a stack returns the last element that has been pushed on a stack.

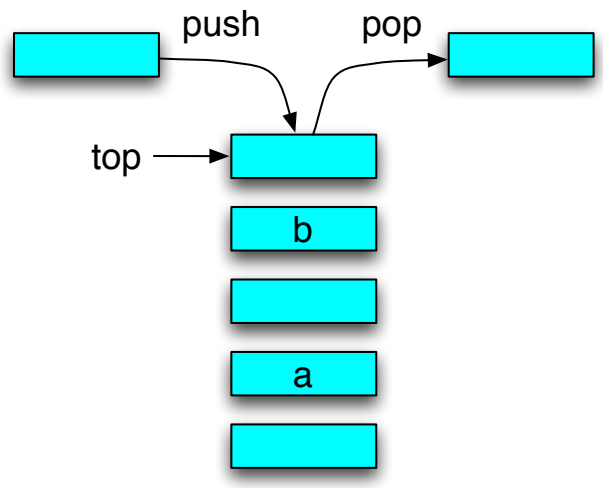


Figure 7.1.: Push and Pop on a Stack

We consider only stacks that have a finite *capacity*, that is, that can only hold a maximum number *c* of elements that is constant throughout their lifetime. This restriction allows us to define a stack without relying on dynamic memory allocation. When a stack is *created* or *initialized*, it contains no elements, i.e., its *size* is 0. The function *push* and *pop* increases and decreases the size of a stack by at most one, respectively.

7.1. Methodology Overview

Figure 7.2 gives an overview of our methodology to specify and verify abstract data types (verification of one axiom shown only).

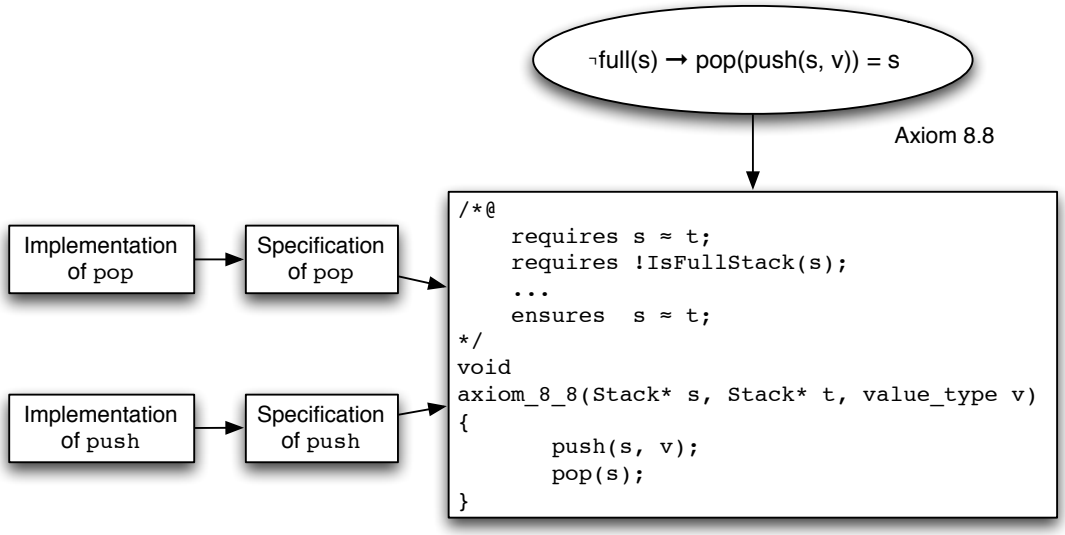


Figure 7.2.: Methodology Overview

What we will basically do is:

1. specify axioms about how the stack functions should interact with each other (Section 7.2),
2. define a basic implementation of C data structures (only one in our example, viz. `struct Stack`; see Section 7.3) and some invariants the instances of them have to obey (Section 7.4),
3. provide for each stack function an ACSL contract and a C implementation (Section 7.7),
4. verify each function against its contract (Section 7.7),
5. transform the axioms into ACSL-annotated C code (Section 7.8), and
6. verify that code, using access function contracts and data-type invariants as necessary (Section 7.8).

Section 7.5 provides an ACSL-predicate deciding whether two instances of a `struct Stack` are considered to be equal (indication by "≈" in Figure 7.2), while Section 7.6 gives a corresponding C implementation. The issue of an appropriate definition of equality of data instances is familiar to any C programmer who had to replace a faulty comparison `if(s1 == s2)` by the correct `if(strcmp(s1,s2) == 0)` to compare two strings `char *s1,*s2` for equality.

7.2. Stack Axioms

To specify the interplay of the stack access functions, we use a set of axioms[45], all but one of them having the form of a conditional equation.

Let V denote an arbitrary type. We denote by S_c the type of stacks with capacity $c > 0$ of elements of type V. The aforementioned functions then have the following signatures.

$$\begin{aligned} \text{init} &: S_c \to S_c, \\ \text{push} &: S_c \times V \to S_c, \\ \text{pop} &: S_c \to S_c, \\ \text{top} &: S_c \to V, \\ \text{size} &: S_c \to \mathbb{N}. \end{aligned}$$

With $\mathbb{B}$ denoting the *boolean* type we will also define two auxiliary functions

$$\begin{aligned} \text{empty} &: S_c \to \mathbb{B}, \\ \text{full} &: S_c \to \mathbb{B}. \end{aligned}$$

To qualify as a stack these functions must satisfy the following rules which are also referred to as *stack axioms*.

7.2.1. Stack Initialization

After a stack has been initialized its size is 0.

$$\text{size}(\text{init}(s)) = 0. \tag{7.1}$$

The auxiliary functions empty and full are defined as follows

$$\text{empty}(s), \quad \text{iff} \quad \text{size(s)} = 0, \tag{7.2}$$

$$\text{full}(s), \quad \text{iff} \quad \text{size(s)} = c. \tag{7.3}$$

We expect that for every stack s the following condition holds

$$0 \leq \text{size}(s) \leq c. \tag{7.4}$$

[45] There is an analogy in geometry: Euclid (e.g. [16]) invented the use of axioms there, but still kept definitions of *point*, *line*, *plane*, etc. Hilbert [17] recognized that the latter are not only unformalizable, but also unnecessary, and dropped them, keeping only the formal descriptions of relations between them.

7.2.2. Adding an Element to a Stack

To push an element v on a stack the stack must not be full. If an element has been pushed on an eligible stack, its size increases by 1

$$\text{size}(\text{push}(s, v)) = \text{size}(s) + 1, \qquad \text{if} \quad \neg\text{full}(s). \tag{7.5}$$

Moreover, the element pushed on a stack is the top element of the resulting stack

$$\text{top}(\text{push}(s, v)) = v, \qquad \text{if} \quad \neg\text{full}(s). \tag{7.6}$$

7.2.3. Removing an Element from a Stack

An element can only be removed from a non-empty stack. If an element has been removed from an eligible stack the stack size decreases by 1

$$\text{size}(\text{pop}(s)) = \text{size}(s) - 1, \qquad \text{if} \quad \neg\text{empty}(s). \tag{7.7}$$

If an element is pushed on a stack and immediately afterwards an element is removed from the resulting stack then the final stack is equal to the original stack

$$\text{pop}(\text{push}(s, v)) = s, \qquad \text{if} \quad \neg\text{full}(s). \tag{7.8}$$

Conversely, if an element is removed from a non-empty stack and if afterwards the top element of the original stack is pushed on the new stack then the resulting stack is equal to the original stack.

$$\text{push}(\text{pop}(s), \text{top}(s)) = s, \qquad \text{if} \quad \neg\text{empty}(s). \tag{7.9}$$

7.2.4. A Note on Exception Handling

We don't impose a requirement on `push(s, v)` if `s` is a full stack, nor on `pop(s)` or `top(s)` if `s` is an empty stack. Specifying the behavior in such *exceptional* situations is a problem by its own; a variety of approaches is discussed in the literature. We won't elaborate further on this issue, but only give an example to warn about "innocent-looking" exception specifications that may lead to undesired results.

If we'd introduce an additional error value `err` in the element type V and require `top(s) = err` if `s` is empty, we'd be faced with the problem of specifying the behavior of `push(s, err)`. At first glance, it would seem a good idea to have `err` just been ignored by `push`, i.e. to require

$$\text{push}(s, \text{err}) = s. \tag{7.10}$$

However, we then could derive for any non-full and non-empty stack `s`, that

	size(s)	
=	size(pop(push(s, err)))	by 7.8
=	size(pop(s))	as assumed in 7.10
=	size(s) - 1	by 7.7

i.e. no such stacks could exist, or all **`int`** values would be equal.

7.3. The Structure `Stack` and its Associated Functions

We now introduce one possible C implementation of the above axioms. It is centred around the C structure `Stack` shown in Listing 7.3.

```
struct Stack
{
  value_type* obj;

  size_type  capacity;

  size_type  size;
};

typedef struct Stack Stack;
```

Listing 7.3: Definition of `struct Stack`

This struct holds an array `obj` of non-zero length called `capacity`. The capacity of a stack is the maximum number of elements this stack can hold. The field `size` indicates the number elements that are currently in the stack. See also Figure 7.4 which attempts to interpret this definition according to Figure 7.1.

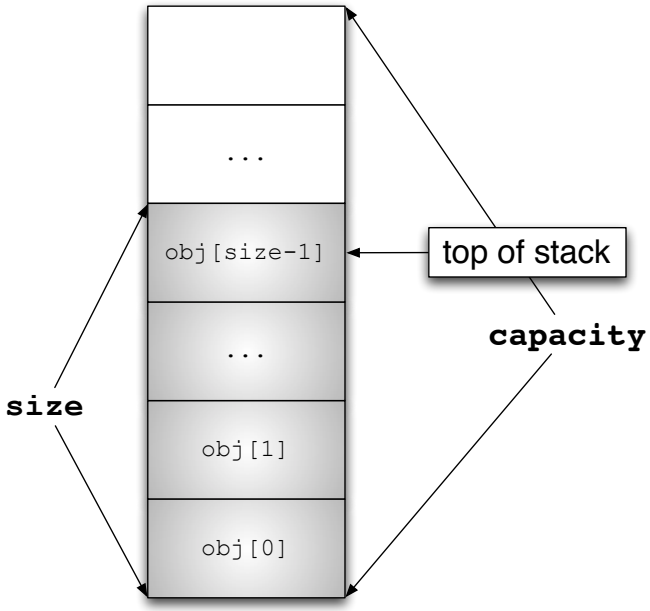


Figure 7.4.: Interpretation of `Stack`

Based on the stack functions from Section 7.2, we declare the following access functions as part of our `Stack` data type.

```
void       init_stack(Stack* s);

bool       equal_stack(const Stack* s, const Stack* t);

size_type  size_stack(const Stack* s);

bool       empty_stack(const Stack* s);

bool       full_stack(const Stack* s);

value_type top_stack(const Stack* s);

void       push_stack(Stack* s, value_type v);

void       pop_stack(Stack* s);
```

Most of these functions directly correspond to methods of the C++ `std::stack` template class.[46] The function `equal_stack` corresponds to the comparison operator `==`, whereas one use of `init_stack` is to bring a stack into a well-defined initial state. The function `full_stack` has no counterpart in `std::stack`. This reflects the fact that we avoid dynamic memory allocation, while `std::stack` does not.

7.4. Stack Invariants

Not every possible instance of `struct Stack` is considered a valid one, e.g., with our definition of `struct Stack` in Listing 7.3, `Stack s={{0,0,0,0},4,5}` is not. Below, we will define an ACSL-predicate `StackInvariants` discriminating valid and invalid instances.

Before, we introduce in Listing 7.5 the auxiliary logical function `CapacityOfStack` and `SizeOfStack` which we can use in specifications to refer to the fields `capacity` and `size` of `Stack`, respectively. This listing also contains the logical function `TopOfStack` which defines the array element with index `size-1` as the top place of a stack. The reader can consider this as an attempt to hide implementation details from the specification.

We also introduce in Listing 7.6 two predicates that express the concepts of empty and full stacks by referring to a stack's size and capacity (see Equations (7.2) and (7.3)).

There are some obvious invariants that must be fulfilled by every valid object of type `Stack`:

- The stack capacity shall be strictly greater than zero (an empty stack is ok but a stack that cannot hold anything is not useful).
- The pointer `obj` shall refer to an array of length `capacity`.
- The number of elements `size` of a stack the must be non-negative and not greater than its capacity.

[46] See `http://www.sgi.com/tech/stl/stack.html`

```
/*@
  logic size_type CapacityOfStack{L}(Stack* s) = s->capacity;

  logic size_type SizeOfStack{L}(Stack* s) = s->size;

  logic value_type* ObjOfStack{L}(Stack* s) = s->obj;

  logic value_type TopOfStack{L}(Stack* s) = s->obj[s->size-1];
*/
```

Listing 7.5: The Logical Functions `CapacityOfStack`, `SizeOfStack` and `TopOfStack`

```
/*@
  predicate IsEmptyStack{L}(Stack* s) =
    SizeOfStack(s) == 0;

  predicate IsFullStack{L}(Stack* s) =
    SizeOfStack(s) == CapacityOfStack(s);
*/
```

Listing 7.6: Predicates for Empty an Full Stacks

These invariants are formalized in the predicate `IsValidStack` of Listing 7.7.

```
/*@
  predicate
    IsValidStack{L}(Stack* s) = \valid(s) &&
      \separated(s, s->obj + (0..s->capacity-1)) &&
      0 < CapacityOfStack(s) &&
      0 <= SizeOfStack(s) <= CapacityOfStack(s) &&
      IsValidRange(s->obj, CapacityOfStack(s));
*/
```

Listing 7.7: The Predicate `IsValidStack`

Note how the use of the previously defined logical functions and predicates (including the predicate
`IsValidRange` from Listing 1.1 on Page 12) allows us to define the stack invariant without directly referring to the fields of `Stack`. As we usually have to deal with a pointer `s` of type `Stack` we add the necessary `valid(s)` to the predicate `IsValidRange`.

7.5. Equality of Stacks

Defining equality of instances of non-trivial data types, in particular in object-oriented languages, is not a trivial task. The book *Programming in Scala*[18, Chapter 28] devotes to this topic a whole chapter of more than twenty pages. In the following two sections we give a few hints how ACSL and Frama-C can help to correctly define equality for a simple data type.

We consider two stacks as equal if they have the same capacity, the same size and if they contain the same objects. To be more precise, let `s` and `t` two pointers of type `Stack`, then we define the predicate `IsEqualStack` as in Listing 7.8.

```
/*@
   predicate
     IsEqualStack{S,T}(Stack* s, Stack* t) =
       CapacityOfStack{S}(s) == CapacityOfStack{T}(t) &&
       SizeOfStack{S}(s) == SizeOfStack{T}(t) &&
       IsEqual{S,T}(ObjOfStack{S}(s), SizeOfStack{S}(s), ObjOfStack{T
           }(t));
*/
```

Listing 7.8: Equality of Stacks

Our use of labels in Listing 7.8 makes the specification somewhat hard to read (in particular in the last line where we reuse the predicate `IsEqual` from Page 25). However, this definition of `IsEqualStack` will allow us later to compare the same stack object at different points of a program. The logical expression `IsEqualStack{A,B}(s,t)` reads informally as: The stack object `*s` at program point `A` equals the stack object `*t` at program point `B`.

The reader might wonder why we do include the capacity of a stack into the definition of stack equality. If equal stacks could have different capacities then, according to our definition of the predicate `IsFullStack` in Listing 7.6, we could have to equal stacks where one is full and the other one not.

A finer, but very important point in our specification of equality of stacks is that the elements of the arrays `s->obj` and `t->obj` are compared only up to `s->size` and *not* up to `s->capacity`. Thus the two stacks `s` and `t` in Figure 7.9 are considered equal although there is an obvious difference in their internal arrays.

Figure 7.9.: Example of Two Equal Stacks

If we define an equality relation (=) of objects for a data type such as `Stack`, we have to make sure that the following rules hold.

$$\text{reflexivity} \qquad \forall s \in S : s = s, \tag{7.11a}$$

$$\text{symmetry} \qquad \forall s, t \in S : s = t \implies t = s, \tag{7.11b}$$

$$\text{transitivity} \qquad \forall s, t, u \in S : s = t \wedge t = u \implies s = u. \tag{7.11c}$$

Any relation that satisfies the conditions (7.11) is referred to as an *equivalence relation*. The mathematical set of all instances that are considered equal to some given instance `s` is called the equivalence class of `s` with respect to that relation.

Listing 7.10 shows a formalization of these three rules for the relation `IsEqualStack`; it can be automatically verified that they are a consequence of the definition of `IsEqualStack` in Listing 7.8.

```
/*@
  lemma IsEqualStackReflexive{S} :
    \forall Stack* s; IsEqualStack{S,S}(s, s);

  lemma IsEqualStackSymmetric{S,T} :
    \forall Stack *s, *t;
      IsEqualStack{S,T}(s, t) ==> IsEqualStack{T,S}(t, s);

  lemma IsEqualStackTransitive{S,T,U}:
    \forall Stack *s, *t, *u;
      IsEqualStack{S,T}(s, t) && IsEqualStack{T,U}(t, u) ==>
        IsEqualStack{S,U}(s, u);
*/
```

Listing 7.10: Equality of Stacks is an Equivalence Relation

The two stacks in Figure 7.9 show that an equivalence class of `IsEqualStack` can contain more than one element.[47] The stacks `s` and `t` in Figure 7.9 are also referred to as two *representatives* of the same equivalence class. In such a situation, the question arises whether a function that is defined on a set with an equivalence relation can be defined in such a way that its definition is *independent of the chosen representatives*.[48] We ask, in other words, whether the function is *well-defined* on the set of all equivalence classes of the relation `IsEqualStack`.[49] The question of well-definition will play an important role when verifying the functions of the `Stack` (see Section 7.7).

[47] This is a common situation in mathematics. For example, the equivalence class of the rational number $\frac{1}{2}$ contains infinitely many elements, viz. $\frac{1}{2}, \frac{2}{4}, \frac{7}{14}, \ldots$.

[48] This is why mathematicians have to *prove* that $\frac{1}{2} + \frac{3}{5}$ equals $\frac{7}{14} + \frac{3}{5}$.

[49] See `http://en.wikipedia.org/wiki/Well-definition`.

7.6. Runtime Equality of Stacks

The function `equal_stack` is the C equivalent for the `IsEqualStack` predicate. The specification of `equal_stack` is shown in Listing 7.11. Note that this specifications explicitly refers to valid stacks.

```
/*@
    requires IsValidStack(s);
    requires IsValidStack(t);

    assigns \nothing;

    ensures \result  <==>  IsEqualStack{Here,Here}(s, t);
    ensures \result == true || \result == false;
*/
bool equal_stack(const Stack* s, const Stack* t);
```

Listing 7.11: Specification of `equal_stack`

The implementation of `equal_stack` in Listing 7.12 compares two stacks according to the same rules as those of `IsEqualStack`.

```
bool equal_stack(const Stack* s, const Stack* t)
{
  //@ assert IsValidRange(s->obj, s->size);
  //@ assert IsValidRange(t->obj, t->size);
  return (s->capacity == t->capacity) &&
         (s->size == t->size) &&
         equal(s->obj, s->size, t->obj);
}
```

Listing 7.12: Implementation of `equal_stack`

7.7. Verification of Stack Functions

In this section we verify the functions `init_stack` (Section 7.7.1), `size_stack` (Section 7.7.2), `empty_stack` (Section 7.7.3), `full_stack` (Section 7.7.4), `top_stack` (Section 7.7.5), and `push_stack` (Section 7.7.6) `pop_stack` (Section 7.7.7), of the data type `Stack`. To be more precise, we provide for each of function `foo_stack`:

- an ACSL specification of `foo_stack`
- a C implementation of `foo_stack`
- a C function `foo_stack_wd`[50] accompanied by a an ACSL contract that expresses that the implementation of `foo_stack` is well-defined. Figure 7.13 shows our methodology for the verification of well-definition in the `pop` example, (≈) again indicating the user-defined `Stack` equality.

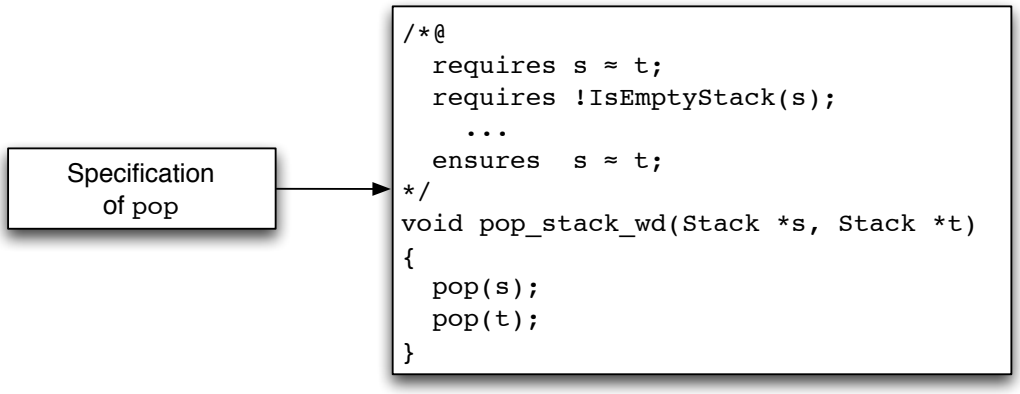


Figure 7.13.: Methodology for the verification of well-definition

Note that the specifications of the various functions will explicitly refer to the *internal state* of `Stack`. In Section 7.8 we will show that the *interplay* of these functions satisfy the stack axioms from Section 7.2.

[50]The suffix `_wd` stands for *well definition*.

7.7.1. The Function `init_stack`

Listing 7.14 shows the ACSL specification of `init_stack`. Note that our specification of the post-conditions contains a redundancy because a stack is empty if and only if its size is zero.

```
/*@
  requires \valid(s);
  requires 0 < capacity;
  requires IsValidRange(storage, capacity);
  requires \separated(s, storage + (0..capacity-1));

  assigns s->obj;
  assigns s->capacity;
  assigns s->size;

  ensures IsValidStack(s);
  ensures CapacityOfStack(s) == capacity;
  ensures SizeOfStack(s) == 0;
  ensures IsEmptyStack(s);
  ensures ObjOfStack(s) == storage;
*/
void init_stack(Stack* s, value_type* storage, size_type capacity);
```

Listing 7.14: Specification of `init_stack`

Listing 7.14 shows the implementation of `init_stack`. It simply sets the field `size` to zero.

```
void init_stack(Stack* s, value_type* storage, size_type capacity)
{
  s->obj      = storage;
  s->capacity = capacity;
  //@ assert \separated(s, s->obj + (0..s->capacity-1));
  s->size     = 0;
}
```

Listing 7.15: Implementation of `init_stack`

7.7.2. The Function `size_stack`

The function `size_stack` is the runtime version of the logical function `SizeOfStack` from Listing 7.5 on Page 103. The specification of `size_stack` in Listing 7.16 simply states that `size_stack` produces the same result as `SizeOfStack`.

```
/*@
    requires IsValidStack(s);

    assigns \nothing;

    ensures \result == SizeOfStack(s);
*/
size_type size_stack(const Stack* s);
```

Listing 7.16: Specification of `size_stack`

As in the definition of the logical function `SizeOfStack` the implementation of `size_stack` in Figure 7.17 simply returns the field `size`.

```
size_type size_stack(const Stack* s)
{
  return s->size;
}
```

Listing 7.17: Implementation of `size_stack`

Listing 7.18 shows our check whether `size_stack` is well-defined. Since `size_stack` neither modifies the state of its `Stack` argument nor that of any global variable we only check whether it produces the same result for equal stacks. Note that we simply may use operator == to compare integers since we didn't introduce a nontrivial equivalence relation on that data type.

```
/*@
  requires IsValidStack(s) && IsValidStack(t);
  requires IsEqualStack{Here,Here}(s, t);

  assigns \nothing;

  ensures \result;
*/
bool size_stack_wd(const Stack* s, const Stack* t)
{
  return size_stack(s) == size_stack(t);
}
```

Listing 7.18: Well-definition of `size_stack`

7.7.3. The Function `empty_stack`

The function `empty_stack` is the runtime version of the predicate `IsEmptyStack` from Listing 7.6 on Page 103.

```
/*@
    requires IsValidStack(s);

    assigns \nothing;

    ensures \result <==> IsEmptyStack(s);
    ensures \result == true || \result == false;
*/
bool empty_stack(const Stack* s);
```

Listing 7.19: Specification of `empty_stack`

We need to make sure that the result of `empty_stack` can only be `true` or `false`, that is `1` or `0`.[51]

As in the definition of the predicate `IsEmptyStack` the implementation of `empty_stack` in Figure 7.20 simply checks whether the size of the stack is zero.

```
bool empty_stack(const Stack* s)
{
  return size_stack(s) == 0;
}
```

Listing 7.20: Implementation of `empty_stack`

Listing 7.21 shows our check whether `empty_stack` is well-defined.

```
/*@
  requires IsValidStack(s);
  requires IsValidStack(t);
  requires IsEqualStack{Here,Here}(s, t);

  assigns \nothing;

  ensures \result;
*/
bool empty_stack_wd(const Stack* s, const Stack* t)
{
  return empty_stack(s) == empty_stack(t);
}
```

Listing 7.21: Well-definition of `empty_stack`

[51] If `true` could denote any nonzero value, we could not use the built-in `==` operator to check the well-definition in `empty_stack_wd` (Listing 7.21), since e.g. `empty_stack(s)` and `empty_stack(t)` could yield the value `2` and `4`, respectively. While both values denote are evaluated as true in C, the comparison `2 == 4` fails. See also the discussion of type **`bool`** in Section 1.2.

7.7.4. The Function `full_stack`

The function `full_stack` is the runtime version of the predicate `IsFullStack` from Listing 7.6 on Page 103.

```
/*@
    requires IsValidStack(s);

    assigns \nothing;

    ensures \result == true || \result == false;
    ensures \result <==> IsFullStack(s);
*/
bool full_stack(const Stack* s);
```

Listing 7.22: Specification of `full_stack`

As in the definition of the predicate `IsFullStack` the implementation of `full_stack` in Figure 7.23 simply checks whether the size of the stack equals its capacity.

```
bool full_stack(const Stack* s)
{
  return size_stack(s) == s->capacity;
}
```

Listing 7.23: Implementation of `full_stack`

Listing 7.24 shows our check whether `full_stack` is well-defined.

```
/*@
  requires IsValidStack(s);
  requires IsValidStack(t);
  requires IsEqualStack{Here,Here}(s, t);

  assigns \nothing;

  ensures \result;
*/
bool full_stack_wd(const Stack* s, const Stack* t)
{
  return full_stack(s) == full_stack(t);
}
```

Listing 7.24: Well-definition of `full_stack`

Observe that the well-definition of `full_stack` depends on the fact that we consider two stacks as equal only if their capacities agree, cf. Listing 7.8. Without that restriction, e.g. the stacks `{{6,7,8,9},4,4}` and `{{6,7,8,9,0,0,0,0},8,4}` would be considered equal, although the former is full while the latter is not.

7.7.5. The Function `top_stack`

The function `top_stack` is the runtime version of the logical function `TopOfStack` from Listing 7.5 on Page 103. The specification of `top_stack` in Listing 7.25 simply states that for non-empty stacks `top_stack` produces the same result as `TopOfStack` which in turn just returns the element `obj[size-1]` of `Stack`.

```
/*@
    requires IsValidStack(s);

    assigns \nothing;

    ensures !IsEmptyStack(s) ==> \result == TopOfStack(s);
*/
value_type top_stack(const Stack* s);
```

Listing 7.25: Specification of `top_stack`

For a non-empty stack the implementation of `top_stack` in Figure 7.26 simply returns the element `obj[size-1]`. Note that our implementation of `top_stack` does not crash when it is applied to an empty stack. In this case we return the first element of the internal, non-empty array `obj`. This is consistent with our specification of `top_stack` which only refers to non-empty stacks.

```
value_type top_stack(const Stack* s)
{
  // repeat known facts to help WP
  //@ assert IsValidRange(s->obj, s->size);
  if (!empty_stack(s))
  {
    return s->obj[s->size - 1];
  }
  else
  {
    return s->obj[0];
  }
}
```

Listing 7.26: Implementation of `top_stack`

Listing 7.27 shows our check whether `top_stack` well-defined for non-empty stacks.

```
/*@
  requires IsValidStack(s) && !IsEmptyStack(s);
  requires IsValidStack(t) && !IsEmptyStack(t);
  requires IsEqualStack{Here,Here}(s, t);

  assigns \nothing;

  ensures \result;
*/
bool top_stack_wd(const Stack* s, const Stack* t)
{
  return top_stack(s) == top_stack(t);
}
```

Listing 7.27: Well-definition of `top_stack`

Since our axioms in Section 7.2 did not impose any restriction on the behavior of `top_stack(s)` for an empty `s`, it is sufficient to prove well-definition of `top_stack(s)` only for nonempty `s`. This justifies the restricting `...&&!IsEmptyStack(...)` in the first and second **`requires`** clause.

7.7.6. The Function `push_stack`

Listing 7.28 shows the ACSL specification of the function `push_stack`. In accordance with Axiom (7.5), `push_stack` is supposed to increase the number of elements of a non-full stack by one. The specification also demands that the value that is pushed on a non-full stack becomes the top element of the resulting stack (see Axiom (7.6)).

```
// Pre is explicitly mentioned in assumes clauses.
// This is due to an error in the coq representation with the store
   model.
/*@
  requires IsValidStack(s);

  assigns s->size;
  assigns s->obj[s->size];

  behavior not_full:
    assumes !IsFullStack{Pre}(s);

    assigns s->size;
    assigns s->obj[s->size];

    ensures IsValidStack(s);
    ensures SizeOfStack(s) == SizeOfStack{Old}(s) + 1;
    ensures TopOfStack(s) == v;
    ensures !IsEmptyStack(s);
    ensures IsEqual{Pre, Here}(ObjOfStack{Pre}(s), SizeOfStack{Pre}(s
       ), ObjOfStack{Here}(s));
    ensures ObjOfStack(s) == ObjOfStack{Old}(s);
    ensures CapacityOfStack(s) == CapacityOfStack{Old}(s);

  behavior full:
    assumes IsFullStack{Pre}(s);

    assigns \nothing;

    ensures IsValidStack(s);
    ensures IsFullStack(s);
    ensures IsEqual{Pre, Here}(ObjOfStack{Pre}(s), SizeOfStack{Here}(
       s), ObjOfStack{Here}(s));
    ensures SizeOfStack(s) == SizeOfStack{Old}(s);
    ensures ObjOfStack(s) == ObjOfStack{Old}(s);
    ensures CapacityOfStack(s) == CapacityOfStack{Old}(s);

  complete behaviors;
  disjoint behaviors;
*/
void push_stack(Stack* s, value_type v);
```

Listing 7.28: Specification of `push_stack`

The implementation of `push_stack` is shown in Listing 7.29. It checks whether its argument is a non-full stack in which case it increases the field `size` by one but only after it has assigned the function argument to the element `obj[size]`.

```
void push_stack(Stack* s, value_type v)
{
  if (!full_stack(s))
  {
    //@ assert IsValidRange(s->obj, s->size);
    //@ assert s->size < s->capacity;
    s->obj[s->size++] = v;
  }
}
```

Listing 7.29: Implementation of `push_stack`

Listing 7.30 shows our check whether `push_stack` is well-defined. Since `push_stack` does not return a value but rather modifies its argument we check whether it turns equal stacks into equal stacks.

```
/*@
  requires IsValidStack(s);
  requires IsValidStack(t);
  requires IsEqualStack{Here,Here}(s, t);
  requires !IsFullStack(s);
  requires !IsFullStack(t);
  requires SeparatedStacks(s, t);

  ensures IsValidStack(s);
  ensures IsValidStack(t);
  ensures IsEqualStack{Here,Here}(s, t);
*/
void push_stack_wd(Stack* s, Stack* t, value_type v)
{
  push_stack(s, v);
  //@ ghost Between:

  //@ assert CapacityOfStack{Pre}(t) == CapacityOfStack{Here}(t);
  //@ assert SizeOfStack{Pre}(t) == SizeOfStack{Here}(t);
  //@ assert ObjOfStack{Pre}(t) == ObjOfStack{Here}(t);

  //@ assert IsEqual{Pre, Here}(ObjOfStack{Pre}(t), SizeOfStack{Here
     }(t), ObjOfStack{Here}(t));
  //@ assert IsEqual{Here,Here}(ObjOfStack(t), SizeOfStack{Here}(t),
     ObjOfStack(s));

  //@ assert !IsFullStack(t);
  //@ assert IsValidStack(t);
  //@ assert SeparatedStacks(s,t);
  push_stack(t, v);
  //@ assert CapacityOfStack(s) == CapacityOfStack(t);
  //@ assert SizeOfStack(s) == SizeOfStack(t);

  //@ assert ObjOfStack{Between}(s) == ObjOfStack{Here}(s);

  //@ assert IsEqual{Between, Here}(ObjOfStack{Between}(s),
     SizeOfStack{Here}(s), ObjOfStack(s));
  //@ assert IsEqual{Here,Between}(ObjOfStack(t), SizeOfStack{Between
     }(t), ObjOfStack{Between}(s));

  //@ assert CapacityOfStack{Between}(s) == CapacityOfStack{Here}(s);
  //@ assert 0 <= SizeOfStack{Here}(s) <= CapacityOfStack{Here}(s);
}
```

Listing 7.30: Well-definition of `push_stack`

7.7.7. The Function `pop_stack`

Listing 7.31 shows the ACSL specification of the function `pop_stack`. In accordance with Axiom (7.7) `pop_stack` is supposed to reduce the number of elements in a non-empty stack by one. In addition to the requirements imposed by the axioms, our specification demands that `pop_stack` changes no memory location if it is applied to an empty stack.

```
// Pre is explicitly mentioned in assumes clauses.
// This is due to an error in the coq representation with the store
   model.
/*@
  requires IsValidStack(s);
  ensures  IsValidStack(s);

  assigns s->size;

  behavior not_empty:
    assumes !IsEmptyStack{Pre}(s);

    assigns s->size;

    ensures SizeOfStack(s) == SizeOfStack{Old}(s) - 1;
    ensures !IsFullStack(s);
    ensures IsEqual{Pre, Here}(ObjOfStack{Pre}(s), SizeOfStack{Here}(
       s), ObjOfStack{Here}(s));
    ensures ObjOfStack(s) == ObjOfStack{Old}(s);
    ensures CapacityOfStack(s) == CapacityOfStack{Old}(s);

  behavior empty:
    assumes IsEmptyStack{Pre}(s);

    assigns \nothing;

    ensures IsEmptyStack(s);
    ensures IsEqual{Pre, Here}(ObjOfStack{Pre}(s), SizeOfStack{Here}(
       s), ObjOfStack{Here}(s));
    ensures SizeOfStack(s) == SizeOfStack{Old}(s);
    ensures ObjOfStack(s) == ObjOfStack{Old}(s);
    ensures CapacityOfStack(s) == CapacityOfStack{Old}(s);

  complete behaviors;
  disjoint behaviors;
*/
void pop_stack(Stack* s);
```

Listing 7.31: Specification of `pop_stack`

The implementation of `pop_stack` is shown in Listing 7.32. It checks whether its argument is a non-empty stack in which case it decreases the field `size` by one.

```
void pop_stack(Stack* s)
{
  if (!empty_stack(s))
  {
    //@ assert 0 <  s->size <= s->capacity;
    --s->size;
  }
}
```

Listing 7.32: Implementation of `pop_stack`

Listing 7.33 shows our check whether `pop_stack` is well-defined. Since `pop_stack` does not return a value but rather modifies its argument we check whether it turns equal stacks into equal stacks.

```
/*@
  requires IsValidStack(s);
  requires IsValidStack(t);
  requires IsEqualStack{Here,Here}(s, t);
  requires SeparatedStacks(s, t);

  assigns s->size;
  assigns t->size;

  ensures IsValidStack(s);
  ensures IsValidStack(t);
  ensures IsEqualStack{Here,Here}(s, t);
*/
void pop_stack_wd(Stack* s, Stack* t)
{
  //@ assert (IsEmptyStack{Here}(s) || !IsEmptyStack{Here}(s));
  pop_stack(s);
  //@ ghost Between:

  //@ assert CapacityOfStack{Pre}(t) == CapacityOfStack{Here}(t);
  //@ assert SizeOfStack{Pre}(t) == SizeOfStack{Here}(t);
  //@ assert ObjOfStack{Pre}(t) == ObjOfStack{Here}(t);

  //@ assert IsEqual{Pre,Here}(ObjOfStack{Pre}(t), SizeOfStack{Here}(
     t), ObjOfStack(t));
  //@ assert IsEqual{Here,Here}(ObjOfStack(t), SizeOfStack{Here}(s),
     ObjOfStack(s));

  //@ assert IsEmptyStack{Here}(t) || !IsEmptyStack{Here}(t);
  //@ assert IsEmptyStack{Here}(t) <==> IsEmptyStack{Pre}(s);
  //@ assert SeparatedStacks(s, t);
  pop_stack(t);

  //@ assert CapacityOfStack(s) == CapacityOfStack(t);
  //@ assert SizeOfStack(s) == SizeOfStack(t);

  //@ assert ObjOfStack{Between}(s) == ObjOfStack{Here}(s);


  //@ assert IsEqual{Between,Here}(ObjOfStack{Between}(s),
     SizeOfStack{Here}(s), ObjOfStack(s));
  //@ assert IsEqual{Here,Between}(ObjOfStack(t), SizeOfStack{Between
     }(s), ObjOfStack{Between}(s));

  //@ assert CapacityOfStack{Between}(s) == CapacityOfStack{Here}(s);
  //@ assert 0 <= SizeOfStack{Here}(s) <= CapacityOfStack{Here}(s);
}
```

Listing 7.33: Well-definition of `pop_stack`

7.8. Verification of Stack Axioms

In this section we show that the stack functions defined in Section 7.7 satisfy the stack axioms of Section 7.2.

7.8.1. Resetting a Stack

Our formulation in ACSL/C of the axiom in Equation (7.1) on Page 99 is shown in Listing 7.34.

```
/*@
  requires \valid(s);
  requires 0 < capacity;
  requires IsValidRange(storage, capacity);
  requires \separated(s, storage + (0..capacity-1));

  assigns s->obj, s->capacity, s->size;

  ensures IsValidStack(s);
  ensures \result == 0;
*/
size_type axiom_size_of_init(Stack* s, value_type* storage, size_type
    capacity)
{
  init_stack(s, storage, capacity);
  return size_stack(s);
}
```

Listing 7.34: Specification of Axiom (7.1)

7.8.2. Adding an Element to a Stack

Axioms (7.5) and (7.6) describe the behavior of a stack when an element is added.

```
/*@
  requires IsValidStack(s);
  requires !IsFullStack(s);

  assigns s->size;
  assigns s->obj[s->size];

  ensures IsValidStack(s);
  ensures \result == SizeOfStack{Old}(s) + 1;
*/
size_type axiom_size_of_push(Stack* s, value_type v)
{
  push_stack(s, v);
  return size_stack(s);
}
```

Listing 7.35: Specification of Axiom (7.5)

Except for the `assigns` clauses, the ACSL-specification refers only to encapsulating logical functions and predicates defined in Section 7.4. If ACSL would provide a means to define encapsulating logical functions returning also sets of memory locations, the expressions in `assigns` clauses would not need to refer to the details of our `Stack` implementation.[52] As an alternative, `assigns` clauses could be omitted, as long as the proofs are only used to convince a human reader.

```
/*@
  requires IsValidStack(s);
  requires !IsFullStack(s);

  assigns s->size;
  assigns s->obj[s->size];

  ensures \result == v;
*/
value_type axiom_top_of_push(Stack* s, value_type v)
{
  push_stack(s, v);
  return top_stack(s);
}
```

Listing 7.36: Specification of Axiom (7.6)

[52] In [9, § 2.3.4], a powerful sublanguage to build memory location set expressions is defined, lacking, however, just function definitions.

7.8.3. Removing an Element from a Stack

This section shows the Listings for axioms 7.7, 7.8 and 7.9 which describe the behavior of a stack when an element is removed.

```
/*@
  requires IsValidStack(s);
  requires !IsFullStack(s);

  assigns s->size;
  assigns s->obj[s->size];

  ensures IsEqualStack{Pre ,Here}(s, s);
*/
void axiom_pop_of_push(Stack* s, value_type v)
{
  push_stack(s, v);
  pop_stack(s);
}
```

Listing 7.37: Specification of Axiom (7.8)

The annotated code has been obtained from the axiom in a fully systematical way. In order to transform a condition equation $p \rightarrow s = t$:

- Generate a clause **requires** `p`.
- Generate a clause **requires** `x1 == ... == xn` for each variable `x` with n occurrences in s and t.
- Change the i-th occurrence of `x` to `xi` in s and t.
- Translate both terms s and t to reversed polish notation.
- Generate a clause **ensures** `y1 == y2`, where `y1` and `y2` denote the value corresponding to the translated s and t, respectively.

This makes it easy to implement a tool that does the translation automatically, but yields a slightly longer contract in our example. For other data types, employing axioms with non-trivial left- and right-hand-sides, e.g. for queues,

$$(\neg\text{full}(q) \wedge \neg\text{empty}(q)) \quad \rightarrow \quad \text{pop}(\text{push}(q, v)) = \text{push}(\text{pop}(q), v)$$

the contract length is not increased.

```
/*@
  requires IsValidStack(s);
  requires !IsEmptyStack(s);

  assigns s->size;

  ensures \result == SizeOfStack{Old}(s) - 1;
*/
size_type axiom_size_of_pop(Stack* s)
{
  pop_stack(s);
  return size_stack(s);
}
```

Listing 7.38: Specification of Axiom (7.7)

```
/*@
  requires IsValidStack(s) && !IsEmptyStack(s);

  assigns s->size;
  assigns s->obj[s->size-1];

  ensures IsEqualStack{Here,Old}(s, s);
*/
void axiom_push_of_pop_top(Stack* s)
{
  const value_type val = top_stack(s);
  pop_stack(s);
  push_stack(s, val);
}
```

Listing 7.39: Specification of Axiom (7.9)

8. Formal Verification

In this chapter we introduce the formal verification tools used in this tutorial. We will afterwards present to what extent the examples from Chapters 3 – 7 could be deductively verified using the following verification tools.

For our experiments we used the WP plugin-in of Fluorine-20130601 release of Frama-C[53] together with automatic theorem prover Alt-Ergo (version 0.95.2):

- Within Frama-C, the WP plug-in [2] enables deductive verification of C programs that have been annotated with the ANSI/ISO-C Specification Language (ACSL)[1]. The WP plug-in uses weakest precondition computations to generate proof obligations. To formally prove the ACSL properties, these proof obligations can be submitted to external automatic theorem provers or interactive proof assistants.
- Alt-Ergo is an automatic theorem prover for program verification developed at LRI [19]. It is freely available under the terms of the CeCILL-C LICENSE.[54]

We called Frama-C with the following options.

```
frama-c
    -cpp-command 'gcc -C -E -nostdinc $(INCLUDES)'
    -pp-annot -no-unicode
    -wp -wp-rte -wp-model Typed+ref -wp-timeout 30
```

For each algorithm we list the number of generated verification conditions (VC), as well as the percentage of proven verification conditions. The tables show that not all verification conditions can be proven automatically.

[53] See http://frama-c.com/install-fluorine-20130601.html
[54] For Details see http://alt-ergo.lri.fr/http/CeCILL-C

Table 8.1 shows the results for the non-mutating algorithms (Chapter 3).

Algorithm	**Section**	**# VC**	**Percentage of Proven VCs**
`equal`	3.1	18	100 %
`equal` (`IsEqual`)	3.1	15	100 %
`equal` (`mismatch`)	3.2	7	100 %
`mismatch`	3.2	20	100 %
`find`	3.3	19	100 %
`find` (reconsidered)	3.4	19	100 %
`find_first_of`	3.5	28	100 %
`adjacent_find`	3.6	23	100 %
`search`	3.7	32	100 %
`count`	3.8	16	93 %

Table 8.1.: Results for Non-mutating Algorithms

Table 8.2 contains the results for the maximum and minimum algorithms (Chapter 4). In the first row we try to show that the < operator satisfies the properties of a partial order and to verify the semantics of the derived operators.

Algorithm	**Section**	**# VC**	**Percentage of Proven VCs**
partial order properties of operator <	4.1	6	100 %
`max_element`	4.2	25	100 %
`max_element` (variant)	4.3	25	100 %
`max_seq`	4.4	8	100 %
`min_element`	4.5	25	100 %

Table 8.2.: Results for Maximum and Minimum Algorithms

Table 8.3 depicts the results for the binary search algorithms described in Chapter 5.

Algorithm	**Section**	**# VC**	**Percentage of Proven VCs**
`lower_bound`	5.1	23	100 %
`upper_bound`	5.2	23	100 %
`equal_range`	5.3	16	68 %
`binary_search`	5.4	10	100 %
`binary_search` (`HasValue`)	5.4	10	100 %

Table 8.3.: Results for Binary Search Algorithms

Table 8.4 contains the results for the mutating algorithms (Chapter 6).

Algorithm	**Section**	**# VC**	**Percentage of Proven VCs**
`fill`	6.1	15	100 %
`swap`	6.2	10	100 %
`swap_ranges`	6.3	22	90 %
`copy`	6.4	16	100 %
`reverse_copy`	6.5	20	85 %
`reverse`	6.6	31	90 %
`rotate_copy`	6.7	16	100 %
`replace_copy`	6.8	23	100 %
`remove_copy`	6.9	28	82 %
`unique_copy`	6.10	34	88 %
`iota`	6.11	19	100 %

Table 8.4.: Results for Mutating Algorithms

Table 8.5 shows the verification results of the `Stack` functions.

Algorithm	Section	# VC	Percentage of Proven VCs
`equal_stack`	7.6	23	100 %
`init_stack`	7.7.1	15	100 %
`size_stack`	7.7.2	6	100 %
`empty_stack`	7.7.3	10	100 %
`full_stack`	7.7.4	11	100 %
`top_stack`	7.7.5	17	100 %
`push_stack`	7.7.6	45	100 %
`pop_stack`	7.7.7	33	100 %

Table 8.5.: Results for `Stack` functions

Table 8.6 shows the verification results for the well-definition of `Stack` functions.

Algorithm	Section	# VC	Percentage of Proven VCs
`size_stack_wd`	7.7.2	12	100 %
`empty_stack_wd`	7.7.3	12	100 %
`full_stack_wd`	7.7.4	12	100 %
`top_stack_wd`	7.7.5	12	100 %
`push_stack_wd`	7.7.6	23	100 %
`pop_stack_wd`	7.7.7	28	100 %

Table 8.6.: Results for the well-definition of the `Stack` functions

Table 8.7 shows the verification results for the `Stack` axioms.

Algorithm	Section	# VC	Percentage of Proven VCs
`axiom_size_of_init`	7.8.1	15	100 %
`axiom_size_of_push`	7.8.2	12	100 %
`axiom_top_of_push`	7.8.2	11	100 %
`axiom_pop_of_push`	7.8.3	10	100 %
`axiom_size_of_pop`	7.8.3	11	100 %
`axiom_push_of_pop_top`	7.8.3	15	100 %

Table 8.7.: Results for Stack Axioms

A. Changes

The version number of this document is related to the versioning of Frama-C [3]. The versions of Frama-C are named consecutively after the elements of the periodic table. Therefore, our version numbering (X.Y.Z) are constructed as follows:

X the major number of our tutorial is the atomic number[55] of the chemical element after which Frama-C is named.

Y the Frama-C subrelease number

Z the subrelease number of this tutorial

A.1. New in Version 9.3.0 (December 2013)

- adjustments for *Fluorine* release of Frama-C
- `swap` now ensures that its pointer arguments are valid after the function has been called
- change definition of **`size_type`** to `unsigned int`
- change implementation of the `iota` algorithm (Section 6.11). The content of the field `a` is calculated by increasing the value `val` instead of sum `val+i`.
- change verification of `fill`.
- The specification/implementation of `Stack` (Chapter 7) has been revised by Kim Völlinger [15] and now has a much better verification rate.

A.2. New in Version 8.1.0 (not published)

- simplified specification and loop annotations of `replace_copy` (Section 6.8)
- add binary search variant `equal_range` (Section 5.3)
- greatly simplified specification of `remove_copy` (Section 6.9) by using the logic function `Count`
- remove chapter on heap operations

[55] See `http://en.wikipedia.org/wiki/Atomic_number`

A.3. New in Version 7.1.1 (August 2012)

- improvements with respect to several suggestions and comments of Yannick Moy, e.g., specification refinements of `remove_copy`, `reverse_copy` and `iota`
- restricted verification of algorithms to WP with Alt-Ergo
- replaced deprecated `\valid_range` by `\valid` in definition of `IsValidRange`
- fixed inconsistencies in the description of the `Stack` data type (Chapter 7)
- binary search algorithms (Chapter 5) can now be proven without additional axioms for integer division
- changed axioms into lemmas to document that provability is expected, even if not currently granted
- adopted new Fraunhofer logo and contact email

A.4. New in Version 7.1.0 (December 2011)

- changed to Frama-C Nitrogen
- changed to Why 2.30
- discussed both plug-ins WP and Jessie
- removed `swap_values` algorithm

A.5. New in Version 6.1.0 (not published)

- changed definition of `Stack`
- renamed `reset_stack` to `init_stack`

A.6. New in Version 5.1.1 (February 2011)

- prepared algorithms for checking by the new WP plug-in of Frama-C
- changed to Alt-Ergo Version 0.92, Z3 Version 2.11 and Why 2.27
- added List of user-defined predicates and logic functions
- added remarks on the relation of logical values in C and ACSL (see Section 1.2)
- rewrote section on `equal` (Section 3.1) and `mismatch` (Section 3.2)
- used a simpler logical function to count elements in an array (see Section 3.8)
- added `search` algorithm (see Section 3.7)

- added Chapter 4 to unite the maximum/minimum algorithms
- added Chapter 5 for the new `lower_bound`, `upper_bound` and `binary_search` algorithms
- added `swap_values` algorithm
- used `IsEqual` predicate for `swap_ranges` (Section 6.3) and `copy` (Section 6.4)
- added `reverse_copy` and `reverse` algorithms (see Section 6.6)
- added `rotate_copy` algorithm (see Section 6.7)
- added `unique_copy` algorithm (see Section 6.10)
- added Chapter 7 on specification of the data type `Stack`

A.7. New in Version 5.1.0 (May 2010)

- adaption to Frama-C Boron and Why 2.26 releases
- changed from the `-jessie-no-regions` command-line option to using the `pragma SeparationPolicy(`*`value`*`)`

A.8. New in Version 4.2.2 (May 2010)

- changed to latest version of CVC3 2.2
- added additional remarks to our implementation of `find_first_of`
- changed **`size_type`** (**`int`**) to **`integer`** in all specifications
- removed casts in `fill` and `iota`
- renamed `is_valid_range` as `IsValidRange`
- renamed `has_value` as `HasValue`
- renamed predicate `all_equal` as `IsEqual`
- extended timeout to 30 sec.

A.9. New in Version 4.2.1 (April 2010)

- added alternative specification of `remove_copy` algorithm that uses **`ghost`** variables
- added Chapter on heap operations
- added `mismatch` algorithm
- moved algorithms `adjacent_find` and `min_element` from the appendix to Chapter 3

- added typedefs **`size_type`** and **`value_type`** and used them in all algorithms
- renamed `is_valid_int_range` as `is_valid_range`

A.10. New in Version 4.2.0 (January 2010)

- complete rewrite of previous release
- adaption to Frama-C Beryllium 2 release

A.11. Unofficial Release as part of the ES_PASS project (March 2009)

Bibliography

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

[2] WP Plug-in. http://frama-c.com/wp.html.

[3] Frama-C Software Analyzers. http://frama-c.com.

[4] CEA LIST, Laboratory of Applied Research on Software-Intensive Technologies. http://www-list.cea.fr/gb/index_gb.htm.

[5] INRIA-Saclay, French National Institute for Research in Computer Science and Control . http://www.inria.fr/saclay/.

[6] LRI, Laboratory for Computer Science at Université Paris-Sud. http://www.lri.fr/.

[7] Fraunhofer-Institut für Offene Kommunikationssysteme (FOKUS). http://www.fokus.fraunhofer.de.

[8] Virgile Prevosto. ACSL Mini-Tutorial. http://frama-c.com/download/acsl-tutorial.pdf.

[9] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ANSI/ISO C Specification Language, Version 1.5 Frama-C Nitrogen implementation. http://frama-c.com/download/acsl-implementation-Nitrogen-20111001.pdf, Oct 2011.

[10] Standard Template Library Programmer's Guide. http://www.sgi.com/tech/stl, 2010.

[11] Programming languages – C, Committee Draft. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1362.pdf, 2009.

[12] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[13] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–32, Providence, RI, 1967. American Mathematical Society.

[14] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J Comput*, 6(2):323–350, Jun 1977.

[15] Kim Völlinger. *Einsatz des Beweisassistenten Coq zur deduktiven Programmverifikation*. Diplomarbeit, Humboldt Universität zu Berlin, Germany, August 2013.

[16] Richard Fitzpatrick J.L. Heiberg. *Euclid's Elements of Geometry*. http://farside.ph.utexas.edu/euclid.html, Austin/TX, 2008.

[17] David Hilbert. *Grundlagen der Geometrie*. B.G.Teubner, Stuttgart, 1968.

[18] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008.

[19] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Homepage of the Alt-Ergo Theorem Prover. `http://alt-ergo.lri.fr/`.