# ACSL By Example

Towards a Verified C Standard Library

Version 4.2.1
for Frama-C Beryllium 2
April 2010

Jochen Burghardt
Jens Gerlach
Kerstin Hartig
Hans Pohl
Juan Soto

Fraunhofer
FIRST

This body of work was completed within the Device-Soft project[1], which is supported by the Programme Inter Carnot Fraunhofer from BMBF (Grant 01SF0804) and ANR.

[1]See `http://www.first.fraunhofer.de/device_soft_en`

# 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 Jessie plug-in [2] of Frama-C [3].

We have chosen our examples from the C++ Standard Library whose initial version is still known as the *Standard Template Library* (STL).[2] 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 ISO-C functions that work on arrays of type `int`.[3]

In March 2009, while working in the context of the ES_PASS[4] ITEA2 project, we released an initial version of this report. This earlier edition referred to the *Lithium* release of Frama-C. The report at hand is a completely revised and extended edition that refers to the *Beryllium* release of the Frama-C.

We plan to extend this report in later releases by describing additional STL algorithms. 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

devicesoft@first.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.

[2]See `http://www.sgi.com/tech/stl/`
[3]We are not directly using `int` in the source code but rather **`value_type`** which is a `typedef` for `int`.
[4]See `http://www.es-pass.org`

# Contents

# 1. Introduction

Frama-C [3] is a suite of software tools dedicated to the analysis of C source code.

Frama-C development efforts are conducted and coordinated at CEA LIST[4], a laboratory of applied research on software-intensive technologies.

Within Frama-C, the Jessie plug-in [2] enables deductive verification of C programs that have been annotated with the ANSI-C Specification Language (ACSL) [1].

> ACSL can express a wide range of functional properties. The paramount notion in ACSL is the function contract. While many software engineering experts advocate the "function contract mindset" when designing complex software, they generally leave the actual expression of the contract to run-time assertions, or to comments in the source code. ACSL is expressly designed for writing the kind of properties that make up a function contract. [1]

The Jessie plug-in uses Hoare-style weakest precondition computations to formally prove ACSL properties of a program fragment. Internally, Jessie relies on the languages and tools contained in the Why platform [5]. Verification conditions are generated and submitted to external automatic theorem provers or interactive proof assistants.

We at Fraunhofer FIRST [6] 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 Jessie 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 [7] first. The principles presented in this paper are also documented in the ACSL reference document [8].

## 1.1. Acknowledgment

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 partners from the Device-Soft[5] project, Pascal Cuoq, Virgile Prevosto and Benjamin Monate, as well as the Jessie developers, Claude Marché and Yannick Moy.

[5]See `http://www.first.fraunhofer.de/device_soft_en`

## 1.2. Structure of this Document

The functions presented in this document were selected from the C++ Standard Template Library (STL) [9]. The original C++ implementation was stripped from its generic implementation and mapped to C-arrays of type `value_type`.

Section 1.3 is devoted to commonly occurring statements that arise frequently throughout this tutorial, whereas Section 1.4 provides comments on the command-line options available in Frama-C and Jessie. Section 1.5 gives a short overview on the automatic theorem provers used for deductive verification. Chapter 2 provides a short introduction into the Hoare Calculus.

Adhering to the classification used on the STL web-page [9], the examples in this tutorial were grouped as follows:

- *non-mutating algorithms* (discussed in Chapter 3)
- *mutating algorithms* (discussed in Chapter 4)
- *heap operations* (discussed in Chapter 5)

Chapter 6 contains a summary of the results of the automatic theorem provers employed for the deductive verification of the aforementioned examples.

## 1.3. Types, Arrays, Ranges and Valid Indices

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 int size_type;
```

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.

### Arrays and Ranges

Many of the algorithms, described in the following sections, rely on common principles. In this section, we will list and explain their usage in finer detail.

The C Standard describes an array as a “contiguously allocated nonempty set of objects” [10, §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.

## Valid Indices

```
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.

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_range(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_range(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_range(a, 0, n-1)`.

Introducing the following predicate based on the previous definition

```
/*@
  predicate is_valid_range(value_type* p, size_type n) =
           (0 <= n) && \valid_range(p, 0, n-1);
*/
```

we can shorten the specification of the function `example` to

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

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

## 1.4. Remarks on Jessie

Jessie is a plug-in that takes ACSL specified C-programs and performs deductive verification. Frama-C can be run using several Jessie options.

The majority of the examples contained in this document were run with the following Frama-C command-line options:

```
frama-c -jessie -jessie-no-regions filename
```

The `-jessie-no-regions` option means that different pointers can point to the same region of memory. We chose to apply this option because it corresponds to the ISO-C Standard. By default Jessie assumes that different pointers point into different memory regions (see [11, Chapter 5]). Note, however, that some of the mutating algorithms in Chapter 4 require the Jessie default. We discuss the reasons for that in the particular sections on those algorithms.

Another option to choose is the integer model. Jessie knows different integer models:

- `exact`, which means that C-integers are considered to behave as unbounded integers in the mathematical sense, i.e., ignoring the potential for an overflow.
- `bounded`, which does not consider integers in the mathematical sense, but rather machine-dependent integers. Since overflows are detected using this integer model, one should select it, whenever one wishes to prevent overflow.
- `modulo`, which allows for overflow. If an overflow occurs, the value is taken modulo $2^n$, where $n$ is the number of bits of the specific type.

Selection and invocation of a particular integer-model should be considered when generating verification conditions. This can be expressed using the following option:

```
frama-c -jessie -jessie-int-model model filename
```

By default, if no specific model is set, the `bounded` integer model is assumed for the generation of verification conditions. Note: All of the examples in this tutorial used the default integer model.

## 1.5. Automatic Theorem Provers

Table 1.1 shows the five *automatic theorem provers* (ATP) used in conjunction with Jessie.

**Alt-Ergo** is an automatic theorem prover dedicated to program verification developed at *Laboratoire de Recherche en Informatique* (LRI) [12]. Alt-Ergo is freely available, under the terms

| **Automatic Theorem Prover** | **Version** |
|---|---|
| Alt-Ergo | 0.9 |
| Yices | 1.0.23 |
| CVC3 | 1.5 |
| Simplify | 1.5.7 |
| Z3 | 2.1 |

Table 1.1.: Automatic theorem provers used during deductive verification

of the CeCILL-C LICENSE.[6]

**CVC3** is an automatic theorem prover for Satisfiability Modulo Theories (SMT) problems [13]. It was developed at New York University (NYU) and there is “essentially no limit on its use for research or commercial purposes”.[7]

**Simplify** is an automatic theorem prover [14]. The development of Simplify is not continued anymore. Nevertheless, it is still a highly efficient prover. As Chapter 6 demonstrates, Simplify is the only prover that could verify all generated proof obligations from this tutorial.

**Yices** is an efficient SMT solver [15].

**Z3** is a high-performance theorem prover developed at Microsoft Research [16]. A version for non-commercial use is also available.[8]

[6]For Details see `http://alt-ergo.lri.fr/http/CeCILL-C` and `http://en.wikipedia.org/wiki/CeCILL`.
[7]See `http://www.cs.nyu.edu/acsys/cvc3/doc/LICENSE.html`.
[8]See `http://research.microsoft.com/en-us/um/redmond/projects/z3/faq.html`

# 2. The Hoare Calculus

In 1969, C.A.R. Hoare introduced a calculus for formal reasoning about properties of imperative programs [17], which became known as "Hoare Calculus".

The basic notion is

```
//@ P
Q
//@ 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 [8, 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

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

(a)

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

(b)

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

(c)

Figure 2.1.: Example source-code fragments and annotations

Figure 2.1 shows three example source-code fragments and annotations. Their informal meanings are as follows:

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

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

(c) "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. 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. Examples (a) and (b) 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 example (b), 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 example (c).
- We can use the predicate `true` to denote the absence of a properly restricting precondition, as we did in example (c).
- It is not possible to express by pre- and postconditions that a given piece of code will always terminate. Example (c) 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 example (c) 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 2.2 to 2.10 in order to reason about programs. We will comment on them in the following sections.

## 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. This is depicted in Figure 2.2, where "`P` {`x`↦`e`}" denotes the result of substituting each occurrence of `x` in `P` by `e`.

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

Figure 2.2.: The assignment rule

For example,

| | | | |
|---|---|---|---|
| if | `P` | is | `0 <= x <= 5 ==> a[2*  x  ] == 0` , |
| then | `P`$\{$`x`$\mapsto$`x+1`$\}$ | is | `0 <=x+1<= 5 ==> a[2*(x+1)] == 0` . |

Hence, we get Figure 2.3 as an example instance of the assignment rule. Note that parentheses are required in the index expression to get the correct `2*(x+1)` rather than the faulty `2*x+1`.

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

Figure 2.3.: An assignment rule example instance

When first confronted with Hoare's assignment rule, most people are tempted to think of a simpler and more intuitive alternative, shown in Figure 2.4. Figures 2.5.(a)–(c) show some example instances.

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

Figure 2.4.: Simpler, but *faulty* assignment rule

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

(a)

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

(b)

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

(c)

Figure 2.5.: Some example instances of the faulty rule from Figure 2.4

While (a) happens to be ok, (b) and (c) lead to postconditions that are obviously nonsensical formulas. The reason is that in the assignment in (b) the left-hand side variable `x` also appears in the right-hand side expression `e`, while the assignment in (c) just destroys the property from its precondition. Note that the correct example (a) can as well be obtained as an instance of the correct rule from Figure 2.2, 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 2.6, 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.

```
//@ P            //@ R           //@ P
Q         and    S         ⇝     Q ; S
//@ R            //@ T           //@ T
```

Figure 2.6.: The sequence rule

## 2.3. The Implication Rule

The implication rule, shown in Figure 2.7, allows us at any time to weaken a postcondition and to sharpen a precondition. We will provide application examples together with the next rule.

```
//@ P          //@ P′
Q        ⇝     Q              if    P′ ==> P
//@ R          //@ R′         and   R ==> R′
```

Figure 2.7.: The implication rule

## 2.4. The Choice Rule

The choice rule, depicted in Figure 2.8, 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 2.9 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.

```
//@ P && B
Q
//@ S
```

and

```
//@ P && ! B
R
//@ S
```

$\rightsquigarrow$

```
//@ P
if (B) {
    Q
} else {
    R
}
//@ S
```

Figure 2.8.: The choice rule

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

Figure 2.9.: An example application of the choice rule

## 2.5. The Loop Rule

The loop rule, shown in Figure 2.10, 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.

```
//@ P && B
S
//@ P
```

$\rightsquigarrow$

```
//@ P
while (B) {
    S
}
//@ ! B && P
```

Figure 2.10.: 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 emphasise 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 2.11.

```
//@ 0 < n                               given precondition
//@ 0 <= 0 < n                          follows trivially
int i = 0;
//@ 0 <= i < n                          by the assignment rule
while (!done) {
  //@ 0 <= i < n && !done               may be assumed by the loop rule
  a[i] = getchar();
  //@ 0 <= i < n && !done               required property of getchar
  //@ 0 <= i < n                        weakened by the implication rule
  if (i < n-1) i++; else i = 0;
  //@ 0 <= i < n                        as seen above (Figure 2.9)
  process(a,i,&done);
  //@ 0 <= i < n                        required property of process
}
//@ 0 <= i < n                          by the loop rule
```

Figure 2.11.: An abstract ring buffer loop

Figure 2.11 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 2.9 as a sub-part. Note the following issues:

- To reuse the proof from Figure 2.9, we had to drop the conjunct `!done`, since we didn't consider it in Figure 2.9. In general, we may *not* infer

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

  since the code fragment `Q` may just destroy the property `S`. This is obvious for `Q` being the fragment from Figure 2.9, and `S` being e.g. `n != 0`.

  Suppose for a moment that `process` had been implemented in 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 2.9. 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 4.1 on Page 54.

- 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`[9] 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 2.9 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.

[9] 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`.

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 "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 [18] 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. Jessie only supports forward 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 latter ones.

All algorithms work on one-dimensional arrays ("ranges"). With one exception, the `max_seq` function, all algorithms are also well-defined for *empty* ranges. Below are the six example algorithms we will discuss next.

**`equal`** (Section 3.1 on Page 22) compares two ranges element by element.

**`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 33, a predicate is introduced in order to simplify the specification.

**`find_first_of`** (Section 3.5, on Page 36) 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`** which can be used to find equal neighbours in an array (Section 3.6).

**`max_element`** (Section 3.7, on Page 42) returns an index to a maximum element in range. Similar to `find` it also returns the smallest of all possible indices.

**`max_seq`** (Section 3.8, on Page 46) 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 3.9).

**`count`** (Section 3.10, on Page 50) 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[10] to that of an array of type **size_type**. The signature now reads:

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

The function returns 1 if for the two sequences

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

holds. Otherwise, `equal` returns 0.

### 3.1.1. Formal Specification of `equal`

The ACSL specification of `equal` is shown in Listing 3.1.

```
 1  /*@
 2    requires is_valid_range(a, n);
 3    requires is_valid_range(b, n);
 4
 5    assigns \nothing;
 6
 7    behavior all_equal:
 8      assumes \forall size_type i; 0 <= i < n ==> a[i] == b[i];
 9      ensures \result == 1;
10
11    behavior some_not_equal:
12      assumes \exists size_type i; 0 <= i < n && a[i] != b[i];
13      ensures \result == 0;
14
15    complete behaviors all_equal, some_not_equal;
16    disjoint behaviors all_equal, some_not_equal;
17  */
18  bool equal(const value_type* a, size_type n, const value_type* b);
```

Listing 3.1: Formal specification of `equal`

**Lines 2 and 3** in Listing 3.1 formalise the requirements 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 1.3).

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

**Line 5** indicates that `equal`, as a non-mutating algorithm, does not modify any memory location outside its scope (see Page 21).

### Behavior `all_equal`

The behavior `all_equal` applies if an element-wise comparison of the two ranges yields that they are equal see Line 8 in Listing 3.1). In this case the function `equal` is expected to return the value 1 (Line 9).

### Behavior `some_not_equal`

The behavior `some_not_equal` applies if there is at least one valid index `i` where the elements `a[i]` and `b[i]` differ (Line 12). In this case the function `equal` is expected to return the value 0 (Line 13).

### Completeness and Disjointness of Behaviors

The negation of the formula

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

in behavior `all_equal` is just the formula

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

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

Line 15 in Listing 3.1 expresses the fact that for all ranges `a` and `b` that satisfy the preconditions of the contract (Lines 2 and 3) *at least one* of the behaviors `all_equal` and `some_not_equal` applies.

Line 16 in Listing 3.1 formalises the fact that for all ranges `a` and `b` that satisfy the preconditions of the contract (Lines 2 and 3) *at most one* of the behaviors `all_equal` and `some_not_equal` applies.

### 3.1.2. Implementation of `equal`

Listing 3.2 shows one way to implement the function `equal`. In our description, we concentrate on the *loop annotations* in Lines 4 to 6.

```
 1  bool equal(const value_type* a, size_type n, const value_type* b)
 2  {
 3    /*@
 4      loop invariant 0 <= i <= n;
 5      loop   variant n-i;
 6      loop invariant \forall size_type k; 0 <= k < i ==> a[k] == b[k];
 7    */
 8    for (size_type i = 0; i < n; i++)
 9      if (a[i] != b[i])
10        return 0;
11
12    return 1;
13  }
```

Listing 3.2: Implementation of `equal`

**Line 4** This loop *invariant* is needed to prove that all accesses to `a` and `b` occur with valid indices.

Note: Although the termination condition in the for loop in Line 8 states that `i` must not be equal to `n`, the loop invariant must read `i <= n`. This is due to the fact that `i` may still increase, especially as the last step before the termination of the loop.

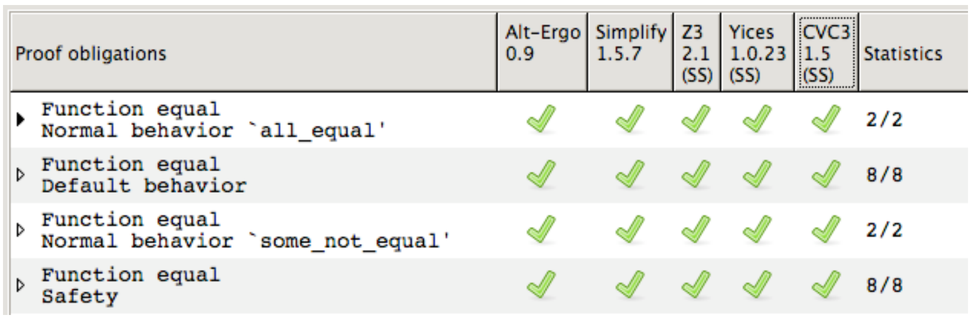
**Line 5** This loop *variant* is needed by newer versions of Jessie[11] to generate correct verification conditions for the termination of the `for`-loop that starts in Line 8.

**Line 6** This loop *invariant* is needed to prove that for each iteration all elements of `a` and `b` up to that iteration step are equal.

[11]As of Version 2.22 of Why and Jessie.

## 3.1.3. Formal Verification of `equal`

Figure 3.1 shows a screen-shot depicting the results of the formal verification of `equal` with five different automatic theorem provers. Each prover was able to prove all verification conditions generated by Jessie.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| ▸ Function equal<br>Normal behavior `all_equal' | ✓ | ✓ | ✓ | ✓ | ✓ | 2/2 |
| ▹ Function equal<br>Default behavior | ✓ | ✓ | ✓ | ✓ | ✓ | 8/8 |
| ▹ Function equal<br>Normal behavior `some_not_equal' | ✓ | ✓ | ✓ | ✓ | ✓ | 2/2 |
| ▹ Function equal<br>Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 8/8 |

Figure 3.1.: Results of the formal verification of `equal`

## 3.2. The `mismatch` Algorithm and `equal` again

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

```
int 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]

The close relation of `mismatch` and `equal` will lead to very similar specifications. The motto *don't repeat yourself* is not just good programming practice.[13] It is also true for concise and easy to understand specifications. We will therefore introduce specification elements that we can apply to both `mismatch` and another specification and implementation of `equal`.

### 3.2.1. Formal Specification of `equal` and `mismatch`

Our specification of mismatch starts with introducing the predicate `all_equal`.

```
/*@
  predicate
    all_equal{La,Lb}(value_type* a, size_type n, value_type* b) =
      \forall value_type i; 0 <= i < n ==>
        \at(a[i], La) == \at(b[i], Lb);
*/
```

Listing 3.3: The predicate `all_equal`

This predicate formalizes the fact that the elements of the range `a` observed at the label `La` are all equal to those of range `b` observed at `Lb`. We chose this most general form to extend the area of applicability of `all_equal` as far as possible. In our particular examples `equal` and `mismatch`, we will always provide identical labels as `La` and `Lb`.

Using this predicate we can reformulate the specification of `equal` from Listing 3.1 as shown in Listing 3.4.

Note that `all_equal` is both the name of the predicate and of a behavior. This is no conflict, as both belong to different name spaces.

We use the `all_equal` predicate also for the formal specification of `mismatch` that is shown in Listing 3.5. Note in particular the use of `all_equal` in Line 15 of Listing 3.5 in order to express that `mismatch` returns the smallest index where the two arrays differ. Moreover observe that completeness and disjointness 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.

[12]See also `http://www.sgi.com/tech/stl/mismatch.html`.
[13]Compare `http://en.wikipedia.org/wiki/Don't_repeat_yourself`.

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

  assigns \nothing;

  behavior all_equal:
    assumes all_equal{Here,Here}(a, n, b);
    ensures \result == 1;

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

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

Listing 3.4: Another formal specification of `equal`

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

  assigns \nothing;

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

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

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

Listing 3.5: Formal specification of `mismatch`

### 3.2.2. Implementation of mismatch and equal

Listing 3.6 shows an implementation of mismatch that we have enriched with some loop annotations to support the deductive verification of the specification given in Listing 3.5.

```
 1  size_type mismatch(const value_type* a, size_type n,
 2                     const value_type* b)
 3  {
 4    /*@
 5      loop invariant 0 <= i <= n;
 6      loop   variant n-i;
 7      loop invariant all_equal{Here,Here}(a, i, b);
 8    */
 9    for (size_type i = 0; i < n; i++)
10      if (a[i] != b[i])
11        return i;
12
13    return n;
14  }
```

Listing 3.6: Implementation of mismatch

We use the predicate all_equal in Line 7 of Listing 3.6 in order to express that the 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.

Listing 3.7 shows an implementation of the equal algorithm by a simple call of mismatch.[14]

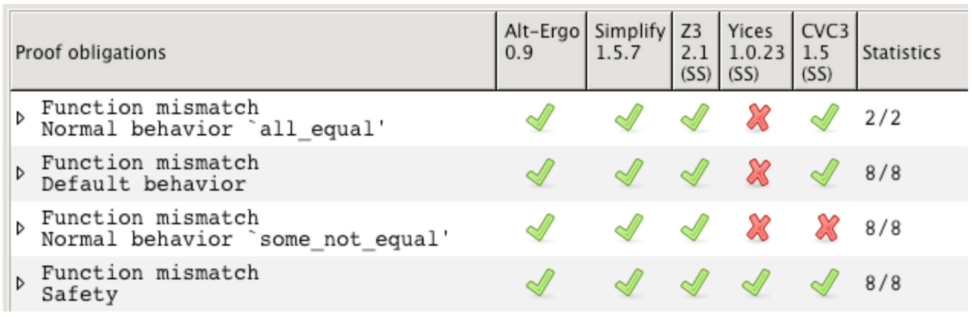
```
int equal(const value_type* a, size_type n, const value_type* b)
{
  return mismatch(a, n, b) == n ? 1 : 0;
}
```

Listing 3.7: Implementation of equal with mismatch

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

### 3.2.3. Formal Verification of `mismatch` and `equal`

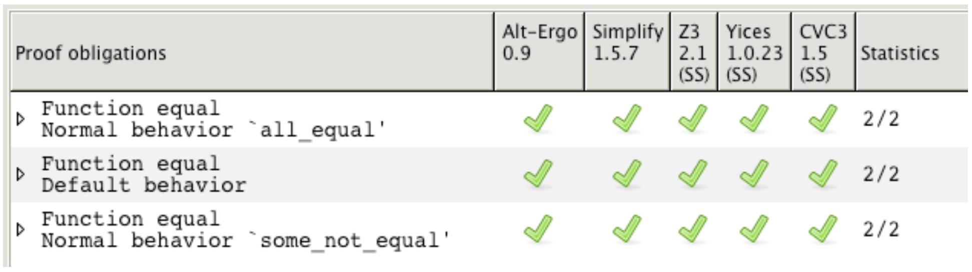
Figure 3.2 shows the results of the formal verification of `mismatch`. Three out of five automatic theorem provers were able to prove all verification conditions generated by Jessie.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| Function mismatch<br>Normal behavior `all_equal' | ✓ | ✓ | ✓ | ✗ | ✓ | 2/2 |
| Function mismatch<br>Default behavior | ✓ | ✓ | ✓ | ✗ | ✓ | 8/8 |
| Function mismatch<br>Normal behavior `some_not_equal' | ✓ | ✓ | ✓ | ✗ | ✗ | 8/8 |
| Function mismatch<br>Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 8/8 |

Figure 3.2.: Results of the formal verification of `mismatch`

Figure 3.3 shows the results of the formal verification of `equal` with `mismatch`. Each automatic theorem provers was able to prove all verification conditions generated by Jessie.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| Function equal<br>Normal behavior `all_equal' | ✓ | ✓ | ✓ | ✓ | ✓ | 2/2 |
| Function equal<br>Default behavior | ✓ | ✓ | ✓ | ✓ | ✓ | 2/2 |
| Function equal<br>Normal behavior `some_not_equal' | ✓ | ✓ | ✓ | ✓ | ✓ | 2/2 |

Figure 3.3.: Results of the formal verification of `equal` with `mismatch`

## 3.3. The find Algorithm

The find algorithm in the C++ standard library implements *sequential search* for general sequences.[15] 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 v);
```

The function find returns the least *valid* index i of a where the condition

```
a[i] == v
```

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 3.8. We discuss the specification now line by line.

```
 1 /*@
 2   requires  is_valid_range(a, n);
 3
 4   assigns  \nothing;
 5
 6   behavior some:
 7     assumes  \exists size_type i; 0 <= i < n && a[i] == val;
 8     ensures  0 <= \result < n;
 9     ensures  a[\result] == val;
10     ensures  \forall size_type i; 0 <= i < \result ==> a[i] != val;
11
12   behavior none:
13     assumes  \forall size_type i; 0 <= i < n ==> a[i] != val;
14     ensures  \result == n;
15
16   complete behaviors some, none;
17   disjoint behaviors some, none;
18 */
19 size_type find(const value_type* a, size_type n, value_type val);
```

Listing 3.8: Formal specification of find

**Line 2** indicates that n is non-negative and that the pointer a points to *n* contiguously allocated objects of type **value_type** (see Section 1.3).

**Line 4** indicates that find (as a non-mutating algorithm), does not modify any memory location outside its scope (see Page 21).

We have subdivided the specification of find into two behaviors (named some and none), which we describe in the following subsections.

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

### Behaviour `some`

The behavior `some` checks that the sought-after value is contained in the array. We express this condition by the formula in Line 7 in Listing 3.8.

**Line 8** expresses that if the assumptions of the behavior are satisfied then `find` will return a valid index.

**Line 9** indicates that for the returned (valid) index `i`, `a[i] == v` holds.

**Line 10** is important because it expresses that `find` returns the smallest index `i` for which `a[i] == v` holds.

### Behaviour `none`

The behavior `none` covers the case that the sought-after value is *not* contained in the array (see Line 13 in Listing 3.8). In this case, `find` must return the length `n` of the range `a` (see Line 14).

### Completeness and Disjointness of Behaviors

The formula in the assumes clause (Line 7) of the behavior `some` is the negation of the assumes clause of the behavior `none`. Therefore, we can express in Lines 16 and 17 that these two behaviors are *complete* and *disjoint*.

### 3.3.2. Implementation of `find`

Listing 3.9 shows a straightforward implementation of `find`. The only noteworthy elements of this implementation are the three *loop annotations* in Lines 4 to 6.

```
 1  size_type find(const value_type* a, size_type n, value_type val)
 2  {
 3    /*@
 4      loop invariant 0 <= i <= n;
 5      loop   variant n-i;
 6      loop invariant \forall size_type k; 0 <= k < i ==> a[k] != val;
 7     */
 8    for (size_type i = 0; i < n; i++)
 9      if (a[i] == val)
10        return i;
11
12    return n;
13  }
```

Listing 3.9: Implementation of `find`

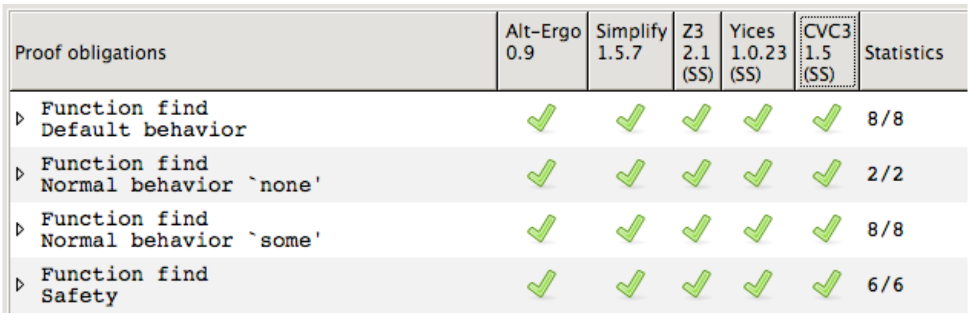
**Line 4** This loop *invariant* is needed to prove that accesses to `a` only occur with valid indices.

**Line 5** This loop *variant* is needed by newer versions of Jessie[16] to generate correct verification conditions for the termination of the loop.

**Line 6** This loop *invariant* is needed for the proof of the postconditions in Lines 8–10 of the behavior `some` (see Listing 3.8). It expresses that for each iteration the sought-after value is not yet found up to that iteration step.

### 3.3.3. Formal Verification of `find`

Figure 3.4 shows a screen-shot depicting the results of the formal verification of `find` with five different automatic theorem provers. Each prover was able to prove all verification conditions generated by Jessie.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| Function find Default behavior | ✓ | ✓ | ✓ | ✓ | ✓ | 8/8 |
| Function find Normal behavior `none' | ✓ | ✓ | ✓ | ✓ | ✓ | 2/2 |
| Function find Normal behavior `some' | ✓ | ✓ | ✓ | ✓ | ✓ | 8/8 |
| Function find Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 6/6 |

Figure 3.4.: Results of the formal verification of `find`

[16]As of Version 2.22 of Why and Jessie.

## 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 amount of closely related formulas. We have in Listings 3.8 and 3.9 the following formulas

| |
|---|
| `\exists size_type i; 0 <= i < n && a[i] == val;` |
| `\forall size_type i; 0 <= i < \result ==> a[i] != val;` |
| `\forall size_type i; 0 <= i < n ==> a[i] != val;` |
| `\forall size_type 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 `have_value` (see Listing 3.10), which describes the situation that there is a valid index `i` such that

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

holds.

```
1 /*@
2   predicate
3     have_value{A}(value_type* a, size_type n, value_type val) =
4       \exists size_type i; 0 <= i < n && a[i] == val;
5 */
```

Listing 3.10: The predicate `have_value`

Note the label `A` in Line 3 in Listing 3.10. According to the ACSL specification it is used in order to safely describe dependencies of a *hybrid* predicate such as `have_value` on one or more program points, see [8, § 2.6.9].

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 3.11 and 3.12.

### 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.

```
 1  /*@
 2    requires  is_valid_range(a, n);
 3
 4    assigns  \nothing;
 5
 6    behavior some:
 7      assumes  have_value(a, n, val);
 8      ensures  0 <= \result < n;
 9      ensures  a[\result] == val;
10      ensures  !have_value(a, \result, val);
11
12    behavior none:
13      assumes  !have_value(a, n, val);
14      ensures  \result == n;
15
16    complete behaviors some, none;
17    disjoint behaviors some, none;
18  */
19  size_type find(const value_type* a, size_type n, value_type val);
```

Listing 3.11: Formal specification of `find` using the `have_value` predicate

In particular, it can be seen immediately that the conditions in the assume clauses of the two behaviors `some` and `none` are mutually exclusive since one is the negation of the other. Moreover, the requirement that `find` returns the smallest index can also be expressed using the `have_value` predicate, as depicted in Line 10 in Listing 3.11.

### 3.4.2. Implementation of `find`

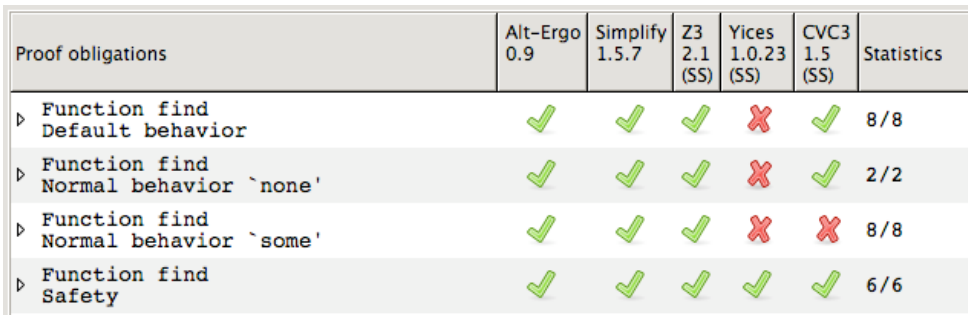
The predicate `have_value` is also used in the loop annotation inside the implementation of `find` (see Line 6 in Listing 3.12).

```
 1  size_type find(const value_type* a, size_type n, value_type val)
 2  {
 3    /*@
 4      loop invariant 0 <= i <= n;
 5      loop   variant n-i;
 6      loop invariant !have_value(a, i, val);
 7     */
 8    for (size_type i = 0; i < n; i++)
 9      if (a[i] == val)
10        return i;
11
12    return n;
13  }
```

Listing 3.12: Implementation of `find` with loop annotations based on `have_value`

### 3.4.3. Formal Verification of `find`

One drawback of introducing the `have_value` predicate is that some provers cannot prove all verification conditions (see Figure 3.4 and Figure 3.5).



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| Function find Default behavior | ✓ | ✓ | ✓ | ✗ | ✓ | 8/8 |
| Function find Normal behavior `none' | ✓ | ✓ | ✓ | ✗ | ✓ | 2/2 |
| Function find Normal behavior `some' | ✓ | ✓ | ✓ | ✗ | ✗ | 8/8 |
| Function find Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 6/6 |

Figure 3.5.: Results of the formal verification of `find` using the `have_value` predicate

## 3.5. The find_first_of Algorithm

The find_first_of algorithm[17] 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 have_first_of that formalises 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 have_first_of as follows:

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

However, we have chosen to reuse the predicate have_value (Listing 3.10) to define the have_first_of predicate. The result is shown in Listing 3.13.

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

Listing 3.13: The predicate have_first_of

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

Both the predicates have_first_of and have_value occur in the formal specification of find_first_of (see Listing 3.14). Note how similar the specification of find_first_of becomes to that of find (Listing 3.11) when using these predicates.

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

  assigns \nothing;

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

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

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

Listing 3.14: Formal specification of find_first_of

### 3.5.2. Implementation of **`find_first_of`**

Our implementation of `find_first_of` is shown in Listing 3.15.

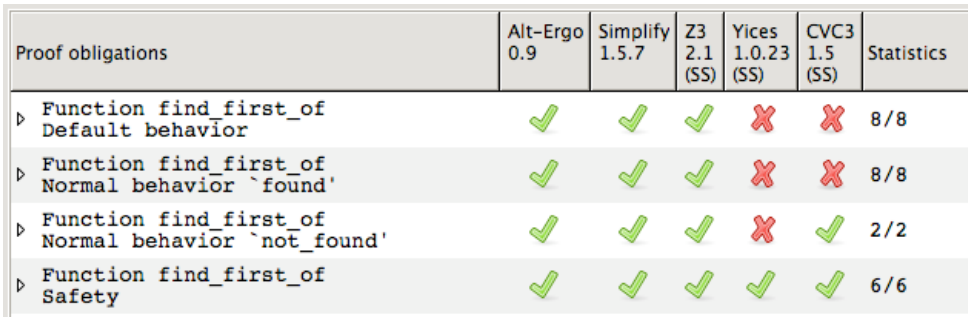
```
 1  size_type find_first_of (const value_type* a, size_type m,
 2                           const value_type* b, size_type n)
 3  {
 4    /*@
 5      loop invariant 0 <= i <= m;
 6      loop   variant m-i;
 7      loop invariant !have_first_of(a, i, b, n);
 8    */
 9    for (size_type i = 0; i < m; i++)
10      if (find(b, n, a[i]) < n)
11        return i;
12
13    return m;
14  }
```

Listing 3.15: Implementation of `find_first_of`

Note the call of the `find` function in Line 10 above. Besides, leading to a more concise implementation, it also enables us to save time in writing additional loop annotations.

### 3.5.3. Formal Verification of `find_first_of`

Figure 3.6 shows the results of the formal verification of `find_first_of` with five different automatic theorem provers. Alt-Ergo, Simplify and Z3 were able to prove each of the generated verification conditions.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| Function find_first_of Default behavior | ✓ | ✓ | ✓ | ✗ | ✗ | 8/8 |
| Function find_first_of Normal behavior `found' | ✓ | ✓ | ✓ | ✗ | ✗ | 8/8 |
| Function find_first_of Normal behavior `not_found' | ✓ | ✓ | ✓ | ✗ | ✓ | 2/2 |
| Function find_first_of Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 6/6 |

Figure 3.6.: Results of the formal verification of `find_first_of`

## 3.6. The `adjacent_find` Algorithm

The `adjacent_find` algorithm[18]

```
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 `have_equal_neighbours` (see Listing 3.16) that captures the essence of finding two adjacent indices at which the array holds equal values.

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

Listing 3.16: The predicate `have_equal_neighbours`

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

  assigns \nothing;

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

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

  complete behaviors some, none;
  disjoint behaviors some, none;
*/
size_type adjacent_find(const value_type* a, size_type n);
```

Listing 3.17: Formal specification of `adjacent_find`

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

We use the predicate `have_equal_neighbours` in Lines 7, 10 and 13 of the formal specification of `adjacent_find` (see Listing 3.17).

## 3.6.2. Implementation of `adjacent_find`

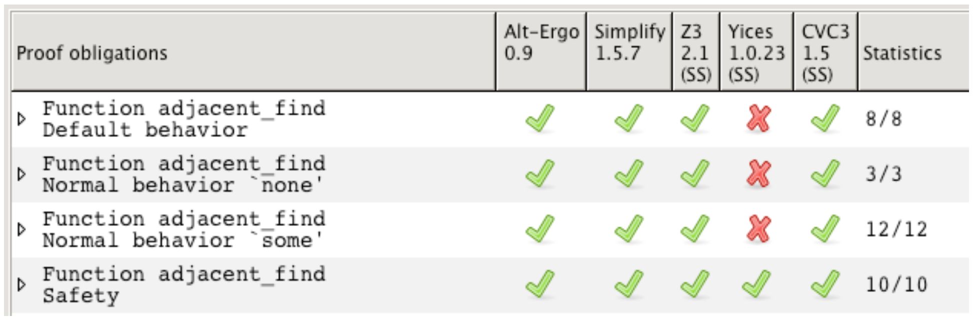
The implementation of `adjacent_find`, including loop (in)variants is shown in Listing 3.18. Please note the use of the predicate `have_equal_neighbours` in loop invariant in Line 8.

```
 1  size_type adjacent_find(const value_type* a, size_type n)
 2  {
 3    if (0 == n) return n;
 4
 5    /*@
 6      loop invariant 0 <= i < n;
 7      loop   variant n-i;
 8      loop invariant !have_equal_neighbours(a, (size_type)(i+1));
 9    */
10    for (size_type i = 0; i < n-1; i++)
11      if (a[i] == a[i+1])
12        return i;
13
14    return n;
15  }
```

Listing 3.18: Implementation of `adjacent_find`

## 3.6.3. Formal Verification of `adjacent_find`

The screen-shot in Figure 3.7 shows the results of the formal verification of `adjacent_find` with five different automatic theorem provers. Except for Yices, all provers were able to prove all verification conditions.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| Function adjacent_find Default behavior | ✓ | ✓ | ✓ | ✗ | ✓ | 8/8 |
| Function adjacent_find Normal behavior `none' | ✓ | ✓ | ✓ | ✗ | ✓ | 3/3 |
| Function adjacent_find Normal behavior `some' | ✓ | ✓ | ✓ | ✗ | ✓ | 12/12 |
| Function adjacent_find Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 10/10 |

Figure 3.7.: Results of the formal verification of `adjacent_find`

## 3.7. The `max_element` Algorithm

The `max_element` algorithm in the C++ Standard Template Library[19] 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 smallest 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`.

### 3.7.1. Formal Specification of `max_element`

A formal specification of `max_element` in ACSL is shown in Listing 3.19. Now we shall discuss the specification line by line.

```
 1 /*@
 2   requires  is_valid_range(a, n);
 3 
 4   assigns \nothing;
 5 
 6   behavior empty:
 7     assumes n == 0;
 8     ensures \result == 0;
 9 
10   behavior not_empty:
11     assumes 0 < n;
12     ensures 0 <= \result < n;
13 
14     ensures \forall size_type i;
15         0 <= i < n ==> a[i] <= a[\result];
16 
17     ensures \forall size_type i;
18         0 <= i < \result ==> a[i] < a[\result];
19 
20   complete behaviors empty, not_empty;
21   disjoint behaviors empty, not_empty;
22 */
23 size_type max_element(const value_type* a, size_type n);
```

Listing 3.19: Formal specification of `max_element`

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

**Line 2** indicates that `n` is non-negative and that the pointer `a` points to `n` contiguously allocated objects of type **`value_type`** (see Section 1.3).

**Line 4** indicates that `max_element` (as a non-mutating algorithm), does not modify any memory location outside its scope (see Page 21).

We have subdivided the specification of `max_element` into two behaviors (named `empty` and `not_empty`), which we describe in the following subsections.

### Behaviour `empty`

The behavior `empty` checks that the range contains no elements.

**Line 7** formalises the condition under which the behavior `empty` is active. In our case, that is, `n == 0`.

**Line 8** expresses that under the assumptions of the behavior `empty` the function `max_element` will return the value 0.

### Behaviour `not_empty`

The behavior `not_empty` checks that the range has a positive length.

**Line 11** formalises the condition under which the behavior `not_empty` is active. In our case, that is, `0 < n`.

**Line 12** expresses that if the assumption of the behavior is satisfied `max_element` will return a valid index `k`, i.e., `0 <= k < n`, must hold.

**Lines 13–24** indicates that the returned valid index `k` refers to a maximum value of the array.

**Lines 17–18** expresses that `k` is indeed the *first* occurrence of a maximum value in the array.

### Completeness and Disjointness of Behaviors

The formula in the assumes clause (Line 7) of the behavior `empty` is the negation of the assumes clause of the behavior `not_empty`. Therefore, we can express in Lines 20 and 21 that these two behaviors are *complete* and *disjoint*.

### 3.7.2. Implementation of `max_element`

Listing 3.20 shows an implementation of `max_element`. In our description, we concentrate on the *loop annotations* in the Lines 7 to 12.

```
 1  size_type max_element(const value_type* a, size_type n)
 2  {
 3    if (n == 0) return 0;
 4
 5    size_type max = 0;
 6    /*@
 7      loop invariant 0 <= i <= n;
 8      loop   variant n-i;
 9
10      loop invariant 0 <= max <  n;
11      loop invariant \forall size_type k;
12          0 <= k < i    ==> a[k] <= a[max];
13
14      loop invariant \forall size_type k;
15          0 <= k < max ==> a[k] <  a[max];
16    */
17    for (size_type i = 0; i < n; i++)
18      if (a[max] < a[i])
19        max = i;
20
21    return max;
22  }
```

Listing 3.20: Implementation of `max_element`

**Line 7** This loop *invariant* expresses the fact that inside the loop the index `i` will be greater than or equal to 0 and less than or equal to `n`. Note that at the end of the loop the index `i` indeed takes the value `n`. This loop annotation is used to prove the preservation of the loop invariant in Line 10 in Listing 3.20, and to prove that all accesses to `a` occur only with *valid* indices.

**Line 8** This loop *variant* is needed by newer versions of Jessie[20] to generate correct verification conditions for the termination of the `for`-loop.

**Line 10** This loop *invariant* is needed to prove of the postcondition in Line 12 of the behavior `not_empty` in Listing 3.19, and in order to prove that all pointer accesses occur with valid indices.

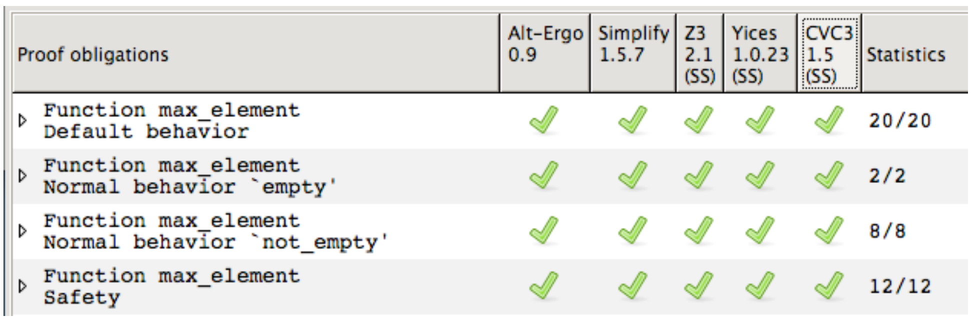
**Line 11–12** This loop *invariant* is used to prove the postcondition in Line 14 of the behavior `not_empty` in Listing 3.19, and to prove that the loop invariant in Line 12 of Listing 3.20 is preserved.

**Line 14–15** This loop *invariant* is used to prove the postcondition in Line 17 of the behavior `not_empty` (see Listing 3.19).

[20]As of Version 2.22 of Why and Jessie.

## 3.7.3. Formal Verification of `max_element`

The screen-shot in Figure 3.8 shows the results of the formal verification of `max_element` with five different automatic theorem provers. Each prover was able to prove all verification conditions generated by Jessie.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| Function max_element Default behavior | ✓ | ✓ | ✓ | ✓ | ✓ | 20/20 |
| Function max_element Normal behavior `empty' | ✓ | ✓ | ✓ | ✓ | ✓ | 2/2 |
| Function max_element Normal behavior `not_empty' | ✓ | ✓ | ✓ | ✓ | ✓ | 8/8 |
| Function max_element Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 12/12 |

Figure 3.8.: Results of the formal verification of `max_element`

.

## 3.8. The max_seq Algorithm

In this section we consider the function max_seq (see Chapter 3, [7]) that is very similar to the max_element function of Section 3.7. 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 3.22). Moreover, using only the formal specification of max_element in Listing 3.19 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 proved to implement the contract in Listing 3.19 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.

### 3.8.1. Formal Specification of max_seq

A formal specification of max_seq in ACSL is shown in Listing 3.21.

```
1  /*@
2    requires n > 0;
3    requires \valid(p+ (0..n-1));
4
5    assigns  \nothing;
6
7    ensures  \forall size_type i; 0 <= i <= n-1 ==> \result >= p[i];
8    ensures  \exists size_type e; 0 <= e <= n-1 &&  \result == p[e];
9  */
10 value_type max_seq(const value_type* p, size_type n) ;
```

Listing 3.21: Formal specification of max_seq

**Lines 2 and 3** express that max_seq requires a valid and, in particular, *non-empty* range as input.

**Line 5** indicates again that max_seq is a non-mutating algorithm.

**Lines 7 and 8** formalise that max_seq indeed returns the maximum value of the range.

### 3.8.2. Implementation of **max_seq**

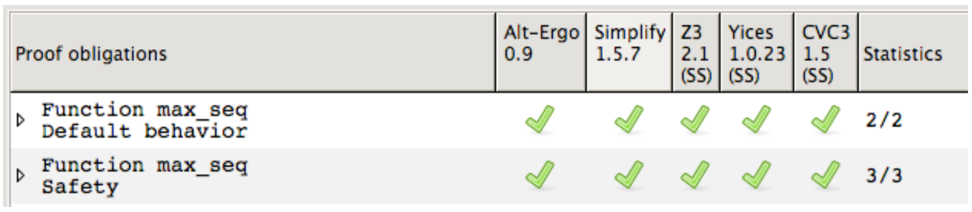
Listing 3.22 shows the trivial implementation of max_seq using max_element. Since max_seq requires a non-empty range the call of max_element returns an index to a maximum value in the range.[21]

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

Listing 3.22: Implementation of max_seq

### 3.8.3. Formal Verification of **max_seq**

The screen-shot in Figure 3.9 shows the results of the formal verification of max_seq with five different automatic theorem provers. Each prover was able to prove all verification conditions generated by Jessie.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| Function max_seq Default behavior | ✓ | ✓ | ✓ | ✓ | ✓ | 2/2 |
| Function max_seq Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 3/3 |

Figure 3.9.: Results of the formal verification of max_seq

[21]The fact that max_element returns the smallest index is of no importance in this context.

## 3.9. The min_element Algorithm

The min_element algorithm in the C++ Standard Template Library[22] searches the minimum in 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 smallest 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.

### 3.9.1. Formal Specification of min_element

The ACSL specification of min_element is shown in Listing 3.23. Given the great similarity to the max_element specification (see Section 3.7) we did not provide a detailed explanation here.

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

  assigns \nothing;

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

  behavior not_empty:
    assumes 0 < n;
    ensures 0 <= \result < n;

    ensures \forall size_type i;
      0 <= i < n  ==>  a[\result] <= a[i];

    ensures \forall size_type i;
      0 <= i < \result ==> a[\result] < a[i];
*/
size_type min_element(const value_type* a, size_type n);
```

Listing 3.23: Formal specification of min_element

### 3.9.2. Implementation of min_element

Listing 3.24 shows the implementation of min_element. Given the great similarity with respect to the max_element implementation (see Section 3.7) we did not provide a detailed explanation

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

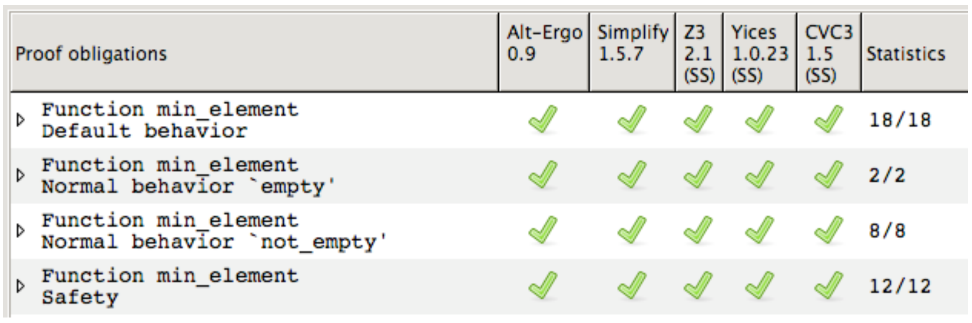
here.

```
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   variant n-i;

    loop invariant \forall size_type k;
      0 <= k < i   ==> a[min] <= a[k];

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

  return min;
}
```

Listing 3.24: Implementation of min_element

### 3.9.3. Formal Verification of min_element

The screen-shot in Figure 3.10 shows the results of the formal verification of min_element with five different automatic theorem provers. Each prover was able to prove all verification conditions generated by Jessie.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| Function min_element Default behavior | ✓ | ✓ | ✓ | ✓ | ✓ | 18/18 |
| Function min_element Normal behavior `empty' | ✓ | ✓ | ✓ | ✓ | ✓ | 2/2 |
| Function min_element Normal behavior `not_empty' | ✓ | ✓ | ✓ | ✓ | ✓ | 8/8 |
| Function min_element Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 12/12 |

Figure 3.10.: Results of the formal verification of min_element

## 3.10. 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[23] to that of arrays of type **`value_type`**. The signature now reads:

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

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

### 3.10.1. Formal Specification of `count`

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

```
1 /*@
2   requires is_valid_range(a, n);
3
4   assigns \nothing;
5
6   ensures \result == counting(a, n, val);
7 */
8 size_type count(const value_type* a, size_type n, value_type val);
```

Listing 3.25: Formal specification of `count`

This specification is fairly short because it employs the *logic function* `counting` (see Line 6 of Listing 3.25). The considerably longer definition is given in Listing 3.26. This definition of `counting` is derived from the *logic function* `nb_occ` of the ACSL specification [8].

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

```
 1  /*@
 2    axiomatic counting_axioms
 3    {
 4      logic integer counting{L}(value_type* a, integer n,
 5                                value_type val) reads a[0..n-1];
 6
 7      axiom counting_empty{L}:
 8        \forall value_type* a, integer n, value_type val; n <= 0 ==>
 9           counting(a, n, val) == 0;
10
11      axiom counting_hit{L}:
12        \forall value_type* a, integer n, value_type val; n >= 0 && a[n]
              == val ==>
13           counting(a, n+1, val) == counting(a, n, val) + 1;
14
15      axiom counting_miss{L}:
16        \forall value_type* a, integer n, value_type val; n >= 0 && a[n]
              != val ==>
17           counting(a, n+1, val) == counting(a, n, val);
18     }
19  */
```

Listing 3.26: Axiomatic definition of count

**Line 2** The ACSL keyword `axiomatic` is used to gather the logic function `counting` and its defining axioms.

**Lines 4 and 5** The logic function `counting` determines the number of occurrences of a value `val` in an array of type **`value_type`** that has the length `n`. Note that the length of the array, `n`, and the return value for `counting` are both of type `integer`. The reads clause in Line 5 specifies the set of memory locations which `counting` depends on (see [8, §2.6.10] for more details).

**Lines 7–9** The axiom `counting_empty` covers the base case, i.e., when the array is empty. In this case `counting` yields 0.

**Lines 11–13** The axiom `counting_hit` detects the number of occurrences of `val` in the array `a[]`.

**Lines 15–17** The axiom `counting_miss` detects when there are no occurrences of `val` in the array `a[]`.

### 3.10.2. Implementation of count

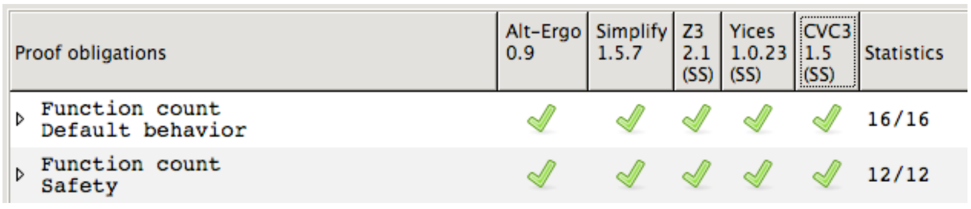
Listing 3.27 shows a possible implementation of count and the corresponding loop (in)variants. Note the reference to counting in the loop invariant in Line 8. The loop invariant in Line 7 is necessary to prove that the local variable cnt does not exceed the range of **size_type**.

```
 1  size_type count(const value_type* a, size_type n, value_type val)
 2  {
 3    size_type cnt = 0;
 4    /*@
 5      loop invariant 0 <= i <= n;
 6      loop   variant n-i;
 7      loop invariant 0 <= cnt <= i;
 8      loop invariant cnt == counting(a, i, val);
 9    */
10    for (size_type i = 0; i < n; i++)
11      if (a[i] == val)
12        cnt++;
13
14    return cnt;
15  }
```

Listing 3.27: Implementation of count

### 3.10.3. Formal Verification of count

The screen-shot in Figure 3.11 shows the results of the formal verification of count with five different automatic theorem provers. Each prover was able to prove all verification conditions generated by Jessie.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| Function count Default behavior | ✓ | ✓ | ✓ | ✓ | ✓ | 16/16 |
| Function count Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 12/12 |

Figure 3.11.: Results of the formal verification of count

.

# 4. 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 [8, Subsection 2.3.2]. Below are the seven example algorithms we will discuss next.

**`swap`** (Section 4.1 on Page 54) exchanges the values two pointers reference.

**`swap_ranges`** (Section 4.2 on Page 56) 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`..

**`fill`** (Section 4.3 on Page 60) initializes each element of an array by a given fixed value.

**`copy`** (Section 4.4 on Page 62) copies a source array to a destination array.

**`replace_copy`** (Section 4.5 on Page 64) 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 4.6 on Page 68) copies a source array to a destination array, but omits each occurrence of a value. In Section 4.7 we discuss an alternative—and in our opinion—better specification of `remove_copy`.

**`iota`** (Section 4.8 on Page 78) writes consecutive integers into an array. Here we have do deal with possible integer overflows when `iota` is executed.

## 4.1. The swap Algorithm

The `swap` algorithm[24] in the C++ STL exchanges the contents of two variables. Similarly, the `iter_swap` algorithm[25] 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);
```

### 4.1.1. Formal Specification of swap

The ACSL specification for the `swap` function is shown in Listing 4.1.

```
 1
 2 /*@
 3   requires \valid(p);
 4   requires \valid(q);
 5
 6   assigns *p;
 7   assigns *q;
 8
 9   ensures *p == \old(*q);
10   ensures *q == \old(*p);
11 */
12 void swap(value_type* p, value_type* q);
```

Listing 4.1: Formal specification of `swap`

**Lines 2–3** states that both argument pointers to the `swap` function must be dereferenceable.

**Lines 5–6** formalises 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 [8, Subsection 2.3.2]. To forbid a function to do any modifications outside its scope, a clause

```
assigns \nothing;
```

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

**Lines 8–9** Upon termination of `swap`, the entries must be mutually exchanged. 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.

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

### 4.1.2. Implementation of swap

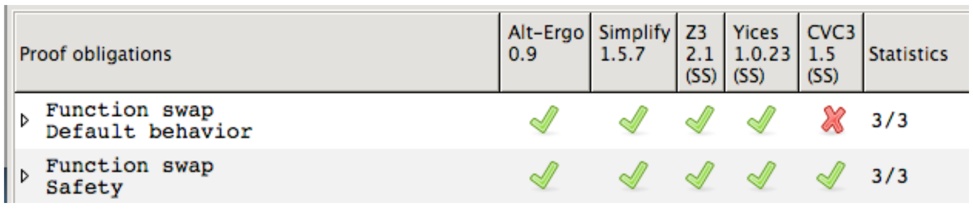
Listing 4.2 shows the usual straight-forward implementation of swap. No interspersed ACSL is needed to get it verified by Frama-C.

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

Listing 4.2: Implementation of swap

### 4.1.3. Formal Verification of swap

In Figure 4.1, the results of the formal verification of swap are shown with five different automatic theorem provers. Each prover, except for CVC3, was able to prove all verification conditions generated by Jessie.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| Function swap Default behavior | ✓ | ✓ | ✓ | ✓ | ✗ | 3/3 |
| Function swap Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 3/3 |

Figure 4.1.: Results of the formal verification of swap

## 4.2. The swap_ranges Algorithm

The `swap_ranges` algorithm[26] 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`.

### 4.2.1. Formal Specification of swap_ranges

The ACSL specification for the `swap_ranges` is shown in Listing 4.3.

```
 1 /*@
 2   requires is_valid_range(a, n);
 3   requires is_valid_range(b, n);
 4   //requires \separated(a+(0..n-1), b+(0..n-1));
 5
 6   assigns a[0..n-1];
 7   assigns b[0..n-1];
 8
 9   ensures \forall size_type k; 0 <= k < n ==> a[k] == \old(b[k]);
10   ensures \forall size_type k; 0 <= k < n ==> b[k] == \old(a[k]);
11 */
12 void swap_ranges(value_type* a, size_type n, value_type* b);
```

Listing 4.3: Formal specification of the `swap_ranges` function

**Lines 2–3** We refer to Section 1.3 for a discussion of how valid ranges are specified.

**Line 4** This line is commented out not only for the C compiler (by the `/*@` in Line 1), but also for Frama-C.

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

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

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

where `b` is different from the old `a`.

The `separated`-clause in Line 4 was intended to tell Frama-C that `a` and `b` must not overlap. However, the `separated`-clause is marked “experimental” in [8, Subsection 2.7.2] and does not work for our example. Therefore, we commented it out and issued command-line options to Frama-C to ensure the assumption that all different pointers point to disjoint locations, cf. the remarks in Section 1.4 on Page 10.

**Lines 6–7** The `swap_ranges` algorithm alters the elements contained in two distinct ranges, modifying the corresponding elements and nothing else.

**Lines 9–10** The postcondition of `swap_ranges` specifies that the content of each element in its post-state must equal the pre-state of its counterpart. Rephrasing, the old value `\old(b[k])` must be equal to the new value, `a[k]` and vice-versa. Note that the `\old` operator may be applied not only to a variable, but also to an arbitrary expression.

## 4.2.2. Implementation of swap_ranges

Listing 4.4 shows an implementation of `swap_ranges` together with the necessary loop annotations.

```
 1 void swap_ranges(value_type* a, size_type n, value_type* b)
 2 {
 3   /*@
 4     loop assigns a[0..i-1];
 5     loop assigns b[0..i-1];
 6
 7     loop invariant 0 <= i <= n;
 8     loop   variant n-i;
 9
10     loop invariant \forall size_type k; 0 <= k < i ==>
11                      a[k] == \at(b[k],Pre);
12     loop invariant \forall size_type k; 0 <= k < i ==>
13                      b[k] == \at(a[k],Pre);
14   */
15   for (size_type i = 0; i != n; i++)
16     swap(&a[i], &b[i]);
17 }
```

Listing 4.4: Implementation of the `swap_ranges` function

**Line 7** Of course, we must ensure that the index lies within the array bounds. However, we may *not* require simply

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

since the very last loop iteration would violate this formula. Therefor, we have to weaken the formula to that shown in Line 7 of Listing 4.4, which is preserved by *all* iterations of the loop. Note that `0 <= i < n` is still valid immediately before the array accesses in Line 16, 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 2.10 on Page 17.

**Line 10** For the postcondition in Line 9 and 10 in Listing 4.3 to hold, 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`. This is the reason why `k < i` appears rather than `k <= i` in Lines 10 and 12 in Listing 4.4. Figure 4.2 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.

This part of the loop invariant is necessary to prove the postcondition in Line 9 and 10 in Listing 4.3. It is, however, not sufficient. We also need that the parts `a[i..n-1]` and `b[i..n-1]` are left unchanged by the loop body. This additional requirement, which is often forgotten, is entailed by the `loop assigns` clauses in Lines 4–5 of Listing 4.4.

**Lines 4–5** The `assigns` clause declares that nothing except the sub-ranges `a[0..i-1]` and `b[0..i-1]` are modified [8, Subsection 2.4.2]. It is worth noting again the appearance of

|  | 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 4.2.: Loop invariant check points

`i-1` rather than `i`; cf. Figure 4.2. Moreover, note that we have to allow all alterations of all earlier loop cycles, too. For this reason, it is not sufficient to specify

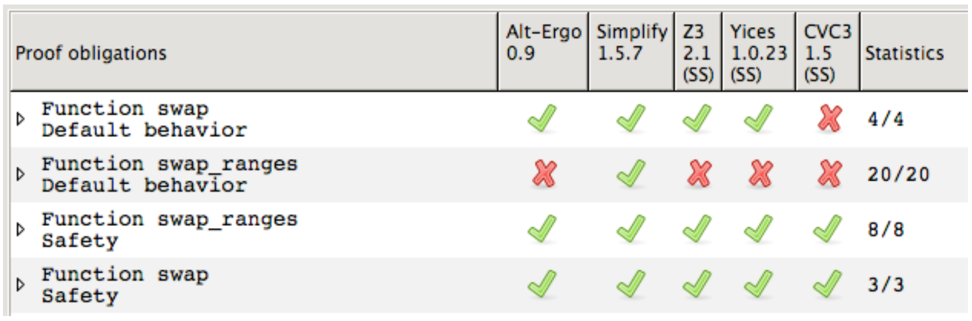
```
loop assigns a[i-1];
loop assigns b[i-1];
```

which would be sufficient only for the current loop cycle.

**Line 8** 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[27] [8, Subsections 2.4.2, 2.5.1]. For a `for` loop as simple as that in Line 15–16, the expression `n-i` is sufficient for that purpose.

### 4.2.3. Formal Verification of `swap_ranges`

Figure 4.3 shows a screen-shot depicting the results of the formal verification of `swap_ranges` with five different automatic theorem provers. Only Simplify was able to prove all verification conditions generated by Jessie.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| Function swap Default behavior | ✓ | ✓ | ✓ | ✓ | ✗ | 4/4 |
| Function swap_ranges Default behavior | ✗ | ✓ | ✗ | ✗ | ✗ | 20/20 |
| Function swap_ranges Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 8/8 |
| Function swap Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 3/3 |

Figure 4.3.: Results of the formal verification of `swap_ranges`

[27]Except for possibly the very last iteration.

## 4.3. The fill Algorithm

The fill algorithm in the C++ Standard Library initialises general sequences with a particular value. For our purposes we have modified the generic implementation[28] to that of an array of type **value_type**. The signature now reads:

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

### 4.3.1. Formal Specification of fill

Listing 4.5 shows the formal specification of fill in ACSL.

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

  assigns a[0..n-1];

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

Listing 4.5: Formal specification of fill

**Line 2** indicates that n is non-negative and that the pointer a points to *n* contiguously allocated objects of type **value_type** (see Section 1.3).

**Line 4** expresses that fill (as a mutating algorithm), may only modify the corresponding elements of a.

**Line 6** formalises the postcondition that for each element of a, a[i] == val holds.

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

### 4.3.2. Implementation of `fill`

Listing 4.6 shows an implementation of `fill`. The only noteworthy elements of this implementation are the three *loop annotations* in Lines 4 to 6.

```
1  void fill(value_type* a, size_type n,  value_type val)
2  {
3    /*@
4      loop invariant 0 <= i <= n;
5      loop   variant n-i;
6      loop invariant \forall size_type k; 0 <= k < i ==> a[k] == val;
7    */
8    for (size_type i = 0; i < n; i++)
9      a[i] = (value_type)val;
10 }
```

Listing 4.6: Implementation of `fill`

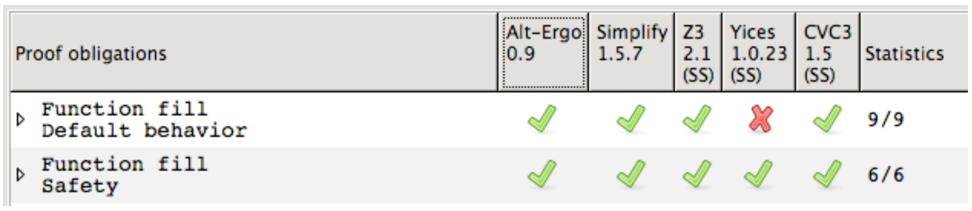
**Line 4** This loop *invariant* is needed to prove that accesses to `a` only occur with valid indices.

**Line 5** This loop *variant* is needed by newer versions of Jessie[29] to generate correct verification conditions for the termination of the loop.

**Line 6** This loop *invariant* 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 Line 6 in Listing 4.5.

### 4.3.3. Formal Verification of `fill`

Figure 4.4 shows a screen-shot of the results of the formal verification of `fill` with five different automatic theorem provers. Only Yices was not able to prove all verification conditions.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| ▷ Function fill Default behavior | ✓ | ✓ | ✓ | ✗ | ✓ | 9/9 |
| ▷ Function fill Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 6/6 |

Figure 4.4.: Results of the formal verification of `fill`

[29]As of Version 2.22 of Why and Jessie.

## 4.4. 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[30] 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 4.2. To tell Frama-C that the arrays `a` and `b` must not overlap, we used appropriate command-line options, cf. the discussion under 4.2.1 on Page 56 f. and Section 1.4 on Page 10.

### 4.4.1. Formal Specification of copy

The ACSL specification of `copy` is shown in Listing 4.7.

```
1  /*@
2    requires is_valid_range(a, n);
3    requires is_valid_range(b, n);
4
5    assigns b[0..n-1];
6
7    ensures \forall size_type i; 0 <= i < n ==> b[i] == a[i];
8  */
9  void copy(const value_type* a, size_type n, value_type* b);
```

Listing 4.7: Formal specification of the `copy` function

**Lines 2–3** We refer to Section 1.3 for a discussion of how valid ranges are specified.

**Line 5** The `copy` algorithm 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.

**Line 7** ensures that the contents of `a` were actually copied to `b`.

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

### 4.4.2. Implementation of copy

Listing 4.8 shows an implementation of the `copy` function.

```
 1  void copy(const value_type* a, size_type n, value_type* b)
 2  {
 3    /*@
 4      loop assigns b[0..i-1];
 5      loop invariant 0 <= i <= n;
 6      loop   variant n-i;
 7      loop invariant \forall size_type k; 0 <= k < i ==> a[k] == b[k];
 8    */
 9    for (size_type i = 0; i < n; i++)
10      b[i] = a[i];
11  }
```

Listing 4.8: Implementation of the `copy` function

Here are some remarks on its loop invariants.

**Line 4** The `assigns` clause ensures that nothing, but the range `b[0..i-1]` is modified.

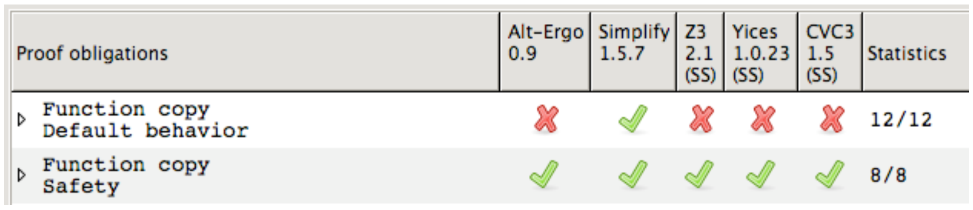
**Line 5** We ensure that the index lies within the bounds of the arrays.

**Line 6** We provide a positive expression decreased at each loop iteration to enable Frama-C to prove the loop termination.

**Line 7** For the postcondition to be true, we must ensure that for every element `i`, the comparison `a[i] == b[i]` is `true`. This loop invariant is necessary to prove the postcondition in Line 7 in Listing 4.7.

### 4.4.3. Formal Verification of copy

In Figure 4.5 a screen-shot shows the results of the formal verification of `copy` with five different automatic theorem provers. Only Simplify was able to prove all verification conditions generated by Jessie.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| ▷ Function copy Default behavior | ✗ | ✓ | ✗ | ✗ | ✗ | 12/12 |
| ▷ Function copy Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 8/8 |

Figure 4.5.: Results of the formal verification of `copy`

## 4.5. The replace_copy Algorithm

The replace_copy algorithm of the C++ Standard Library substitutes specific elements from general sequences. Here, the general implementation[31] 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);
```

replace_copy 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. Again, we need to issue appropriate command-line options to tell Frama-C that the ranges a and b must not overlap, cf. the discussion under 4.2.1 on Page 56 f. and Section 1.4 on Page 10.

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

### 4.5.1. Formal Specification of `replace_copy`

The ACSL specification of `replace_copy` is shown in Listing 4.9.

```
 1  /*@
 2    requires is_valid_range(a, n);
 3    requires is_valid_range(b, n);
 4
 5    assigns b[0..n-1];
 6
 7    ensures \forall size_type j; 0 <= j < n ==>
 8              (a[j] == old_val && b[j] == new_val) ||
 9              (a[j] != old_val && b[j] == a[j]);
10    ensures \result == n;
11  */
12  size_type replace_copy(const value_type* a, size_type n,
13                         value_type* b,
14                         value_type  old_val, value_type new_val);
```

Listing 4.9: Formal specification of the `replace_copy` function

**Lines 2–3** We refer to Section 1.3 for a discussion of how valid ranges are specified.

**Line 5** The algorithm modifies nothing but the destination range viz. `b[0..n-1]`.

**Lines 7–9** 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]`. Instead of Lines 8–9, we could just as well have specified

```
(a[j] == old_val ==> b[j] == new_val) &&
(a[j] != old_val ==> b[j] == a[j]);
```

because the formulae `p && q || !p && r` and `(p ==> q) && (!p ==> r)` are logically equivalent.

**Line 10** formalises that the returned value is equal to the length of the range.

### 4.5.2. Implementation of `replace_copy`

An implementation (including loop annotations) of `replace_copy` is shown in Listing 4.10.

```
 1  size_type replace_copy(const value_type* a, size_type n,
 2                         value_type* b,
 3                         value_type  old_val, value_type new_val)
 4  {
 5    /*@
 6      loop invariant 0 <= i <= n;
 7      loop   variant n-i;
 8      loop assigns b[0..i-1];
 9      loop invariant \forall size_type j; 0 <= j < i ==>
10                     (a[j] == old_val && b[j] == new_val) ||
11                     (a[j] != old_val && b[j] == a[j]);
12    */
13    for (size_type i = 0; i < n; i++)
14      b[i] = (a[i] == old_val ? new_val : a[i]);
15
16    return n;
17  }
```

Listing 4.10: Implementation of the `replace_copy` function

**Line 6** indicates that the loop index must stay within bounds.

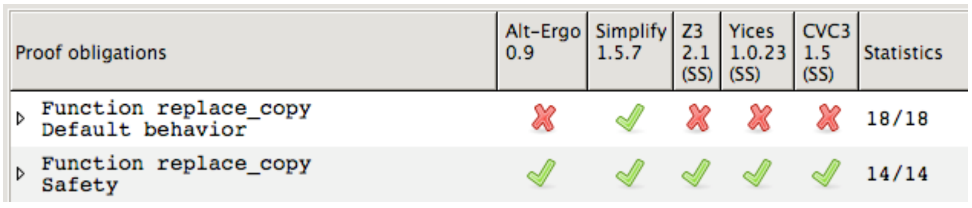
**Line 7** provides a positive expression decreased by every loop iteration to prove loop termination.

**Line 8** The `assigns` clause states that nothing but the range `b[0..i-1]` is modified.

**Lines 9–11** is necessary to prove the Lines 7–9 of the postcondition in Listing 4.9.

### 4.5.3. Formal Verification of `replace_copy`

Figure 4.6 shows a screen-shot depicting the results of the formal verification of `replace_copy`. Only Simplify was able to prove all verification conditions generated by Jessie.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| `Function replace_copy` `Default behavior` | ✗ | ✓ | ✗ | ✗ | ✗ | 18/18 |
| `Function replace_copy` `Safety` | ✓ | ✓ | ✓ | ✓ | ✓ | 14/14 |

Figure 4.6.: Results of the formal verification of `replace_copy`

## 4.6. The **remove_copy** Algorithm

The remove_copy algorithm of the C++ Standard Library allows to remove specific elements from general sequences. Here, the general implementation[32] has been altered to process **value_type** ranges. The new signature reads:

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

The most important facts of this algorithms are

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

Again, we need to issue appropriate command-line options to tell Frama-C that the ranges a and b must not overlap, cf. the discussion under 4.2.1 on Page 56 f. and Section 1.4 on Page 10.

### 4.6.1. A Predicate for **remove_copy**

The ACSL specification of remove_copy in Listing 4.12. refers to a predicate rmcp defined by axioms shown in Listing 4.11.

The predicate rmcp(a, i, b, j, val) is true if, and only if, b[0..j-1] is a copy of a[0..i-1], but with all occurrences of val removed. We first comment on its definition shown in Listing 4.11.

**Lines 4–6** Here we defined the signature of the predicate rmcp, similar to a function prototype in C. Additionally, a **reads** clause is used to state that the truth value of rmcp(a, i, b, j, val) depends only on the contents of a[0..i] and b[0..j]. Although this property is a consequence of the definition in Lines 8–23, we need to state it explicitly, since none of the provers is able to establish it by itself.

**Lines 8–10** Axiom empty states that an empty array a has an empty filtered copy b.

**Lines 12–16** Axiom equal assumes that b[0..j-1] is already a filtered copy of a[0..i-1] and that a[i] equals the value val to be removed. It states that under these circumstances, b[0..j-1] is also a filtered copy of a[0..i].

**Lines 18–23** Axiom not_equal assumes again that b[0..j-1] is already a filtered copy of a[0..i-1], but that a[i] is now different from val. It states that if in this situation a[i] is copied to b[j], then b[0..j] is a filtered copy of a[0..i].

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

```
/*@
  axiomatic rmcp_axioms
  {
    predicate rmcp{L}(value_type* a, integer i,
                      value_type* b, integer j, integer val)
                reads b[0..j-1], a[0..i-1];

    axiom empty{L}:
      \forall value_type* a, value_type* b, integer val;
                rmcp(a, 0, b, 0, val);

    axiom equal{L}:
      \forall value_type* a, integer i,
              value_type* b, integer j, integer val;
                rmcp(a, i, b, j, val) && a[i] == val ==>
                  rmcp(a, i+1, b, j, val);

    axiom not_equal{L}:
      \forall value_type*a, integer i,
              value_type* b, integer j, integer val;
                rmcp(a, i, b, j, val) &&
                  a[i] != val && a[i] == b[j] ==>
                    rmcp(a, i+1, b, j+1, val);
  }
*/
```

Listing 4.11: Axiomatic definition used in the specification of the `remove_copy` function

### 4.6.2. Formal Specification of **remove_copy**

Using the predicate `rmcp`, we can specify the function contract of `remove_copy` in a brief and elegant way, as shown in Listing 4.12.

```
 1  /*@
 2    requires is_valid_range(a, n);
 3    requires is_valid_range(b, n);
 4
 5    assigns  b[0..n-1];
 6    ensures  \forall int k; \result <= k < n ==> b[k] == \old(b[k]);
 7
 8    ensures  rmcp(a, n, b, \result, val);
 9    ensures  0 <= \result <= n;
10  */
11  size_type remove_copy(const value_type* a, size_type n,
12                        value_type* b, value_type val);
```

Listing 4.12: Formal specification of the `remove_copy` function

**Lines 2–3** We refer to Section 1.3 for a discussion of how valid ranges are specified.

**Lines 5–6** The algorithm modifies nothing but the destination range.

To specify this, the most convenient way would be:

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

However, the `assigns` clause refers to the pre-state of the function, where the identifier `\result` is unknown.

A second alternative would be:

```
assigns b[0..\at(\result-1,Post)];
```

However, this feature is not yet supported in the Beryllium release of Frama-C. We therefore eventually came up with Lines 5–6 in the specification. Since an **`ensures`** clause refers to the post-state of the function, this works fine.

**Line 8** This is the essential part of the specification. It simply says that `remove_copy` should behave according to `rmcp`.

**Line 9** We included some redundancy into the specification by asserting that the return-value, i.e., the fill-degree of `b`, needs to be in `[0..n-1]`. Similar to the paradigm of "defensive programming", we consider it a good idea to specify in a defensive way, too. There is no reason why a formal specification couldn't contain a bug, i.e., a discrepancy between what its author had in mind and what it actually does mean. Note that "defensive specification", in contrast to "defensive programming", cannot increase the execution time of the verified program at all.

### 4.6.3. Implementation of `remove_copy`

An implementation (including loop annotations) of `remove_copy` is shown in Listing 4.13.

```
 1  size_type remove_copy(const value_type* a, size_type n,
 2                        value_type* b, value_type val)
 3  {
 4    size_type j = 0;
 5    /*@
 6      loop invariant 0 <= j <= i <= n;
 7      loop   variant n - i;
 8
 9      loop invariant rmcp(a,i,b,j,val);
10
11      loop assigns   b[0..j-1];
12    */
13    for (size_type i = 0; i < n; i++)
14      if (a[i] != val)
15        b[j++] = a[i];
16
17    return j;
18  }
```

Listing 4.13: Implementation of the `remove_copy` function

Almost each assertion of the loop invariant is motivated by a corresponding assertion of the specification (Listing 4.12). The former usually tells the provers how the latter is to be achieved during the loop at each step.

**Line 6** The indices `i` and `j` must stay within the range bounds. Moreover, we always have `j <= i`; the latter is needed to prove Line 8 of the specification.

**Line 7** The expression `n-i` is always positive, and is decreased in every loop cycle, thus ensuring loop termination.

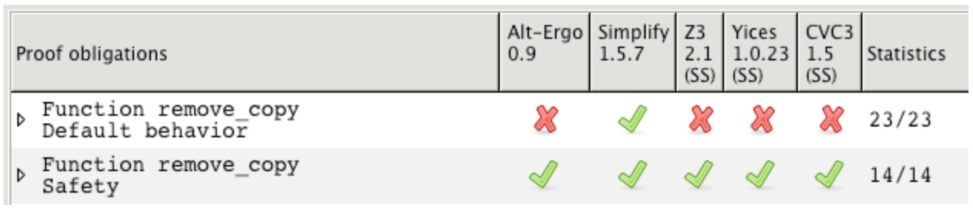
**Line 9** Again, we simply refer to the predicate `rmcp` to specify the most essential part of the loop invariant. This is needed to prove Line 7 of the specification in Listing 4.12.

**Line 11** The **`loop`** `assigns` clause ensures that nothing but `b[0..j-1]` is modified. It is necessary to prove Line 5 of the specification.

During the proof attempts, we found that it is necessary to tell the provers that `rmcp(a, i, b, j, val)` depends only on `a[0..i-1]` and `b[0..j-1]`, but no other elements in `a[]` or `b[]`. While this is obvious to a human reader, once he has understood the definition of `rmcp` in Listing 4.11, a prover would have to perform an induction proof to establish that property. However, none of the currently employed provers is prepared to do that. For this reason, we had to tell them that property explicitly, which was done using the **`reads`** clause in Line 6 in Listing 4.11.

### 4.6.4. Formal Verification of remove_copy

Figure 4.7 shows the results of the formal verification of remove_copy. Again, only Simplify was able to prove all verification conditions generated by Jessie.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| Function remove_copy Default behavior | ✗ | ✓ | ✗ | ✗ | ✗ | 23/23 |
| Function remove_copy Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 14/14 |

Figure 4.7.: Results of the formal verification of remove_copy

## 4.7. An Alternative Specification of the remove_copy Algorithm

The specification of remove_copy in Section 4.6 essentially relies on the predicate rmcp, which is defined (in Listing 4.11 on Page 69) in an algorithmic manner. Its definition may be considered as semantically equivalent to the C-function shown in Listing 4.14[33]. On the other hand, Listing 4.14 may be considered as a recursive version of the implementation of remove_copy from Listing 4.13 on Page 71.

```
size_type rmcp_C(const value_type* a, size_type i,
                 value_type* b, value_type val)
{
  size_type j;
  if (i == 0)                             // axiom empty
  {
    j = 0;
  }
  else if (a[i-1] == val)                 // axiom equal
  {
    j = rmcp_C(a, i-1, b, val);
  }
  else /*  a[i-1] != val */               // axiom not_equal
  {
    j = rmcp_C(a, i-1, b, val) + 1;
    b[i-1] = a[j-1];
  }
  return j;
}
```

Listing 4.14: C-implementation of the predicate rmcp from Listing 4.11

From this point of view, we verified in Section 4.6 a program against itself, which is not the most

[33]Readers familiar with Prolog will have noticed that Listing 4.11 can be viewed as a predicate definition consisting of three clauses, viz. one for each axiom. Listing 4.11 is just a translation of this Prolog program into ACSL.

suitable approach to increase our confidence in the correctness of that program [34].

In our opinion, the verification approach reaches its full strength when the specification describes just *what* task is to be achieved while only the implementation describes in detail *how* to achieve it. As an analogy, the implementation corresponds to the process of solving some mathematical equation system, while a good specification should correspond to the process of checking whether the found solution makes both sides of any equation in fact equal. For example, the correctness of a cryptographic algorithm may be specified easily as shown in Listing 4.15, while the implementation of `encrypt` and its `decrypt` counterpart may be of arbitrary complexity. The same applies to the square root function, for which a brief and obvious specification is shown in Listing 4.16.

```
extern void encrypt(char* code,  const char* clear);
extern void decrypt(char* clear, const char* code);
extern int  strcmp(const char* string1, const char* string2);

//@ ensures \result == 0;
int simulated_channel(char* clearReceived, const char* clearSent)
{
    char code[maxCodeLength];
    encode(&code, clearSent);
    decode(clearReceived, &code);
    return strcmp(clearReceived, clearSent);
}
```

Listing 4.15: Specification of a cryptographic algorithm

```
extern double const epsilon;
/*@ requires 0.0 <= x;
    ensures  x - epsilon <= \result * \result <= x + epsilon;
*/
extern double sqrt(double x);
```

Listing 4.16: Specification of the square-root function

Both (classes of) algorithms are too complex to be handled in this introductory tutorial. Instead, we show in this section an alternative specification of `remove_copy` which is non-algorithmic and, as we believe, closer to the informal description.

[34] It is, however, still superior to the classical "clean-room" approach in yielding a *formal equivalence proof* of two algorithms for the same task.

### 4.7.1. Formal Specification of remove_copy

The ACSL specification of the alternative version of `remove_copy` is shown in Listing 4.17.

```
 1  //@ ghost static size_type g[];
 2
 3  /*@
 4    requires is_valid_range(a,n);
 5    requires is_valid_range(b,n);
 6    requires is_valid_range(g,n);
 7
 8    assigns  b[0..n-1], g[0..n-1];
 9    ensures  \forall integer k; \result <= k < n ==> b[k] == \old(b[k]);
10
11    ensures  \forall integer k; 0 <= k < \result ==> b[k] != val;
12    ensures  \forall value_type x; x != val ==>
13               counting(a, n, x) == counting(b, \result, x);
14
15    ensures  \result == n - counting(a, n, val);
16    ensures  0 <= \result <= n;
17
18    ensures  \forall integer k; 0 <= k < \result ==> b[k] == a[g[k]];
19    ensures  \forall integer k; 0 <  k < \result ==> g[k-1] < g[k];
20  */
21  size_type remove_copy(const value_type* a, size_type n,
22                        value_type* b, value_type val);
```

Listing 4.17: Alternative formal specification of the `remove_copy` function

**Line 1** The informal description (see Page 68) requires that `remove_copy` preserves the order of copied elements. To formalize this, we construct an order-preserving mapping `g` such that `b[k] == a[g[k]]` for all relevant values of `k`. Since this construction is only needed for verification purposes, we realize it using ghost code, that is, additional C code which is visible only to Frama-C, but not to an ordinary C compiler; see [8, Sect. 2.12] for details. First, we have to declare the array `g[]` that will hold the constructed mapping. We have to do this before and outside of the function contract of `remove_copy`. An admitted disadvantage of this approach is that `g` is visible (for Frama-C) in the entire file containing `remove_copy`'s implementation.

**Lines 4–6** We refer to Section 1.3 for a discussion of how valid ranges are specified.

Note that we also specify `g`'s length here, viz. as `[0..n-1]`. We didn't need to provide the length in `g`'s declaration in Line 1, since no memory will be allocated for `g`. Moreover, we weren't even able to provide it there, since it depends on `n`, which is available not earlier than in `remove_copy`'s function contract.

**Lines 8–9** The algorithm modifies nothing but the destination range and the order-preserving map `g`.

We refer to the discussion on Line 5 of Listing 4.12, concerning the issue of `\separated`. Line 8–9 may be considered as a translation of the second sentence of `remove_copy`'s informal description on Page 68 into ACSL.

**Line 11** The `remove_copy` algorithm eliminates all occurrences of `val` in the destination sequence. Hence, no entry in the destination range may be equal to `val`.

**Lines 12–13** Except for `val`, each value is found in `b` as often as in `a`. We referred to the logical function `counting`, defined in Listing 3.26 on Page 51 to express this property in a formal way. Note that we made an assertion about infinitely many integer numbers `x`. While the approach of theorem proving is still sufficiently powerful to verify that assertion, there is no way to verify it using only `C` code like, e.g., **`assert`**`(...)`.

Altogether, Line 11–13 is a direct translation of the first sentence of `remove_copy`'s informal description on Page 68 into ACSL.

**Line 15** The length up to which the output array `b` is filled is just the length `n` of the input array `a`, diminished by the number of ignored copies of `val`.

**Line 16** We included some redundancy into the specification by asserting that the return-value, i.e., the fill-degree of `b`, needs to be in `[0..n]`.

**Line 18–19** We finally give a translation of the third sentence of `remove_copy`'s informal description on Page 68 into ACSL. We require that each place `k` in `b[]` can be mapped back to a place `g[k]` in `a[]` where `b`'s contents originated from (Line 18), and that this mapping `g[]` preserves order (Line 19). Figure 4.8 illustrates this idea; the left example is admitted by Line 19, while the red crossing arrows in the right example are forbidden to occur due to Line 19.

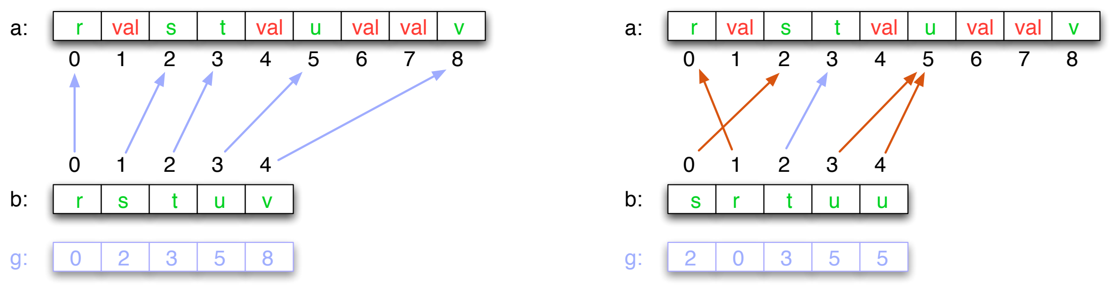


Figure 4.8.: Allowed (left) and forbidden (right) contents of `g[]` in `remove_copy`

## 4.7.2. Modified Implementation of remove_copy

Listing 4.18 shows the same implementation of remove_copy as Listing 4.13, but with an additional ghost statement and a loop invariant adapted to our new specification.

```
 1  size_type remove_copy(const value_type* a, size_type n,
 2                        value_type* b, value_type val)
 3  {
 4    size_type j = 0;
 5    /*@
 6      loop invariant 0 <= j <= i <= n;
 7      loop   variant n - i;
 8
 9      loop assigns   g[0..j-1], b[0..j-1];
10      loop invariant \forall integer k; j <= k < n ==>
11        b[k] == \at(b[k],Pre);
12
13      loop invariant \forall integer k; 0 <= k < j ==> b[k] != val;
14      loop invariant \forall value_type x; x != val ==>
15        counting(a, i, x) == counting(b, j, x);
16
17      loop invariant j == i - counting(a, i, val);
18
19      loop invariant \forall integer k; 0 <= k < j ==> b[k]==a[g[k]];
20      loop invariant \forall integer k; 0 <  k < j ==> g[k-1] < g[k];
21      loop invariant 0 < j ==> g[j-1] < i;
22    */
23    for (size_type i = 0; i < n; i++)
24      if (a[i] != val)
25      {
26        //@ ghost g[j] = i;
27        b[j++] = a[i];
28      }
29
30    return j;
31  }
```

Listing 4.18: Alternative implementation of the remove_copy function

We first comment on our new loop invariant, which uses the ghost array g[] declared in Line 1 in Listing 4.17 to hold the order-preserving mapping from b's back to a's places.

**Line 6** The indices i and j must stay within the range bounds. Moreover, we always have j <= i; the latter is needed to prove Line 16 of the specification in Listing 4.17.

**Line 7** The expression n-i is always positive, and is decreased in every loop cycle, thus ensuring loop termination.

**Line 9** The **loop** assigns clause ensures that nothing but g[0..j-1] and b[0..j-1] is modified. It is necessary to prove Line 8 of the specification.

**Line 10** The array b is not changed beyond the index j. This loop invariant is needed to prove

Line 9 of the specification.

**Line 13** The changed part of `b` does not contain the value `val`. This loop invariant is needed to prove Line 11 of the specification.

**Line 14** In every loop cycle, we have read the range `a[0..i-1]` and written the range `b[0..j-1]`. We require that except for `val`, every value occurs the same number of times in both ranges. This is needed to prove Lines 12–13 of the specification.

**Line 17** The current fill-degree of `b`, viz. `j`, equals the currently processed range upper bound `i` of `a`, diminished by the number of ignored occurrences of `val` therein. This is needed to prove Line 15 of the specification.

**Line 19** The mapping `g[]` maps each place `k` in `b[]` back to a place `g[k]` in `a[]` where `b`'s contents originated from. This is needed to prove Line 18 of the specification.

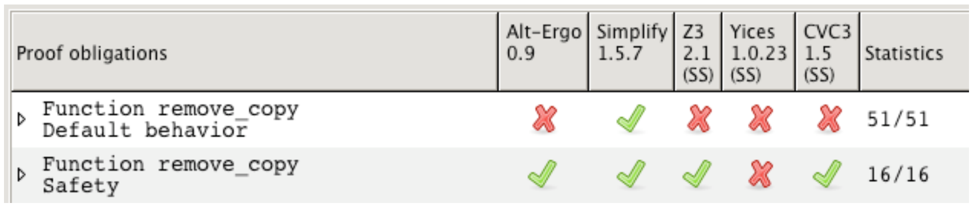
**Line 20** The mapping `g[]` preserves order. This is needed to prove Line 19 of the specification.

**Line 21** The last value assigned to `g`, viz. `g[j-1]` is less that the value we possibly assign to `g` in the current loop cycle, viz. `i`. This is needed to establish Line 19 of our loop invariant.

Only one modification to the `C` code was necessary, viz. the additional Line 26. There, in order to construct the mapping `g[]`, we assign `i` to `g[j]` whenever we assign `a[i]` to `b[j]`. Since this line is ghost code, it is not visible to a `C` compiler. Hence the machine code generated from Listing 4.18 will be the very same than that generated from Listing 4.13.

### 4.7.3. Formal Verification of the Alternative `remove_copy` Function

Figure 4.9 shows the results of the formal verification of the alternative version of `remove_copy`. Simplify was the only prover that was able to prove all verification conditions generated by Jessie.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| Function remove_copy Default behavior | ✗ | ✓ | ✗ | ✗ | ✗ | 51/51 |
| Function remove_copy Safety | ✓ | ✓ | ✓ | ✗ | ✓ | 16/16 |

Figure 4.9.: Results of the formal verification of `remove_copy`

## 4.8. 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[35] 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.

### 4.8.1. Formal Specification of `iota`

The ACSL specification of `iota` is shown in Listing 4.19.

```
1  /*@
2    requires is_valid_range(a, n);
3    requires val + n < MAX_INT;
4
5    assigns a[0..n-1];
6
7    ensures \forall size_type k; 0 <= k < n ==> a[k] == val + k;
8  */
9  void iota(value_type* a, size_type n, value_type val);
```

Listing 4.19: Formal specification of the `iota` function

**Lines 2** We refer to Section 1.3 for a discussion of how valid ranges are specified.

**Lines 3** is necessary to avoid integer overflows in `iota`.

**Line 5** The `iota` algorithm initializes and modifies the elements in range `a`. Elements outside the range `a[0..n-1]` are not altered.

**Line 7** Upon termination, each element of `a` contains the sum of its index within `a` and the argument `val`.

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

### 4.8.2. Implementation of `iota`

Listing 4.20 shows an implementation of the `iota` function.

```
1  void iota(value_type* a, size_type n, value_type val)
2  {
3    /*@
4      loop assigns a[0..i-1];
5      loop invariant 0 <= i <= n;
6      loop   variant n-i;
7      loop invariant \forall size_type k; 0 <= k < i ==> a[k] == val + k;
8    */
9    for (size_type i = 0; i < n; i++)
10     a[i] = val + (value_type)i;
11 }
```

Listing 4.20: Implementation of the `iota` function

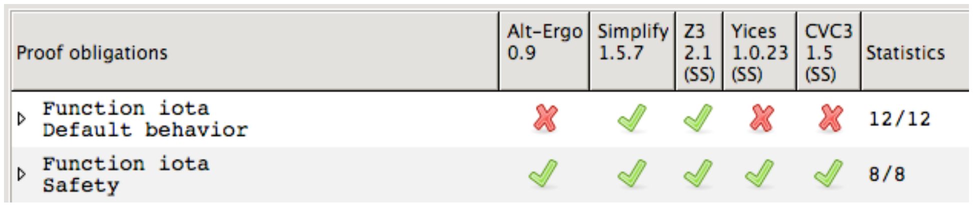
**Line 4** The `assigns` clause ensures that nothing but the range `a` is modified:

**Line 5** We must ensure that the index lies within the range bounds.

**Line 7** This loop invariant ensures that for each iteration, the elements contain the proper value. It is essential to prove the postcondition in Line 5 in Listing 4.19.

### 4.8.3. Formal Verification of `iota`

Figure 4.10 shows a screen-shot of the results of the formal verification of `iota` with five different automatic theorem provers. Simplify and Z3 were able to prove each of the generated verification conditions.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| ▷ Function iota Default behavior | ✗ | ✓ | ✓ | ✗ | ✗ | 12/12 |
| ▷ Function iota Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 8/8 |

Figure 4.10.: Results of the formal verification of `iota`

# 5. Heap Operations

We cite the Apache C++ Standard Library User's Guide heap operations:

> A heap is a binary tree in which every node is larger than the values associated with either child. A heap and a binary tree, for that matter, can be very efficiently stored in a vector, by placing the children of node $i$ at positions $2i + 1$ and $2i + 2$.
>
> Using this encoding, the largest value in the heap is always located in the initial position, and can therefore be very efficiently retrieved. In addition, efficient logarithmic algorithms exist that permit a new element to be added to a heap and the largest element removed from a heap. For these reasons, a heap is a natural representation for the priority queue data type.[36]

Figure 5.1 shows the binary tree of the heap indices 0, 1, . . . , (n-1) for n = 12.

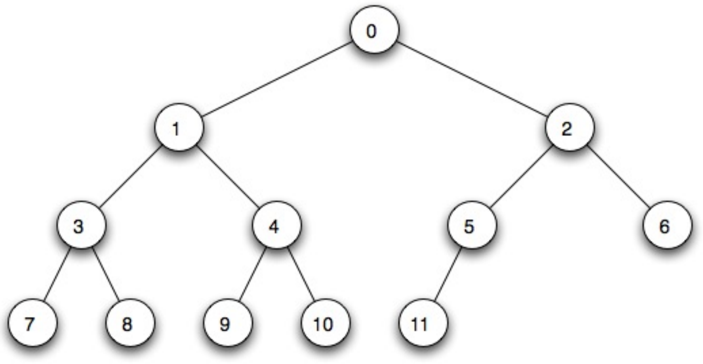


Figure 5.1.: Binary tree of the heap indices 0, 1, . . . , 11

Only in case the heap-size `n` is even, there is exactly one parent node with one child only. All other nodes have two or no children at all.

As mentioned in the citation above for performance reasons a (complete) ordering of the heap's data array is not appropriate . The fractional ordering—parents value is not less than children's values— is called weak ordering. This weak ordering can be maintained with logarithmic effort.

In this chapter we consider various heap-related algorithms:

**is_heap** from Section 5.2 allows to test at run time whether a given array is arranged as a heap.

**push_heap** from Section 5.3 *adds* an element to a given heap in such a way that resulting array is again a heap.

[36]See Section 14.7 in `http://stdcxx.apache.org/doc/stdlibug/14.html`

**pop_heap** from Section 5.4 *removes* an element from a given heap in such a way that the resulting array is again a heap.

**make_heap** from Section 5.5 turns a given array into a heap.

**sort_heap** from Section 5.6 turn a given heap into a sorted range.

**heap_sort** from Section 5.7 does not directly correspond to an STL algorithms. It uses `make_heap` and `sort_heap` to implement a robust sorting algorithm.

The essential predicates that formalize the properties of heap are introduced in Section 5.1.

In this Chapter, we deliberately omitted all properties speaking about preservation of array contents.

For example, the function contract of `heap_sort` ensures the the resulting array is ordered, but it does not ensure that it contains the same elements as before the `heap_sort`-call. Such a specification could also be implemented by assigning `0` to all heap elements `c[i]`, which is certainly not what a user of this algorithm has in mind. It is a common pitfall to forget to specify contents-preservation properties, since they seem to be all too obvious to be kept in mind explicitly.

We will supply the missing specification parts and the corresponding correctness proofs in a coming issue of this tutorial. Here we only indicate this gap by placing a non-ACSL comment

```
// ensures Permutation\{Old, Here\}(c, n)}
```

in each of the function contracts.

## 5.1. Axiomatic Description of Heaps

The ACSL specification for the edge relation `ParentChild` in the tree and the definition of the heap's weak ordering predicate `IsHeap` is shown in Listing 5.1.

The meanings of **axiom** `A1` and **axiom** `A2` are identical, however (up to now) the provers ”don't know it” or can't proof it. It depends on the upwards or downwards direction the tree algorithms are operating which of these axioms is to be applied.

The **lemma** `A3` can be proofed and is a necessary hint for the provers in the verification processes.

```
/*@
  axiomatic ParentChildAxioms
  {
    predicate ParentChild(integer i, integer j);

    axiom A1:
      \forall integer i, j; ParentChild(i, j) <==>
         0 <= i < j && (j == 2*i+1 || j == 2*i+2);

    axiom A2:
      \forall integer i, j; ParentChild(i, j) <==>
        0 < j && i == (j-1)/2;
  }

  lemma A3:
    \forall integer i; 0 < i ==>
      ParentChild((i-1)/2, i) && 0 <= (i-1)/2 < i;

  predicate IsHeap{L}(value_type* c, integer n) =
    \forall integer i, j;
      j < n && ParentChild(i, j) ==> c[i] >= c[j];
*/
```

Listing 5.1: Axiomatic definition used in the heap-specification of the binary tree relation and the weak ordering property

`ParentChild(i,j)` holds if the heap index `i` corresponds to the parent node of the node corersponding to index `j`. Note that is does not depend on the contents of the heap array.

`IsHeap(c,n)` holds if the array `c[0..n-1]` forms a heap in the sense of the above definition from the STL User's Guide (Page 81), i.e. if it is weakly ordered.

## 5.2. The is_heap Algorithm

The function is_heap tests whether a given array has the heap-property described on Page 5.

The is_heap algorithm of the STL in the Library works on generic sequences. For our purposes we have modified the generic implementation[37] to that of an array of type **size_type**. The signature now reads:

```
bool is_heap(const value_type* a, int n);
```

### 5.2.1. Formal Specification of is_heap

The ACSL specification of is_heap is shown in Listing 5.2. The function returns 1 if and its argument satisfies the properties of a heap. Otherwise, is_heap returns 0.

```
/*@
   requires is_valid_range(c, n);

   assigns \nothing;

   ensures \result == 0 || \result == 1;
   ensures IsHeap(c, n) <==> \result == 1;
*/
bool is_heap(const value_type* c, size_type n);
```

Listing 5.2: Function contract of is_heap

[37]See http://www.sgi.com/tech/stl/is_heap.html. Note that is_heap is not part of the C++ standard library.

### 5.2.2. Axiomatic Description of even and odd Numbers

Our implementation of `IsHeap` uses the distinction of even and odd numbers to increase run-time efficiency. It test whether a number is even or odd by using the C-operator `&` for *bit-wise and*. As it turns out we need additional axioms (see Listing 5.3 to help the provers understanding properties of operator `&`.

```
/*@
  axiomatic Ampersand
  {
    predicate Even(integer i) = (i&1) == 0;

    predicate Odd(integer i)  = (i&1) == 1;

    axiom D1: \forall integer i;
      Odd(i) && Even(i+1) || Even(i) && Odd(i+1);

    axiom D2: \forall integer i;
      i > 0 && Odd(i) ==> (i-1)/2 == i/2;

    axiom D3: \forall integer i;
      i > 0 && Even(i) ==> (i-1)/2 + 1 == i/2;
  }

  lemma D0: Even(0); // not needed as axiom
*/
```

Listing 5.3: Axioms concerning even and odd

We suppose that our readers are familiar with the properties stated by the axioms. We therefore give only short explanation of the axioms in Listing 5.3.

**Axiom D1** state that integers in their natural order are alternating even or odd.

**Axiom D2** says with other words in our heap-tree terminology: If `i > 0` and `i` is odd, then child `i` and child `i+1` have the same parent.

**Axiom D3** is similar to D2.

**Lemma D0** shows that the prover "knows" something about the function `x -> x&1`.

It is interesting, that neither in the specification nor in the implementation of `is_heap` the predicates `Even` and `Odd` are explicitly used.

### 5.2.3. Implementation of is_heap

Listing 5.4 shows one way to implement the function is_heap.

```
bool is_heap(const value_type* c, size_type n)
{
  size_type parent = 0;
  /*@
    loop invariant 0 <= parent < child <= n+1;
    loop   variant n - child;

    loop invariant parent == (child-1)/2;
    loop invariant \forall integer i;
      0 < i < child ==> c[(i-1)/2] >= c[i];

    loop invariant IsHeap(c, child-1);
  */
  for (size_type child = 1; child < n; child++)
  {
    if (c[parent] < c[child])
      return 0;

    if ((child & 1) == 0)
    {
      //@ assert ParentChild(parent+1, child+1);
      parent++;
    }
    //@ assert (child & 1) == 1 ==> ParentChild(parent, child+1);
  }
  return 1;
}
```

Listing 5.4: Implementation of is_heap

The working scheme of is_heap is easily to explain with Figure 5.2.

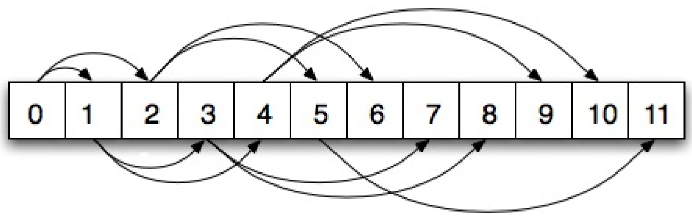


Figure 5.2.: Array sight of the binary heap tree

While incrementing a child it may happen, that the corresponding parent does not change or alternatively that the parent must also be incremented. The second variant happens exactly in case the child is even. The implementation of is_heap exploits this to traverse the complete binary heap tree.

**Lines 19–23** in connection with the parent initialization and the `for`-loop are responsible for the tree parsing.

**Line 8** This loop invariant preserves the parent-child connection.

**Lines 9–12** establish the heap property as long the test `if (c[parent] < c[child])` failed.

**Lines 21 and 24** are hints given to the prover.

Finally we remark that the slightly different program of Listing 5.5 would be much easier to verify.

```
int simple_is_heap(const value_type* c, size_type n)
{
  for (size_type i = 1; i < n; i++)
    if (c[i] > c[(i-1)/2])
      return 0;

  return 1;
}
```

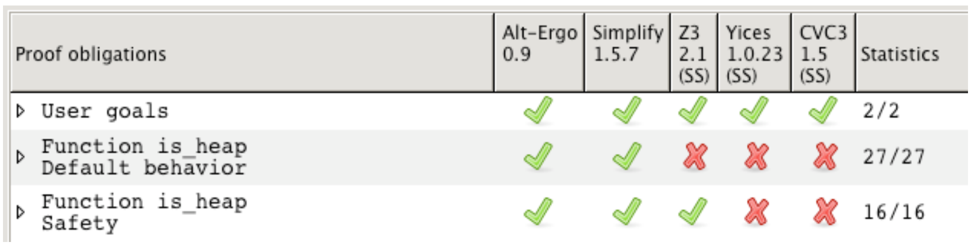
Listing 5.5: An implementation of `is_heap` that is easier to verify

However, the former one is more interesting for two reasons.

1. Memory access is costly and an optimizing compiler can more easily avoid the memory access to `parent`'s value in every second iteration step.
2. It was a challenge to find and exploit the even/odd axioms of Listing 5.3. There is for example the interesting fact, that after substituting axiom D1 for `Even(i) ==> Odd(i+1)` and `Odd(i) ==> Even(i+1)` the prover seems to enter an endless loop.

### 5.2.4. Formal Verification of `is_heap`

Figure 5.3 shows the results of the formal verification of `is_heap` with five different automatic theorem provers. Alt-Ergo and Simplify were able to prove all verification conditions generated by Jessie.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| ▷ User goals | ✓ | ✓ | ✓ | ✓ | ✓ | 2/2 |
| ▷ Function is_heap Default behavior | ✓ | ✓ | ✗ | ✗ | ✗ | 27/27 |
| ▷ Function is_heap Safety | ✓ | ✓ | ✓ | ✗ | ✗ | 16/16 |

Figure 5.3.: Results of the formal verification of `is_heap`

## 5.3. The push_heap Algorithm

The push_heap algorithm adds an element that is at the end of an array to an existing heap consisting of the prior elements in the array. When push_heap returns, the heap-property of the whole array including the new element is established. Whereas in the STL push_heap works on a pair of random access iterators,[38] our version operators on a range of **value_type**. Thus the signature of push_heap reads

```
void push_heap(value_type* c, size_type n);
```

### 5.3.1. Formal Specification of push_heap

Listing 5.6 shows our formal specification of push_heap in ACSL.

```
 1  /*@
 2     requires n > 0;
 3     requires is_valid_range(c, n);
 4     requires IsHeap(c, n-1);
 5
 6     assigns c[0..(n-1)];
 7
 8     ensures IsHeap(c, n);
 9     // ensures Permutation{Old, Here}(c, n);
10  */
11  void push_heap(value_type* c, size_type n);
```

Listing 5.6: Specification of push_heap

**Line 2** n > 0 is required because at least the "new" element c[n-1] must be in the array.

**Line 4** requires that before the call of push_heap all the prior elements form a heap.

**Line 8** ensures that after the call all the elements including c[n-1] form a heap.

**Line 9** indicates the missing content-preservation specification.

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

## 5.3.2. Implementation of push_heap

Listing 5.7 shows the implementation of push_heap.

```
 1  void push_heap(value_type* c, size_type n)
 2  {
 3    const value_type tmp = c[n-1];
 4    size_type hole = n-1;
 5    /*@
 6       loop invariant 0 <= hole < n;
 7       loop   variant hole;
 8  
 9       loop invariant hole < n-1 ==> IsHeap(c, n);
10       loop invariant \forall integer i, j;
11         j < n && j != hole && ParentChild(i, j) ==> c[i] >= c[j];
12  
13       loop invariant \forall integer i;
14         i < n  && ParentChild(hole, i) ==> c[i] <= tmp;
15    */
16    while (hole > 0)
17    {
18      const size_type parent = (hole-1)/2;
19  
20      if (c[parent] < tmp)
21        c[hole] = c[parent];
22      else
23        break;
24  
25      hole = parent;
26    }
27    c[hole] = tmp;
28  }
```

Listing 5.7: Implementation of push_heap

**Line 3** saves the new element `c[n-1]` in a variable `tmp` and

**Line 4** initializes `hole` with the top index `n-1`.

**Lines 16–26** The loop is searching for the right place for the new element `tmp` by moving the `hole` upwards and shifting down the passed elements. In each loop step the position `hole` is tested whether it is the right place to store `tmp`.

**Line 27** The right place `hole` for `tmp` has been found now and `tmp` can be dropped.

**Lines 9–11** These loop invariants are needed to guarantee finally the heap property. Note that `IsHeap(c, n)` is valid only after the first iteration step.

**Lines 13–14** This invariant ensures that dropping `tmp` to the final `hole`-place does not destroy the heap property.

Figure 5.4 illustrates the iteration steps of the `push_heap` function. When `push_heap(c, 12)` is called, the range `c[0..10]` forms a valid heap, and `c[11]` contains the value `77` that has been recently appended. The task of `push_heap` is to move the new `77` to its appropriate place in order to make the whole range `c[0..11]` a valid heap.

First, the `77` is moved from `c[11]` to the local variable `tmp` (Line 3 in Listing 5.7). Thereby, we get a “hole” at `c[11]` (Line 4). The loop in Lines 16–26 will move that hole upwards in the tree until the `77` can be dropped (Line 27) without violating the heap property.

In the first loop cycle, we compare the `8` at the parent position `c[5]` against the `77`. Since the `77` is still larger, we move the hole one level upwards in the tree, i.e. we move the `8` downwards. The resulting situation is depicted in Figure 5.4.

In the second cycle, we similarly compare the `17` at `c[2]` against the `77`, and move the hole up to `c[2]`. In the third cycle, we move the hole up again, to `c[0]`. After that, we leave the loop since `hole > 0` in Line 16 no longer holds, and drop the `77` to `c[0]`. Now the whole range `c[0..11]` is a valid heap.

11

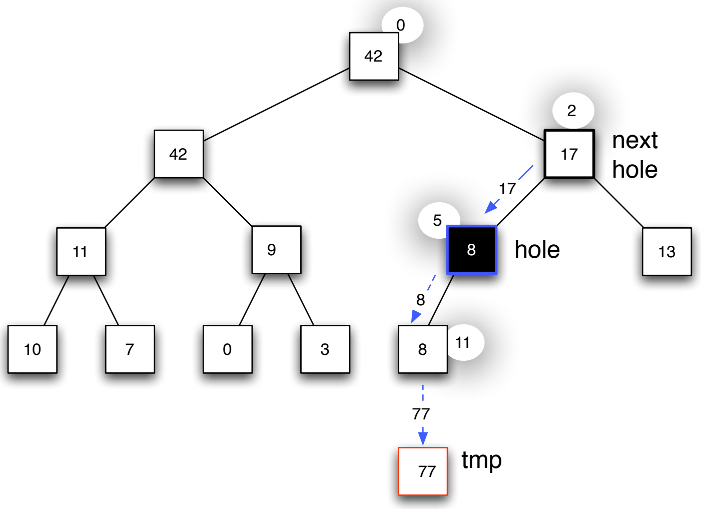
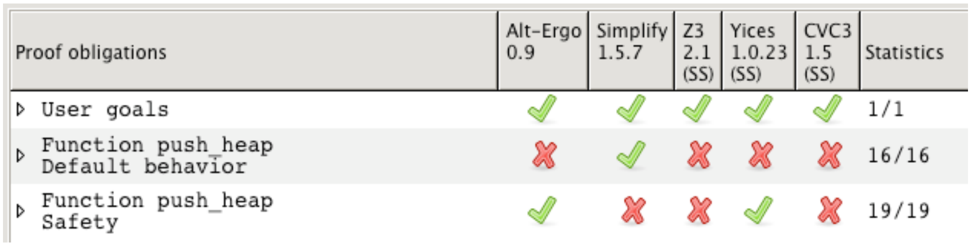


Figure 5.4.: An iteration step of `push_heap`

### 5.3.3. Formal Verification of `push_heap`

Figure 5.5 shows the results of the formal verification of `push_heap` with five different automatic theorem provers. None of those provers was able to prove all verification conditions generated by Jessie. However, Simplify was able to prove each verification condition except for the ones generated to examine arithmetic overflows in the "safety" group of Figure 5.5. This group of proof obligations was proven by Alt-Ergo and Yices instead.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| User goals | ✓ | ✓ | ✓ | ✓ | ✓ | 1/1 |
| Function push_heap Default behavior | ✗ | ✓ | ✗ | ✗ | ✗ | 16/16 |
| Function push_heap Safety | ✓ | ✗ | ✗ | ✓ | ✗ | 19/19 |

Figure 5.5.: Results of the formal verification of `push_heap`

## 5.4. The pop_heap Algorithm

The function pop_heap is in some sense converse to push_heap (see Section 5.3). It moves the largest element from the front of a heap to the last position in the array and then forms a new heap from the remaining elements.

Whereas in the STL pop_heap works on a pair of random access iterators,[39] our version operators on a range of **value_type**. Thus the signature of pop_heap reads

```
void pop_heap(value_type* c, size_type n);
```

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

### 5.4.1. Formal Specification of pop_heap

The function contract of pop_heap in ACSL is given by Listing 5.8.

```
 1  /*@
 2     requires 0 < n <  (MAX_INT-2)/2;
 3     requires is_valid_range(c, n);
 4     requires IsHeap(c, n);
 5
 6     //assigns c[0..(n-1)];
 7
 8     ensures IsHeap(c, n-1);
 9     ensures c[n-1] == \old(c[0]);
10     ensures \forall integer i; 0 <= i < n ==> c[n-1] >= c[i];
11     // ensures Permutation{Old, Here}(c, n);
12  */
13  void pop_heap(value_type* c, size_type n);
```

Listing 5.8: Function contract of pop_heap

**Line 6** The property that only the elements of c are assigned is commented out here. It seems that adding this little bit overburdens the provers. However we were able to prove it separately by omitting other proof goals (Line 4, 8–10 in the specification in Listing 5.8, Line 3–4, 16–20, 47–49 in the implementation in Listing 5.11).

**Line 8** ensures that the remaining elements form a heap of length n - 1.

**Lines 9–10** ensure that the maximal element of the input heap is finally at the last position n - 1.

**Line 11** indicates the missing content-preservation specification.

### 5.4.2. Handling Induction in `pop_heap`

Before we show the implementation of `pop_heap` we want to discuss a problem that we encountered during verification.

It is obvious that, due to the heap property of the original array, the maximal element is at position `0`. However, if we take a closer look at the formalization of this fact, we see that the predicate `IsHeap(c,n)` in Listing 5.1 establishes facts like

```
c[0]  <=  c[1]
c[0]  <=  c[2]
c[1]  <=  c[3]  hence  c[0]  <=  c[3]
c[1]  <=  c[4]  hence  c[0]  <=  c[4]
c[2]  <=  c[5]  hence  c[0]  <=  c[5]
c[2]  <=  c[6]  hence  c[0]  <=  c[6]
 ...
```

While it is immediately obvious to a human that all these facts together imply the minimality of `c[0]`, an automated theorem prover is unable to

- derive all these facts,
- derive transitivity consequences from arbitrary-length (<=)-chains,
- observe their regular structure, i.e. that `c[0] <= c[i]` can eventually be derived for each `i < n`, and
- conclude that `\forall integer i; i < n ==> c[0] <= c[i]` holds.

All the more, it cannot do this in a single step, like humans can do. Rather, it needs to perform a proof by induction on `n` to establish the desired conclusion `\forall integer i; i < n ==> c[0] <= c[i].`

Although none of the provers employed by Frama-C is prepared to do induction proofs, we found a way to make them doing it, abusing Hoare's loop rule (cf. Sections 2.5 and 2.6) for that purpose.

We define a `C`-function `pop_heap_induction` that does not contribute to the functionality of `pop_heap`, but rather encapsulates the induction proof needed for its verification. This function will be called as ghost. The function contract of `pop_heap_induction`, shown in Listing 5.9, essentially **`requires`** our assumption, viz. `IsHeap(c,n)` and **`ensures`** our proof goal, viz. `\forall integer i; 0 <= i < n ==> c[0] >= c[i].`

In order to cause the prover to do an induction on `n`, we use a `for`-loop (Lines 3–13 in Listing 5.10) with a corresponding range and the induction hypothesis as **`loop invariant`**s. The loop body is empty, except for an additional hint to the prover in Line 12.

One can see in Listing 5.10 that a loop with an empty body makes perfectly sense. The assertion in Line 12 of Listing 5.10 is necessary to help the automatic provers.

```
/*@
  requires 0 < n && is_valid_range(c, n);
  requires IsHeap(c, n);

  assigns \nothing;

  ensures \forall integer i; 0 <= i < n ==> c[0] >= c[i];
*/
void pop_heap_induction(const value_type* c, size_type n);
```

Listing 5.9: Contract of ghost-function to perform induction

```
void pop_heap_induction(const value_type* c, size_type n)
{
  /*@
  loop   variant n - i;
  loop invariant 0 <= i <= n;

  loop invariant  \forall integer j;
     0 <= j < i <= n ==> c[0] >= c[j];
  */
  for (size_type i = 1; i < n; i++)
  {
    //@ assert 0 < i ==> ParentChild((i-1)/2, i);
  }
}
```

Listing 5.10: Implementation of the ghost-function to perform induction

### 5.4.3. Implementation of `pop_heap`

The implementation of `pop_heap` is shown in Listing 5.11 and is considerably more complex than that of `push_heap` (see Listing 5.7).

In `push_heap` we were traversing the binary heap tree bottom-up. Thereby every node had at most one parent, there was no choice. Now, in the case of `pop_heap` the tree traversing operates downwards and in every loop step the appropriate child must be chosen. Additionally, we include like the corresponding algorithm of the C++-STL an optimization concerning the number of comparisons which must be done in each loop step.

**Lines 3–4** perform the induction described in Section 5.4.2.

**Line 6** saves the maximum to put it finally to the position `n-1`.

**Lines 17–18** support the proof that the `max` is the largest one at end.

**Lines 19–20** and the assertion in **Lines 47–48** support the proof that the final dropping of `tmp` to the `hole`-position does not destroy the heap property.

**Lines 22–46** The loop tests in each step, whether `hole` is the right position for `tmp`. If not, it percolates down the position `hole` to the appropriate child-position and shifts upward the corresponding child element.

**Line 25** We test only, whether the larger child-position is smaller than `n-1`. If the answer is true the corresponding test for the smaller child is dispensable. This way we have in one loop step three comparisons only and not four. The case that a parent has exactly one child (the smaller one) can occur only at the loop end and is dealt with in **Lines 37–45**

**Line 34** The `break` is performed if `hole` is the right position for the value stored in `tmp`.

**Line 39** asks among others whether the smaller child is the only one. In this case it depends on its value whether a further step must be done.

**Line 44** The `break` exits the loop because here the right position for `tmp` must be found.

**Line 50** puts `tmp` that equals the former element `c[n-1]` to its final place `hole`.

**Line 51** puts `max` to the position `n - 1`.

```
void pop_heap(value_type* c, size_type n)
{
  //@ ghost pop_heap_induction(c, n);
  //@ assert \forall integer i; 0 < i < n ==> c[0] >= c[i];
  value_type tmp = c[n-1];
  value_type max  = c[0];

  size_type hole = 0;

  /*@
     loop invariant 0 <= hole < n;
     loop   variant n - hole;

     loop assigns c[0..hole];

     loop invariant IsHeap(c, n-1);
     loop invariant \forall integer i;
        0 <= i < n ==> c[i] <= max;
     loop invariant \forall integer i;
        hole < n-1 && ParentChild(i, hole) ==> c[i] >= tmp;
  */
  while (1)
  {
    size_type child = 2*hole + 2;
    if (child < n-1)
    {
      if (c[child] < c[child-1]) child--;
      //@ assert ParentChild(hole, child);
      if (c[child] > tmp)
      {
        c[hole] = c[child];
        hole = child;
      }
      else break;
    }
    else
    {
      child = 2*hole + 1;
      if (child == n-2 && c[child] > tmp)
      {
        c[hole] = c[child];
        hole = child;
      }
      break;
    }
  }
  /*@  assert \forall integer i;
    hole < n-1 && ParentChild(i, hole) ==> c[i] >= tmp;
  */
  c[hole] = tmp;
  c[n-1]  = max;
}
```

Listing 5.11: Implementation of pop_heap

Figure 5.6 illustrates an iteration step of the `pop_heap` function. The dashed arrows show what has been done before.

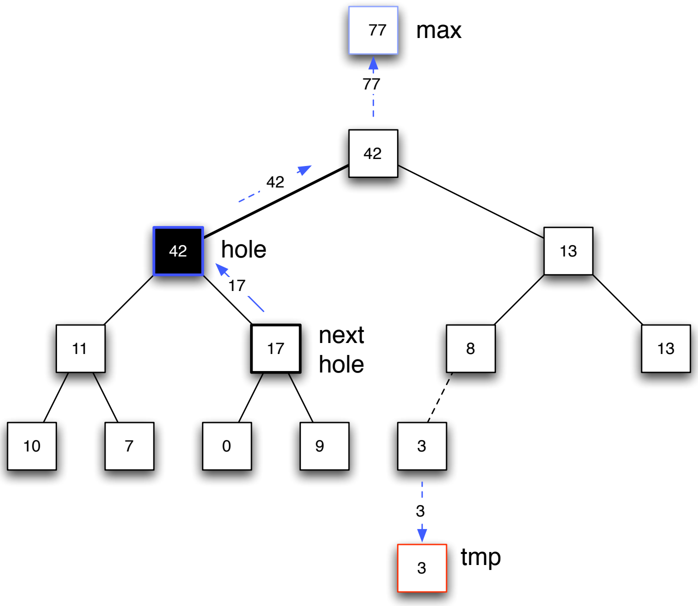
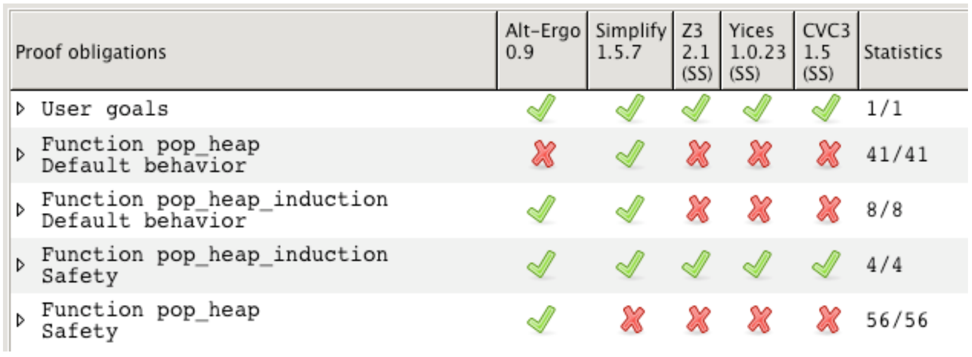


Figure 5.6.: An iteration step of `pop_heap`

### 5.4.4. Formal Verification of `pop_heap`

Figure 5.7 shows the results of the formal verification of `pop_heap` with five different automatic theorem provers. None of those provers was able to prove all verification conditions generated by Jessie. Again, Simplify was able to prove each verification condition except for the ones generated to examine arithmetic overflows in the "safety" group of Figure 5.7, which was proven by Alt-Ergo instead.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| User goals | ✓ | ✓ | ✓ | ✓ | ✓ | 1/1 |
| Function pop_heap Default behavior | ✗ | ✓ | ✗ | ✗ | ✗ | 41/41 |
| Function pop_heap_induction Default behavior | ✓ | ✓ | ✗ | ✗ | ✗ | 8/8 |
| Function pop_heap_induction Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 4/4 |
| Function pop_heap Safety | ✓ | ✗ | ✗ | ✗ | ✗ | 56/56 |

Figure 5.7.: Results of the formal verification of `pop_heap`

## 5.5. The make_heap Algorithm

Whereas in the STL make_heap works on a pair of generic random access iterators,[40] our version operators on a range of **value_type**. Thus the signature of make_heap reads

```
void make_heap(value_type* c, size_type n);
```

The function make_heap rearranges the elements of the given array c[0..n-1] such that they form a heap.

### 5.5.1. Formal Specification of make_heap

The ACSL specification of make_heap is shown in Listing 5.12. Line 7 indicates the missing content-preservation specification.

```
 1  /*@
 2     requires n < MAX_INT;
 3     requires is_valid_range(c, n);
 4  
 5     assigns c[0..(n-1)];
 6  
 7     ensures IsHeap(c, n);
 8     // ensures Permutation{Old, Here}(c, n);
 9  */
10  void make_heap(value_type* c, size_type n);
```

Listing 5.12: The specification of make_heap

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

### 5.5.2. Implementation of make_heap

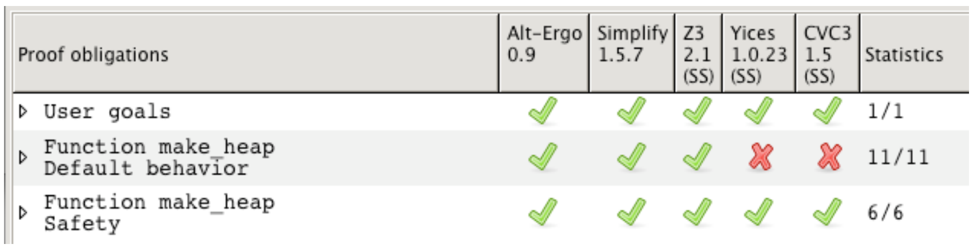
The implementation of make_heap, shown in Listing 5.13, is straightforward. From low to high the array's elements were pushed to the growing heap. The iteration starts with two, because an array with length one is a heap already.

```
void make_heap(value_type* c, size_type n)
{
  if (n == 0) return;
  /*@
     loop invariant IsHeap(c, i-1);
     loop invariant 2 <= i <= n+1;
     loop   variant n - i;
  */
  for (size_type i = 2; i <= n; i++)
    push_heap(c, i);
}
```

Listing 5.13: The implementation of make_heap

### 5.5.3. Formal Verification of make_heap

Figure 5.8 shows the results of the formal verification of make_heap with five different automatic theorem provers. The three provers Alt-Ergo, Simplify and Z3 were able to prove each of the generated verification conditions.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| User goals | ✓ | ✓ | ✓ | ✓ | ✓ | 1/1 |
| Function make_heap Default behavior | ✓ | ✓ | ✓ | ✗ | ✗ | 11/11 |
| Function make_heap Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 6/6 |

Figure 5.8.: Results of the formal verification of make_heap

## 5.6. The sort_heap Algorithm

The function `sort_heap` takes an array `c[0..n-1]` with the heap-property as input and sorts its elements in ascending order. Thus the signature of `sort_heap` reads

```
void sort_heap(value_type* c, size_type n);
```

Note that in the STL `sort_heap` works on a pair of generic random access iterators,[41]

### 5.6.1. Formal Specification of sort_heap

The ACSL specification of `sort_heap` is given in Listing 5.14. Line 8 indicates the missing content-preservation specification.

```
 1 /*@
 2    requires is_valid_range(c, n);
 3    requires IsHeap(c, n);
 4 
 5    assigns c[0..(n-1)];
 6 
 7    ensures \forall integer i; 0 <= i < n-1 ==> c[i] <= c[i+1];
 8    // ensures Permutation{Old, Here}(c, n);
 9 */
10 void sort_heap(value_type* c, size_type n);
```

Listing 5.14: The specification of `sort_heap`

### 5.6.2. Implementation of sort_heap

The implementation of `sort_heap` is shown in Listing 5.15.

The iteration of `sort_heap` operates from high to low with respect to the numbers `pop_heap` is called with.

**Line 7** states that the elements below `i` form a heap and

**Lines 9–10** state that the other elements including the `i`th are already sorted.

**Line 8** is the most interesting one. In the next iteration step the element `c[0]` will be put in front of `c[i]`. Therefore it is not allowed to be larger than `c[i]`.

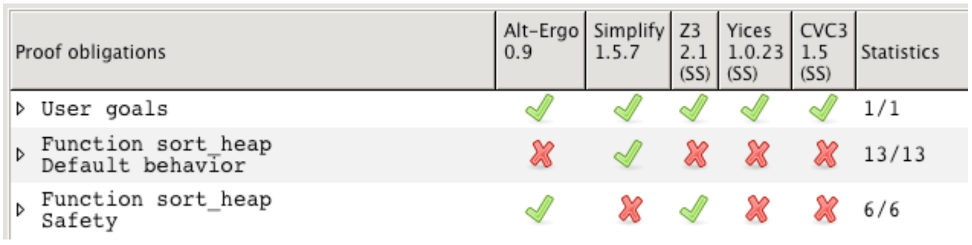
[41]See `http://www.sgi.com/tech/stl/sort_heap.html`

```
void sort_heap(value_type* c, size_type n)
{
  /*@
     loop invariant 0 <= i <= n;
     loop   variant i;

     loop invariant IsHeap(c, i);
     loop invariant i < n ==> c[0] <= c[i];
     loop invariant \forall integer j;
        i <= j < n-1 ==> c[j] <= c[j+1];
  */
  for (size_type i = n; i > 0; i--)
    pop_heap(c, i);
}
```

Listing 5.15: The implementation of `sort_heap`

### 5.6.3. Formal Verification of `sort_heap`

Figure 5.9 shows the results of the formal verification of `sort_heap` with five different automatic theorem provers. None of the provers was able to prove each generated verification condition. Simplify was able to prove each proof obligation except for the ones describing arithmetic overflows. However, this group of proof obligations was proven by Alt-Ergo and Z3 instead.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| User goals | ✓ | ✓ | ✓ | ✓ | ✓ | 1/1 |
| Function sort_heap Default behavior | ✗ | ✓ | ✗ | ✗ | ✗ | 13/13 |
| Function sort_heap Safety | ✓ | ✗ | ✓ | ✗ | ✗ | 6/6 |

Figure 5.9.: Results of the formal verification of `sort_heap`

## 5.7. The Sorting Algorithm `heap_sort`

Clearly, the concatenation of `make_heap` (Section 5.5) and `sort_heap` (Section 5.6) is a sorting algorithm that is commonly referred to as *heapsort*.[42]

Listing 5.16 shows the ACSL specification of `heap_sort`. Note that the postcondition in the comment in Line 7 is the only missing piece to achieve a complete specification of a sorting algorithm.

```
1  /*@
2     requires is_valid_range(c, n);
3
4     assigns c[0..(n-1)];
5
6     ensures \forall integer i; 0 <= i < n-1 ==> c[i] <= c[i+1];
7     // ensures Permutation{Old, Here}(c, n);
8  */
9  void heap_sort(value_type* c, size_type n);
```

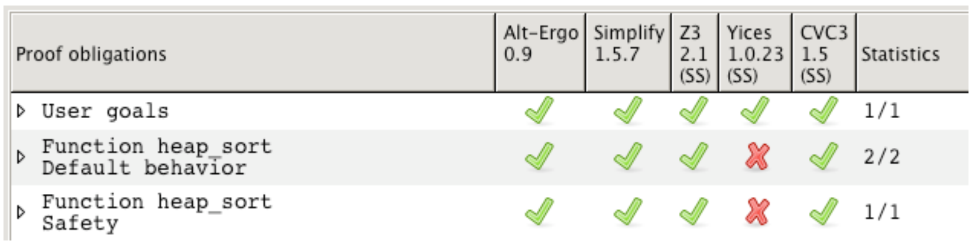
Listing 5.16: The specification of `heap_sort`

The implementation of `heap_sort` is shown in Listing 5.17.

```
void heap_sort(value_type* c, size_type n)
{
  make_heap(c, n);
  sort_heap(c, n);
}
```

Listing 5.17: The implementation of `heap_sort`

Figure 5.10 shows the results of the formal verification of `sort_heap` with five different automatic theorem provers. Alt-Ergo, Simplify and Z3 were able to prove all verification conditions generated by Jessie.



| Proof obligations | Alt-Ergo 0.9 | Simplify 1.5.7 | Z3 2.1 (SS) | Yices 1.0.23 (SS) | CVC3 1.5 (SS) | Statistics |
|---|---|---|---|---|---|---|
| User goals | ✓ | ✓ | ✓ | ✓ | ✓ | 1/1 |
| Function heap_sort Default behavior | ✓ | ✓ | ✓ | ✗ | ✓ | 2/2 |
| Function heap_sort Safety | ✓ | ✓ | ✓ | ✗ | ✓ | 1/1 |

Figure 5.10.: Results of the formal verification of `heap_sort`

[42]See `http://en.wikipedia.org/wiki/Heapsort`

# 6. Results from Deductive Verification

In this chapter we present to what extent the examples from Chapters 3 – 5 could be deductively verified using the Jessie plug-in of Frama-C and several automatic theorem provers. For these experiments we used the Beryllium2-20090902 release of Frama-C and the version 2.23 of the Why-Platform.

For each algorithm, the number of generated verification conditions are listed, as well as the percentage of proved verification conditions for each prover. All of the experiments were conducted on a Mac OS X running the Leopard operating system.

Table 6.1 shows the results of the individual provers for the non-mutating algorithms (Chapter 3). The automatic provers Alt-Ergo, Simplify, and Z3 were able to prove all verification conditions.

| | | | **Percentage of Proved VCs** | | | | |
|---|---|---|---|---|---|---|---|
| **Algorithm** | **Section** | **# VC** | **Alt-Ergo** | **CVC3** | **Simplify** | **Yices** | **Z3** |
| `equal` | 3.1 | 20 | 100 % | 100 % | 100 % | 100 % | 100 % |
| `mismatch` | 3.2 | 26 | 100 % | 92 % | 100 % | 77 % | 100 % |
| `equal` (`mismatch`) | 3.2 | 6 | 100 % | 100 % | 100 % | 100 % | 100 % |
| `find` | 3.3 | 24 | 100 % | 100 % | 100 % | 100 % | 100 % |
| `find` (reconsidered) | 3.4 | 24 | 100 % | 92 % | 100 % | 75 % | 100 % |
| `find_first_of` | 3.5 | 24 | 100 % | 79 % | 100 % | 75 % | 100 % |
| `adjacent_find` | 3.6 | 33 | 100 % | 100 % | 100 % | 91 % | 100 % |
| `max_element` | 3.7 | 42 | 100 % | 100 % | 100 % | 100 % | 100 % |
| `max_seq` | 3.8 | 5 | 100 % | 100 % | 100 % | 100 % | 100 % |
| `min_element` | 3.9 | 40 | 100 % | 100 % | 100 % | 100 % | 100 % |
| `count` | 3.10 | 28 | 100 % | 100 % | 100 % | 100 % | 100 % |

Table 6.1.: Results for non-mutating algorithms

Table 6.2 contains the results for the mutating algorithms. Here only the automatic prover Simplify was able to prove all verification conditions.

| | | | Percentage of Proved VCs | | | | |
|---|---|---|---|---|---|---|---|
| **Algorithm** | **Section** | **# VC** | **Alt-Ergo** | **CVC3** | **Simplify** | **Yices** | **Z3** |
| `swap` | 4.1 | 6 | 100 % | 67 % | 100 % | 100 % | 100 % |
| `swap_ranges` | 4.2 | 35 | 89 % | 80 % | 100 % | 77 % | 86 % |
| `fill` | 4.3 | 15 | 100 % | 100 % | 100 % | 87 % | 100 % |
| `copy` | 4.4 | 20 | 95 % | 85 % | 100 % | 80 % | 95 % |
| `replace_copy` | 4.5 | 32 | 88 % | 81 % | 100 % | 78 % | 94 % |
| `remove_copy` | 4.6 | 37 | 92 % | 89 % | 100 % | 81 % | 95 % |
| `remove_copy` (alternative) | 4.7 | 67 | 93 % | 87 % | 100 % | 66 % | 97 % |
| `iota` | 4.8 | 20 | 90 % | 85 % | 100 % | 80 % | 100 % |

Table 6.2.: Results for mutating algorithms

Table 6.3 contains the results for the heap algorithms. Some of the algorithms could not completely be proven by any automatic prover. For a more detailed analysis we refer to the respective sections in Chapter 5.

| | | | Percentage of Proved VCs | | | | |
|---|---|---|---|---|---|---|---|
| **Algorithm** | **Section** | **# VC** | **Alt-Ergo** | **CVC3** | **Simplify** | **Yices** | **Z3** |
| `is_heap` | 5.2 | 45 | 100 % | 84 % | 100 % | 84 % | 91 % |
| `push_heap` | 5.3 | 36 | 89 % | 64 % | 92 % | 75 % | 72 % |
| `pop_heap` | 5.4 | 110 | 95 % | 75 % | 96 % | 79 % | 93 % |
| `make_heap` | 5.5 | 18 | 100 % | 94 % | 100 % | 83 % | 100 % |
| `sort_heap` | 5.6 | 20 | 95 % | 70 % | 95 % | 75 % | 95 % |
| `heap_sort` | 5.7 | 4 | 100 % | 100 % | 100 % | 75 % | 100 % |

Table 6.3.: Results for heap algorithms

Simplify stood out in comparison to all of the other provers, since those had a higher percentage of unproven verification conditions. Examining the tables above, it is clear that all of the provers had fewer difficulties yielding positive results, when verifying non-mutating algorithms, in contrast to the mutating algorithms or even heap algorithms.

# A. Changes

## A.1. New in Version 4.2.1 (April 2010)

- added alternative specification of `remove_copy` algorithm that uses **`ghost`** variables (Section 4.7)
- added Chapter 5 on heap operations
- added `mismatch` algorithm (Section 3.2)
- 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` to `is_valid_range`

## A.2. New in Version 4.2.0 (January 2010)

- complete rewrite of previous release
- adaption to Frama-C Beryllium 2 release

## A.3. 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] Jessie Plug-in. `http://frama-c.com/jessie.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] Why – Software Verification Platform. `http://why.lri.fr`.

[6] Fraunhofer Institut Rechnerarchitektur und Softwaretechnik (FIRST). `http://www.first.fraunhofer.de`.

[7] Virgile Prevosto. ACSL Mini-Tutorial. `http://frama-c.com/download/acsl-tutorial.pdf`.

[8] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ANSI/ISO C Specification Language, Version 1.4 Frama-C Beryllium implementation. `http://frama-c.com/download/acsl-implementation-Beryllium-20090902.pdf`, Sep 2009.

[9] Standard Template Library Programmer's Guide. `http://www.sgi.com/tech/stl`, 2010.

[10] Programming languages – C, Committee Draft. `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1362.pdf`, 2009.

[11] Claude Marché and Yannick Moy. Jessie Plugin Tutorial, Beryllium Version. `http://frama-c.com/jessie/jessie-tutorial.pdf`, 2010.

[12] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Homepage of the Alt-Ergo Theorem Prover. `http://alt-ergo.lri.fr/`.

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

[14] Homepage of the Simplify Theorem Prover. `http://freshmeat.net/projects/simplifyprover/`, 2007.

[15] SRI International. Homepage of the Yices SMT Solver. `http://yices.csl.sri.com/`.

[16] Microsoft Reasearch. Homepage of the Z3 SMT Solver. `http://research.microsoft.com/en-us/um/redmond/projects/z3/`.

[17] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[18] 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.