Verificación de algoritmos en Escher

Verification of algorithms in Escher

Rafael Andrés Hernández Roldán

Ingeniería del Software. Facultad de Informática
Universidad Complutense de Madrid

Trabajo de Fin de Grado

Madrid, 2019

Directores:

Alberto Verdejo López
Isabel Pita Andreu

Abstract

This project borns from the growing need not only for quality programs but programs that never fail. Nowadays software is several orders of magnitude greater than just a few decades ago and critical software has become even more relevant, since if it fails, enormous, complex infrastructures fall with it, damaging machines, companies or even human lives.

When software testing is not enough, it is imperative to use formal verification of algorithms and data structures. Since it is a meticulous, large task, we need computer-aided program verification to help us to speed up and get confidence on the process. This project aims to assay and understand a tool created for this purpose: Escher C Verifier, made for enabling the development of formally-verifiable sofware in a subset of C. By verifying some known algorithms, we want to study its aptness as an academic tool.

Keywords

Escher C Verifier, algorithms, aided verification, formal verification, specification

Resumen

Este trabajo nace de la creciente necesidad no solo de programas de calidad, sino de programas que nunca fallen. Actualmente el software es varios órdenes de magnitud más grande que hace tan solo unas décadas y el software crítico se ha vuelto más importante aún, dado que si falla, enormes y complejas infraestructuras caen con él, dañando máquinas, empresas o incluso vidas humanas.

Cuando el testing de software no es suficiente, es necesario emplear la verificación formal de algoritmos y estructuras de datos. Puesto que es una tarea meticulosa y extensa, requerimos de la verificación asistida de programas para ayudarnos a acelerar y ganar confianza en el proceso. Este proyecto apunta a analizar y comprender una herramienta creada para este propósito: Escher C Verifier, hecha para permitir el desarrollo de software formalmente verificable en un subconjunto de C. A partir de la verificación de algunos algoritmos conocidos, queremos estudiar su aptitud como una herramienta académica.

Palabras clave

Escher C Verifier, algoritmos, verificación asistida, verificación formal, especificación

Contents

Algorithms

Chapter 1

Introduction

A few years ago, in the Computer Science Faculty (*Facultad de Informática, Universidad Complutense de Madrid*), the teachers of *Data Structures and Algorithms* began to use Dafny, a computer-aided program verifier to help the students in the process of understanding the utility of formal specifications in algorithms: have a way of being mathematically sure of their correctness. The process of verification is long and could be very difficult, since it is a new concept for these students, that must be applied to algorithms that require new techniques to be programmed. Dafny helps in this task and make it easier for the students and it is powerfull enough to verify advanced algorithms and data structures, as shown in *Verifying Algorithms and Data Structures in Dafny* [4].

Nevertheless, Dafny has one issue that students have found problematic: it uses its own programming language for the verifications. On the other hand, Escher C Verifier (that can be found at http://eschertech.com/products/ecv.php)., the tool we study in this project, uses a subset of the language C. It also can use the language C++, which is the language that students know at that point of the degree, so it is interesting for us to know if this tool can be taken to the classrooms, as an alternative to Dafny.

For this reason, the main objective of this project is to prove the competence of Escher C Verifier when verifying algorithms of this level, that we can find already specified and verified in *Especificación, Derivación y Análisis de Algoritmos* [7], *Programación metódica* [1] or *Programming: the derivation of algorithms* [5].

Since this is a new tool for us, we have had to make large use of its manual (http://eschertech.com/support/perfect_developer_self_help.php) [6] and the articles of the web page where we can find the Escher C Verifier [2, 3]. This material give advices for the first approach to this tool and some advanced algorithms.

Introducción

Hace unos pocos años, en la Facultad de Informática de la Universidad Complutense de Madrid, los profesores de *Estructura de Datos y Algoritmos* comenzaron a emplear Dafny, un programa de verificación asistida para ayudar a los estudiantes a la hora de comprender la utilidad de la especificación formal de algoritmos: obtener una manera de estar matemáticamente seguros de que son correctos. El proceso de verificar es largo y puede ser complicado, dado que es un nuevo concepto para estos estudiantes, que deben aplicar a algoritmos que requieren nuevas técnicas para ser programados. Dafny ayuda en esta tarea y la facilita a los estudiantes, y es suficientemente potente como para verificar algoritmos y estructuras de datos avanzadas, como se muestra en *Verifying Algorithms and Data Structures in Dafny* [4].

No obstante, Dafny tiene un inconveniente que los estudiantes han encontrado problemático: utiliza su propio lenguaje de programación para las verificaciones. Por otra parte, Escher C Verifier (que puede encontrarse en `http://eschertech.com/products/ecv.php`), la herramienta que utilizamos para este trabajo, utiliza un subconjunto del lenguaje C. También puede usar C++, que es el lenguaje que conocen los estudiantes en ese punto de la carrera, por lo que nos resulta interesante saber si esta herramienta puede ser llevada a las clases, como una alternativa a Dafny.

Por esta razón, el principal objetivo de este trabajo es probar la competencia de Escher C Verifier para verificar algoritmos de este nivel, que podemos encontrar especificados y verificados en *Especificación, Derivación y Análisis de Algoritmos* [7], *Programación metódica* [1] o *Programming: the derivation of algorithms* [5].

Dado que para nosotros es una herramienta nueva, hemos empleado enormemente su manual (`http://eschertech.com/support/perfect_developer_self_help.php`) [6], así como los artículos de la página web donde podemos encontrar Escher C Verifier [2, 3]. Este material guía en el primer acercamiento a esta herramienta y con algunos algoritmos avanzados.

Chapter 2

Knowing Escher

This chapter is about how to use the main features of the Escher C/C++ Verifier as we learn the basis to work with it. From some point of view, this section is a user guide, pretending to be lighter and more newcomer friendly, in order to make of Escher C/C++ Verifier a learning, academic tool for college students.

2.1 The eCv environment

The Escher C Verifier (or *eCv* for short) is -forgive the redundancy- a formal verifier created and supported by Escher Technologies. It was created to prove software correctness respect to a formal specification —given by a precondition, a postcondition, loop invariants and variants— provided by a designer. In particular, *eCv* proves programs written in the C language, although it has its twin brother Escher C++ Verifier (or *eCv++*) that extends its functionality for programs in C++. We have used this last one in this project.

The main advantage of using *eCv*/*eCv++* is that it is used directly over programs written in C/C++ instead of requiring a special syntax. Another advantage is that it provides a simple, although powerful specification language based on first order logic, complemented with many predefined predicates and functions.

In this project, we use the free version for study purposes and its main view is as follows.

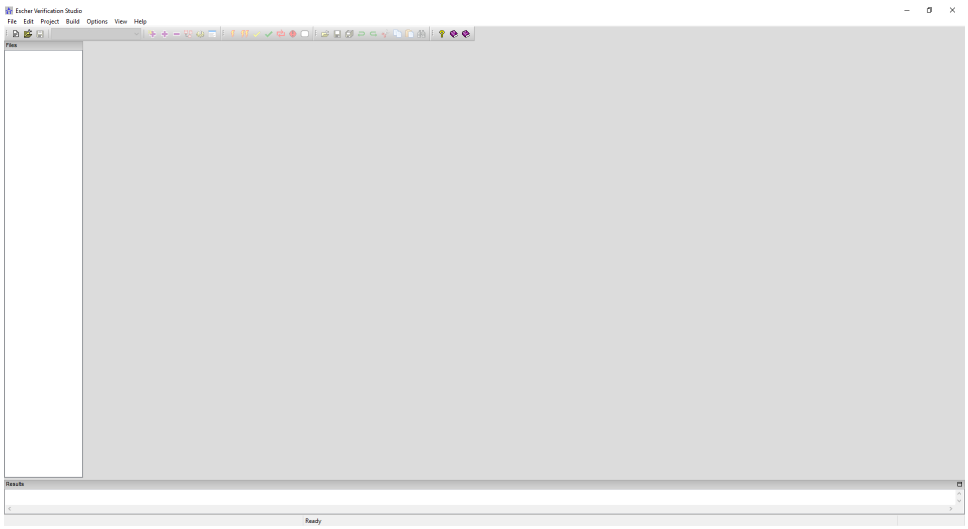

Figure 2.1: main view of eCv

To start using *eCv* we have to create a new project, by clicking **File** ⇒ **Newproject**.... Then, we must select the option "C or C++ files - create a new eCv project", click **Next** and select a name for the project and where it is going to be saved. Now, we must create a new source file clicking the button marked in green (the button at its right is for using some existing source file).

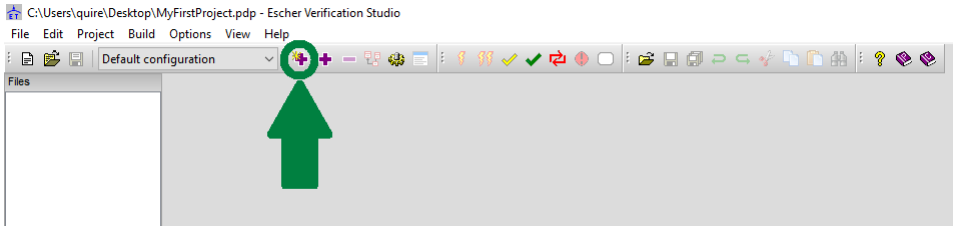

Figure 2.2: button for creating a new source file

After clicking the button we should see something like this:

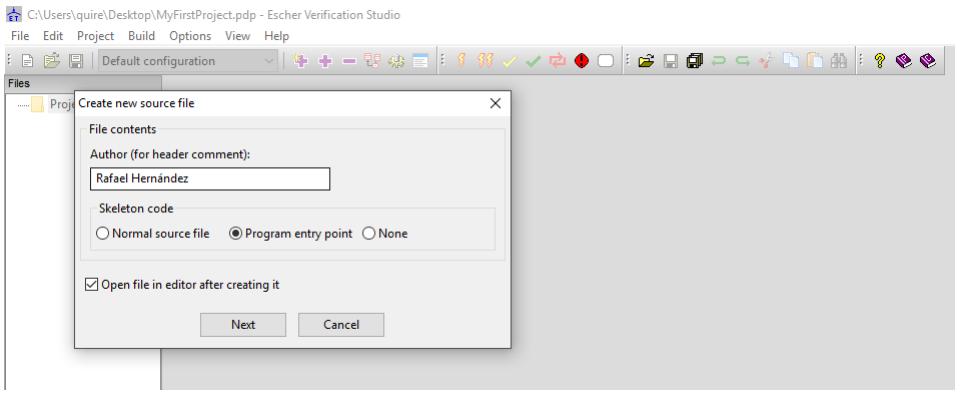

Figure 2.3: view for creating a new source file

We choose any options we want and we name and save the file (if no extension is explicitly indicated, then *eCv* will create a C file. For this example, we have created a C++ file adding its extension .cpp). Now we see the newly created file.

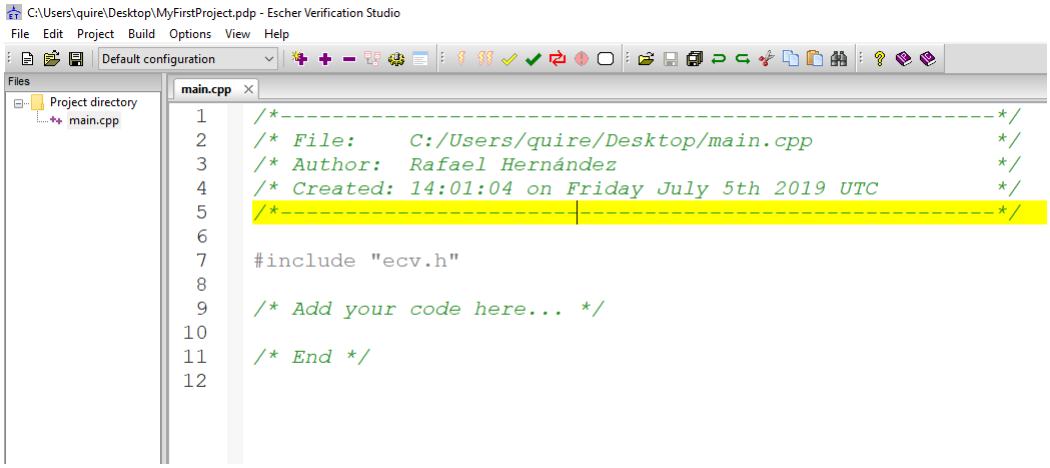

Figure 2.4: the new file

For verifying our algorithms we will need a compiler of C++ (two C compilers are already defined). It can be easily defined going to **Options** $\Rightarrow$ **C/C++compilers** $\Rightarrow$ **New/Copy**. In this view we have many options to set up our compiler. For verifying *eCv* programs we need their libraries, so we have to add them (normally located in the folder C:\Program Files\Escher Technologies\Verification Studio 7\Escher C Verifier).

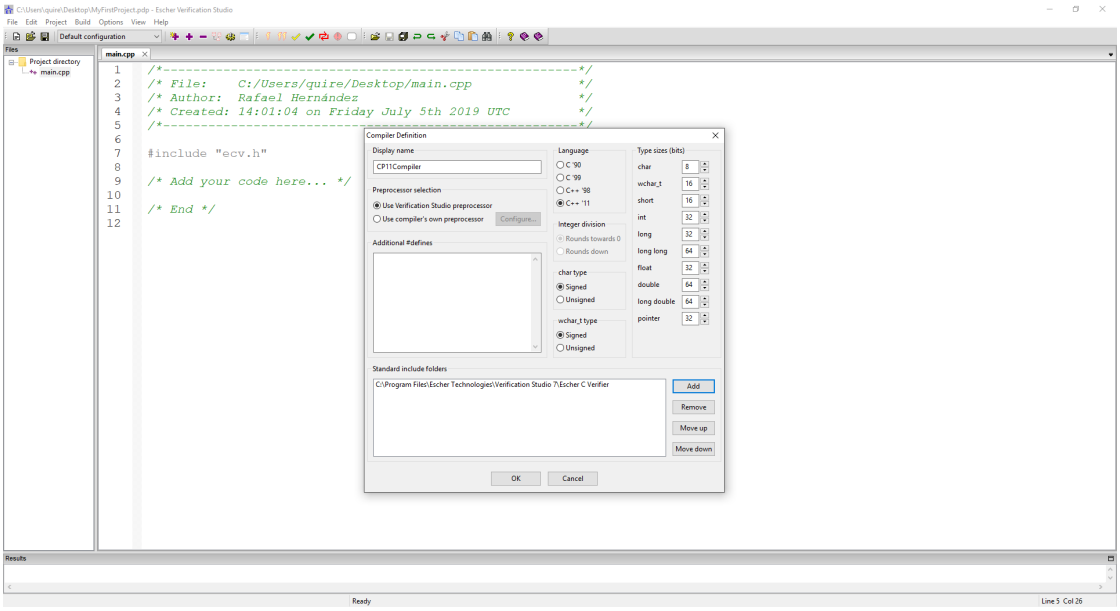

Figure 2.5: C++ '11 compiler definition

The last step is going to **Project** $\Rightarrow$ **Settings**... $\Rightarrow$ **C/C++compiler** and select our new compiler.

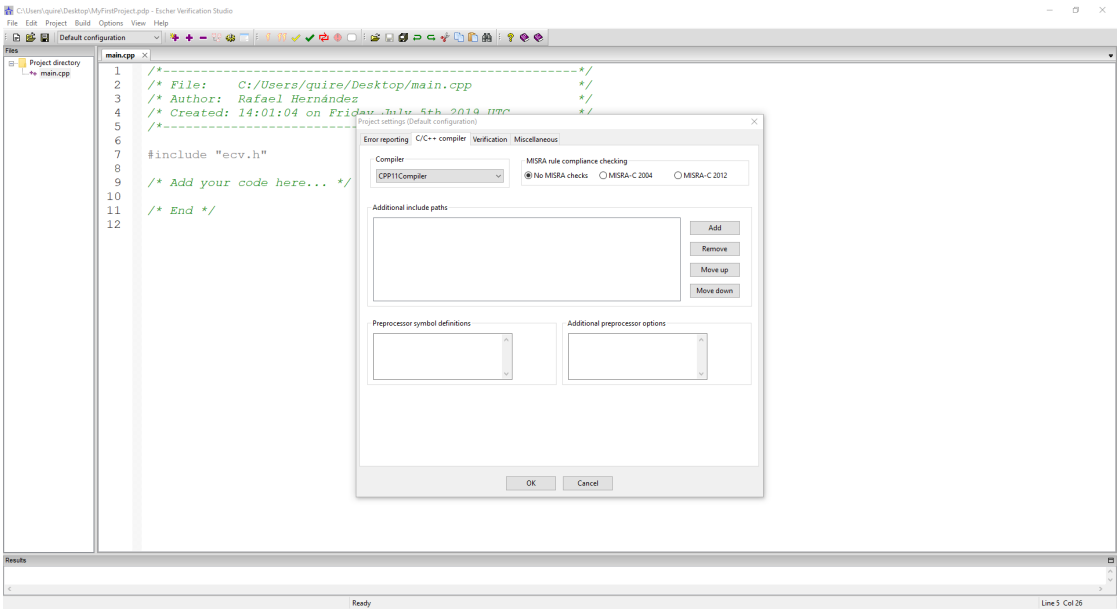

Figure 2.6: choosing the project compiler

Now, we can write a small program and use the buttons **Check** (the yellow mark) and Verify (the green one). Check checks if the code is syntactically correct and Verify checks

the syntax correctness and verifies the code based on its specification.

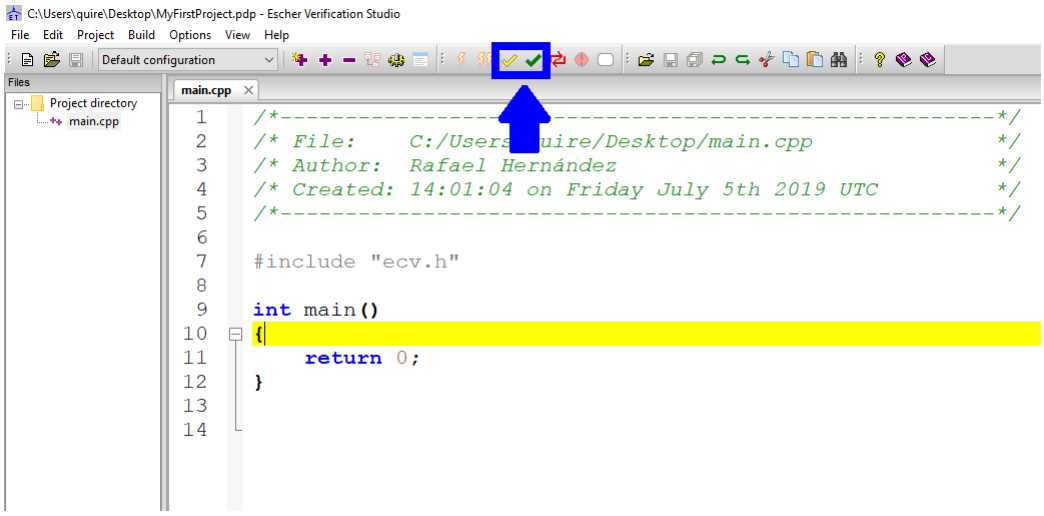

Figure 2.7: buttons Check and Verify

2.2 Our first algorithm

We begin with an algorithm that swaps the values of two variables.

The code should be quite easy to read, since, except for lines 4 - 8, is just a C/C++ program. The function receives the reference of two variables and interchanges the values using a local variable `aux`.

```
1  #include "ecv.h"
2
3  void swap(int& a, int& b)
4  post
5  (
6      a == old(b);
7      b == old(a)
8  )
9  {
10     int aux = b;
11     b = a;
12     a = aux;
13 }
```

Algorithm 2.1: Swap

The specification is given in `lines 4 - 8`. We observe two important words (coloured in red): `post` and `old`. The `post` clause gives the program several postconditions (separated with semicolons) to be fulfilled, that is, the conjunction of these postconditions must be true when the program is finished. Also, we can write postconditions by using several times

the post clause instead of semicolons. And we can talk about the return value (for not void functions) by using the result keyword or the returns clause (see Section 2.3 and Section 2.4, respectively). Finally, the old operator refers to the initial value of a parameter when the program started.

We find these keywords and many others at the library ecv.h, which is included at the first line of the program, as any other library in C/C++.

2.3 Verifying a loop

We illustrate the verification of a loop, taking as an example the algorithm that searches the maximum value of an array. The algorithm traverses the array from left to right, keeping in the variable max the maximum of the values already visited. A similar algorithm that traverses the array from right to left can be found in Algorithm 3.1

Four blocks are highlighted in the specification, corresponding to four keywords of *eCv*: pre, post, keep and decrease.

The pre clause is used to indicate preconditions (assertions we can assume at the beginning of the program) and has the same structure as the post clause. In this algorithm, we need three preconditions (we will see them along many other programs): the first one refers to the parameter n, which state, for the length of the array and should be greater than zero (n > 0). The other two statements refer to the array limits. The parameter n is the size of the array (v.lim == n), and the lowest valid index of the array equals zero (v.lwb == 0). lim and lwb are ghost fields defined by *eCv*, that is, fields that we can use at specifications, but not in the program. lim allows us to reference the array length and lwb the array lower bound.

The two first statements are quite intuitive, nevertheless the third one could be tricky. Since C++ allows negative indexes -which indicate relative memory positions- for arrays, is imperative to indicate that no negative indexes will be used on arrays. Most of ghost fields and ghost functions (see Section 2.4) need this to be established to be used.

The post clause has already been discussed in Section 2.2: it indicates postconditions to be satisfied. In this algorithm we use logical quantifiers to express that the result is greater than any other array value and that it is a value of the array. The expression "k in 0 .. (n - 1)" defines k in the range 0 to n - 1 (both inclusive). result is the returned value (the maximum value in the array).

Before the loop body, we find the block that corresponds to the invariant in the keep clause and the termination condition in the decrease clause. The invariant needs three statements (lines 16 - 18). The first one, for the range of the loop index. The second one, requires that the variable max, that keeps the maximum value of the already visited values should be greater than or equal to those values. And the third one states that the value of the

variable `max` is one of the values of the array.

This way, we can finally verify our program and see the "`Job completed with no problems detected.`" notification.

```
1  #include "ecv.h"
2
3  int searchMax(const int v[], const int n)
4  pre
5  (
6      n > 0; v.lim == n; v.lwb == 0
7  )
8  post
9  (
10     forall k in 0 .. (n - 1) :- result >= v[k];
11     exists k in 0 .. (n - 1) :- result == v[k]
12 )
13 {
14     int max = v[0];
15
16     for(int i = 1; i < n; ++i)
17     keep
18     (
19         1 <= i; i <= n;
20         forall k in 0 .. (i - 1) :- max >= v[k];
21         exists k in 0 .. (i - 1) :- max == v[k]
22     )
23     decrease(n - i)
24     {
25         if(v[i] > max){
26             max = v[i];
27         }
28     }
29
30     return max;
31 }
```

Algorithm 2.2: Search for the maximum value (first implementation)

2.4 Ghost functions

One of most interesting features of *eCv* are `ghost` functions: functions with specification but without implementation. They can be used in specification clauses such as preconditions, loop invariants, other ghost functions, etc.

eCv has some predefined ghost functions for common purposes. For example, we can use the ghost functions `take()` and `max()` to specify the algorithm that calculates the maximum value of an array.

Here we have used a `returns` clause to write the postcondition instead of the post clause. Although both clauses can be combined as long as `returns` goes first. `returns` can only specify what must be returned and —what is more interesting— is the only specifica-

tion clause that allows recursive calls. So, `returns(foo)` is an equivalent postcondition to `post(result == foo)`.

The ghost function `take()` allows us to express a selection of the first `n` elements of a `sequence`, skipping the rest and returning another `sequence` of those elements. A `sequence` is an abstract data type predefined by *eCv* as a class template. We transform an `array` data type of a program into a `sequence` using the *eCv* ghost field `all`.

We call these functions and fields "ghost" because they are defined only for specification purposes, so we can't use them in the source code.

As we can guess, the ghost function `max()` returns the maximum element of a `sequence`. Combining `take()` and `max()` our specification is easier to read.

```
1  #include "ecv.h"
2
3  int searchMax(const int v[], const int n)
4  pre(v.lim == n; v.lwb == 0; n > 0)
5  returns(v.all.max())
6  {
7      int max = v[0];
8
9      for(int i = 1; i < n; ++i)
10     keep
11     (
12         1 <= i; i <= n;
13         max == v.all.take(i).max()
14     )
15     decrease(n - i)
16     {
17         if(v[i] > max) {
18             max = v[i];
19         }
20     }
21
22     return max;
23 }
```

Algorithm 2.3: Search for the maximum value (with take() and max())

In addition to the predefined ghost functions, *eCv* allows us to define our own ghost functions. For example, we can define a ghost function with a recursive specification for searching the maximum value.

An important fact must be noticed in Algorithm Algorithm 2.4: if we call in the code to a function that has a parameter that is an array, *eCv* does not support the syntactic sugar notation `int v[]` for arrays. Then, it is necessary to denote an array as `const int* array v`. The `array` keyword is necessary because, otherwise, *eCv* interprets that the parameter is a pointer to an int, instead of a pointer to an array of ints.

```
1  #include "ecv.h"
2
3  ghost
4  (
```

```
5       int maxGhost(const int* array v, integer i)
6       pre(v.lim > 0; i >= 0; i < v.lim)
7       decrease(v.lim - i)
8       returns
9       (
10          i == (v.lim - 1) ?
11              v[v.lim - 1] : (v[i] > maxGhost(v, i + 1) ?
12                  v[i] : maxGhost(v, i + 1))
13      )
14  )
15
16  int searchMax(const int* array v, int n)
17  pre(n > 0; v.lim == n; v.lwb == 0)
18  returns(maxGhost(v, 0))
19  {
20      int max = v[n - 1];
21
22      for(int i = n - 1; i >= 0; --i)
23      keep
24      (
25          i in -1 .. (n - 1);
26          i < n - 1 => max == maxGhost(v, i + 1);
27          i == n - 1 => max == v[n - 1]
28      )
29      decrease(i)
30      {
31          if(v[i] > max){
32              max = v[i];
33          }
34      }
35
36      return max;
37  }
```

Algorithm 2.4: Search for the maximum value (with user defined ghost function)

2.5 Verifying recursive programs

Once we have seen the potential of *eCv* with iterative algorithms, it is natural to wonder if it will be possible to verify so easily a recursive program. Again, searching for the maximum value will be an illustrative example.

For the specification of this algorithm we have to define the limits of the control variable `i` in the precondition (`line 7`). Also, we need a recursion variant (in order to prove recursion termination). Just as seen with loop variants, the clause is `decrease` and, in the case of recursive functions, it is placed after the precondition clause.

In the definition of the invariant we use the ghost function: `drop()`. Like the `take()` function, `drop()` selects part of a `sequence`, but in this case ignoring the first `n` elements of the `sequence`. The use of `drop()` gives us an elegant invariant, in the same way `take()` did in Algorithm 2.3.

Again, we can not use syntatic sugar for defining the array since a recursive call that uses the array is made.

```
#include "ecv.h"

int searchMax(const int* array v, const int n, const int i)
pre
(
    v.lim == n; v.lwb == 0; n > 0;
    0 <= i; i < n
)
decrease(n - i)
returns(v.all.drop(i).max())
{
    if (i == n - 1) {
        return v[n - 1];
    } else {
        int  max = searchMax(v, n, i + 1);
        return max > v[i] ? max : v[i];
    }
}
```

Algorithm 2.5: Search for the maximum value (a recursive implementation)

Chapter 3

Iterative algorithms

In this chapter we find several iterative algorithms and their specifications in *eCv* . The algorithms are ordered attending to their difficulty, except for the last section, beginning with algorithms that iterate once over an array and ending with nested loops and iterations over 2D arrays.

3.1 Search algorithms

The algorithms of this section iterate over an array and search for a specific value in it (given or not).

3.1.1 Searching from right to left

This is another implementation of the algorithm that searches the maximum value of a collection, which was introduced in Section 2.3. In this case, the algorithm begins with the last element (what we call right side of the array) and ends with the first one (left side). With this new version of the algorithm we want to show the differences between the verification process of the algorithm that traverses an array from left to right and the one that goes from right to left. In Algorithm 3.1 we use the specification with predefined functions used in Algorithm 2.3.

We observe in the specification of the algorithm that the precondition and postcondition clauses are the same as those defined in Section 2.3. This must be so since the specification of an algorithm does not depend on its implementation.

```
#include "ecv.h"

int searchMax(const int v[], const int n)
pre(n > 0; v.lim == n; v.lwb == 0)
returns(v.all.max())
{
    int max = v[n - 1];

    for (int i = n - 2; i >= 0; --i)
    keep
    (
        -1 <= i; i <= n - 2;
        max == v.all.drop(i + 1).max()
    )
    decrease(i)
    {
        if(v[i] > max) {
            max = v[i];
        }
    }

    return max;
}
```

Algorithm 3.1: Search from right to left

3.1.2 Searching and counting the maximum value

Now that we have improved the art of finding the maximum value of an array, we proceed to count how many times it appears in the collection. We implement an algorithm that performs the search and count processes in the same loop.

From this program we can highlight the use of:

- the `struct` declaration of C++.
- The `writes` keyword, to indicate to *eCv* what nonlocal variables the function modifies [6]. Alternatively, if we have several loops in a function and they all overwrite some parameters we can indicate it in the top of the function, before the precondition. This allows all loops to overwrite the parameters, instead of indicating it for each one. However, if this keyword is used, then we must include all nonlocal variables that are going to be modified.
- The `those element in sequence :- condition` expression returns a sequence of the elements in another sequence that fulfill a condition (in this case, the condition is that they are the maximum value) [6].
- The `count` ghost field of a `sequence`, which gives the number of its elements [6].

```
#include "ecv.h"

struct sol{
    int val;
    int counter;
};
// Saves the maximum of v on result.val and returns
// the times it appears on result.counter
sol maxAndCount(const int v[], const int n)
pre
(
    v.lwb == 0; v.lim == n; n > 0
)
post
(
    result.val == v.all.max();
    result.counter ==
        (those x in v.all :- x == result.val).count
)
{
    sol s = {v[0], 1};

    for (int i = 1; i < n; ++i)
    writes(i; s)
    keep
    (
        0 < i; i <= n;
        s.val == v.all.take(i).max();
        s.counter ==
            (those x in v.all.take(i) :- x == s.val).count
    )
    decrease (n - i)
    {
        if (v[i] > s.val) {
            s.val = v[i];
            s.counter = 1;
        } else if (v[i] == s.val) {
            ++s.counter;
        }
    }

    return s;
}
```

Algorithm 3.2: Search and count

3.1.3 Sequential search of a value in an unordered array

Knowing how to find the maximum it seems natural to take the step to a sequential search of an element. The algorithm returns the position of the first occurrence of the element `x` in the array or the size of the array if it is not found.

We notice that the function of Algorithm 3.3 doesn't use syntactic sugar. It is needed since this function is called from Algorithm 3.4.

The postcondition of Algorithm 3.3 requests that all the elements to the left of `result` in the array are not the searched element (`line 8`) and if `result` is not the size of the array, then the element was found (`line 9`). We observe that the first statement of the postcondition (`line 7`) requires the returned value to be between 0 and the size of the array, including both values. Since we require the first occurrence, the second statement (`line 8`) requires that the returned position of the array corresponds to the first occurrence of the value, if any, since all the values on the left of this position are different from the searched value. Finally, the third statement (`line 9`) requires that if the returned value is an index of the array then the value of that array position is the required value.

```
1  #include "ecv.h"
2
3  int seqSearch(const int* array v, const int n, const int x)
4  pre (v.lwb == 0; v.lim == n)
5  post
6  (
7      result in 0 .. n;
8      forall k in 0 .. (result - 1) :- v[k] != x;
9      result < n => v[result] == x
10 )
11 {
12     int pos = 0;
13     while (pos < n && v[pos] != x)
14     writes(pos)
15     keep
16     (
17         pos in 0 .. n;
18         forall k in 0 .. (pos - 1) :- v[k] != x
19     )
20     decrease (n - pos)
21     {
22         ++pos;
23     }
24
25     return pos;
26 }
```

Algorithm 3.3: Find the position of a value in an array

We illustrate the use of a function specification in the verification process of an algorithm in Algorithm 3.4. When calling a function, all the parameters must satisfy the precondition of the function and it is ensured that when the function ends, all output parameters and the returned value (if any) satisfy the postcondition and those statements are taken by *eCv* for verifying the rest of the algorithm. This algorithm checks if a given value belongs to an array, by using Algorithm 3.3. For this reason, we don't include the `ecv.h` header, but the C++ file `main.cpp`. In this file we find the algorithm with the sequential search.

In addition, we notice that extra help is necessary to complete the proof: the `assert` clause forces to check if some assertion is true at the point of the program where it is written and, if it is, *eCv* can use that information onwards.

```
#include "main.cpp"

bool elementExists(const int v[], const int n, const int x)
pre(v.lwb == 0; v.lim == n)
returns(exists y in v.all :- y == x)
{
    int pos = seqSearch(v, n, x);
    assert((pos == n) => !(exists y in v.all :- y == x));
    return pos < n;
}
```

Algorithm 3.4: Find if a value exists in an array

3.1.4 Iterative binary search

In order to complete the search algorithms, we take the iterative binary search algorithm from David Crocker's articles [3].

The precondition ensures (apart from the usual limits) that the array is ordered from the lowest value to the highest. The postcondition ensures that all the values on the left of the `result` position are smaller than the `key` and all the values on the right of the `result` position are greater than the `key`.

It is important to notice that this program forces `n` to be less than or equal to the half of `maxof(int)` and `low + high` to be less than or equal to `n`. This is done to avoid a type overflow caused by the sum of two ints. For example, if the searched element is `v[n - 1]` and the size of `v` is a pair number, the program will try to sum `n + n` sooner or later; if `n` is greater than the half of `maxof(int)`, then the result will be greater than `maxof(int)` (see Section 3.2 for more examples on type overflow).

```
#include "ecv.h"

int binarySearch(const int v[], const int n, const int key)
pre
(
    v.lwb == 0; v.lim == n;
    n <= maxof(int) / 2;
    forall a in v.indices; b in v.indices :- b > a => v[b] > v[a]
)
post
(
    result in 0 .. n;
    forall element in v.all.take(result) :- key >= element;
    forall element in v.all.drop(result) :- key <= element
)
{
    int low = 0, high = n;

    while(high != low)
    writes(high; low)
    keep
```

```
22      (
23          high in 0 .. n;
24          low in 0 .. high;
25          low + high <= maxof(int);
26          forall element in v.all.take(low) :- element <= key;
27          forall element in v.all.drop(high) :- element >= key
28      )
29      decrease(high - low)
30      {
31          const int mid = (low + high)/2;
32          if (key >= v[mid]){
33              low = mid + 1;
34          } else {
35              high = mid;
36          }
37      }
38
39      return low;
40  }
```

Algorithm 3.5: Binary search (iterative)

3.2 Type overflow

eCv checks that there is no overflow in the variables used by the program. We use an algorithm that computes the sum of the elements of an array to illustrate how to specify the limits of the variable values.

eCv provides a summation expression by means of the `over` keyword, which applies a binary operation to a sequence. We use the `+ over` expression to sum the elements of the array.

Next, we have to consider that a summation of `n` elements could lead into some overflow due to type limits. While other settings may be satisfactory, we succeeded returning an `int` and using an *array of shorts* with no more elements than the maximum value of a `short`.

We may think that it will be enough with making `int` all variables and parameters (except `v`, that would be a `const int v[]`). Nevertheless, if `n` is an `int` and `v` an *array of ints*, then (taking into account that `n` is the size of the array), we can receive an array of $2^{31} - 1$ (`maxof(int)`) elements, all with a value of $2^{31} - 1$. This sum has a result of $2^{62} - 2^{32} - 1$, far away of the int limit and we will get several warnings, all of them very similar, saying something like "*Warning! Unable to prove: Arithmetic result of operator '+' is within limit of type 'int' (see PATH:\main_unproven.html#10), did not prove: minof(int) <= (v[heapIs $heap $funcstart_204,1$, i$loopstart_214,5$] + sum$loopstart_214,5$). Suggestion: Add extra 'keep' expression: (i < n) => (-2147483648 <= (v[i] + (+ over (*v.$r).take(i) .ranb())))*". On the other hand, our solution in Algorithm 3.6 can sum up to $(2^{15} - 1) \cdot (2^{15} - 1) = 2^{30} - 2^{16} + 1$, what fits in the `int` limit. Of course, *eCv* makes the respective proofs for the lower limit.

```
#include "ecv.h"

int summation(const short v[], const short n)
pre(v.lwb == 0; v.lim == n)
returns(+ over v.all)
{
    int sum = 0;

    for (short i = 0; i < n; ++i)
    keep
    (
        i in 0 .. n;
        sum == + over v.all.take(i)
    )
    decrease(n - i)
    {
        sum += v[i];
    }

    return sum;
}
```

Algorithm 3.6: Sum of the elements of an array

Still, there's another way to solve this problem: indicating the limits of every operation as precondition. We will use it in several programs, as the one below. Indicating the limits of the operation `* over v.all.take(i)` (at `line 9`), we give to the user the responsibility of being sure that no type overflow will be produced.

```
#include "ecv.h"

int seqProduct(const int v[], const int n)
pre
(
    v.lim == n; v.lwb == 0;
    forall i in 1 .. n
    :- * over v.all.take(i) in minof(int) .. maxof(int)
)
post(result == * over v.all)
{
    int res = 1;

    for (int i = 0; i < n; ++i)
    keep
    (
        i in 0 .. n;
        res == (* over v.all.take(i))
    )
    decrease(n - i)
    {
        res = res * v[i];
    }

    return res;
}
```

Algorithm 3.7: Product of the elements of an array

3.3 Rewriting an array

This section is about algorithms that change the values of the elements of an array. These algorithms have two main issues to be solved: to indicate that the elements will be changed and how they will be.

3.3.1 Convert the elements of an array to absolute values

In this program, we use several concepts already explained such as ghost functions and the `old` expression. As a novelty, we must pay attention at the `writes` clause. We are rewriting the values of the array `v`, so we must specify that all of them (`v.all`) can be modified. If not, we will get an error even before verifying that says: "*Error! This loop requires an explicit writes-clause because its body modifies a value through a pointer.*".

The loop invariant requires special attention: we have three assessments to be kept. The first one (`line 24`) is just the range of the loop control variable, but the second one (`line 25`) checks that all the elements of the array from that index to the right have not been changed and the last one (`line 26`) checks that every element from the left to the index (not included) is the absolute value of the original element.

```
1  #include "ecv.h"
2
3  ghost
4  (
5      int ghostAbs(const int n)
6      pre(n != minof(int))
7      returns(n >= 0 ? n : -n)
8  )
9
10 void abs(int v[], const int n)
11 pre
12 (
13     v.lwb == 0;
14     v.lim == n;
15     forall x in v.all :- x != minof(int)
16 )
17 post(forall k in v.indices :- v[k] == ghostAbs(old v[k]))
18
19 {
20     for (int i = 0; i < n; ++i)
21     writes (i; v.all)
22     keep
23     (
24         i in 0 .. n;
25         forall k in i .. (n - 1) :- v[k] == old v[k];
26         forall k in 0 .. (i - 1) :- v[k] == ghostAbs(old v[k])
27     )
28     decrease (n - i)
```

```
29      {
30          if (v[i] < 0) {
31              v[i] = - v[i];
32          }
33      }
34  }
```

Algorithm 3.8: Convert the elements to absolute values

3.3.2 Product of two arrays

Here we implement an algorithm that multiplies all the elements in the same position of two arrays, resulting in other array. In this case, we must take care of the type overflow problem on the result of the product. The function receives two input arrays (`v` and `w`) and an output array (`x`). In `lines 9 - 10` of the precondition are indicated the limits of the operation `v[k] * w[k]`.

```
1   #include "ecv.h"
2
3   void productOfVectors(const int v[], const int w[],
4       long x[], const int n)
5   pre
6   (
7       v.lwb == 0; w.lwb == 0; x.lwb == 0;
8       v.lim == n; w.lim == n; x.lim == n;
9       forall k in v.indices :-
10          v[k] * w[k] in minof(int) .. maxof(int)
11  )
12  post(forall k in v.indices :- x[k] == v[k] * w[k])
13  {
14      for (int i = 0; i < n; ++i)
15      writes(x.all; i)
16      keep
17      (
18          i in 0 .. n;
19          forall k in 0 .. (i - 1) :- x[k] == v[k] * w[k]
20      )
21      decrease(n - i)
22      {
23          x[i] = v[i] * w[i];
24      }
25  }
```

Algorithm 3.9: Product of two arrays

3.3.3 Reverse an array

In this section we can find two different ways of reverse all the elements of an array.

The first one copies the elements of the array into another one. This is the "simple" solution. However we must be careful with memory sharing, since both `v` and `w` can point

to the same elements. The `disjoint` expression let us specify what we need, because is an expression that returns `true` if the storage associated with their parameters does not overlap. As the `disjoint` expression is in the precondition, the program requires both arrays to be disjointed, so it can be assumed onwards.

```
#include "ecv.h"

void revArray(const int v[], int w[], const int n)
pre
(
    v.lim == n; w.lim == n;
    v.lwb == 0; w.lwb == 0;
    n > 0;
    disjoint(v.all, w.all)
)
post
(
    forall k in 0 .. (n - 1) :- w[k] == v[n - 1 - k]
)
{
    for(int i = 0; i < n; ++i)
    writes(w.all; i)
    keep
    (
        i in 0 .. n;
        forall k in 0 .. (i - 1) :- w[k] == v[n - 1 - k]
    )
    decrease(n - i)
    {
        w[i] = v[n - 1 - i];
    }
}
```

Algorithm 3.10: Reverse an array in a different one

For the second version we only need one array that we are going to modify. For the specification, we make use of `rev()`, a predefined ghost function that returns a `sequence` of the elements of the array in reverse order.

```
#include "ecv.h"

void revArray(int v[], const int n)
pre(v.lim == n; v.lwb == 0)
post(v.all == old v.all.rev())
{
    for(int i = 0; i < n / 2; ++i)
    writes(v.all; i)
    keep
    (
        i in 0 .. (n / 2);
        forall k in i .. (n - i - 1) :- v[k] == old v[k];
        forall k in 0 .. (i - 1)
        :- v[k] == old v[n - k - 1]
            && v[n - k - 1] == old v[k]
    )
    decrease(n - i)
```

```
18      {
19          int aux = v[i];
20          v[i] = v[n - i - 1];
21          v[n - i - 1] = aux;
22      }
23  }
```

Algorithm 3.11: Reverse an array in the array itself

3.4 Checking properties of the array elements

Here we verify several algorithms that check if the elements of an array have some properties, like equality, order or some that we define.

3.4.1 A property for each element: equality

Now the algorithm that checks if all the elements of an array are equal to a given value `X`. Moreover, we will try to be more efficient by stopping the loop when the program finds an element different from `X` (if it happens).A 2D array version of this algorithm can be found at Algorithm 3.19.

```
1   #include "ecv.h"
2
3   bool allEqual(int const v[], const int n, const int X)
4   pre(v.lim == n)
5   returns(forall j in 0 .. (n - 1) :- (v[j] == X))
6   {
7       int i = 0;
8       bool ok = true;
9
10      while (i < n && ok)
11      keep
12      (
13          i in 0 .. n;
14          ok == (forall j in 0 .. (i - 1) :- v[j] == X)
15      )
16      decrease(n - i)
17      {
18          ok = v[i] == X;
19          ++i;
20      }
21
22      return ok;
23  }
```

Algorithm 3.12: All array elements are equal

3.4.2 The elements are in increasing order

Specifications about order are large, as seen in Algorithm 3.5 and it's not surprising that *eCv* provides a ghost function for this: `isndec()` [6]. This function is true if and only if all the `sequence` is in non decreasing order, that is, if the sequence is ordered from low to high.

The invariant requires that when the algorithm begins the loop, one of these conditions must be true (in fact, the invariant allows both to be true, but this can't happen):

- the array has no elements (`n == 0`), so the the result is true.
- the first `i` elements are ordered. If `n` is equal to 0, *eCv* shows a warning (*"Warning! Unable to prove: Loop initialisation establishes loop invariant (defined at PATH:\main.cpp (12,11)), did not prove: i in v.indices."*), since is part of the precondition of the ghost function `take()` that the parameter must be greater than 0 (for this reason, both conditions can't be true at the same time).

```
1  #include "ecv.h"
2
3  bool ordered(const int v[], const int n)
4  pre
5  (   v.lim == n;
6      v.lwb == 0;
7      n >= 0
8  )
9  returns(v.all.isndec())
10 {
11     bool res = true;
12
13     for(int i = 0; res && i < n - 1; ++i)
14     keep
15     (
16         (n == 0 && res)
17         ||
18         (i in 0 .. (n - 1) && res == v.all.take(i + 1).isndec())
19
20     )
21     decrease(n - i)
22     {
23         res = v[i] <= v[i + 1];
24     }
25
26
27     return res;
28 }
```

Algorithm 3.13: The elements are in increasing order

3.4.3 Peaks of an array

We can check other properties defined by ourselves using our own ghost functions.

We define a peak of an array as an element such that all the elements before it in the array are less or equal to it. The program below counts how many of these elements the array has. For this purpose, it is not enough to keep the count, we also have to save the index of the max of the part of the array already explored (`line 28`).

This invariant requires that:

- `i` is an `int` between 0 and `n` (`line 19`).
- `imax` is the the index of the actual max (`lines 20 - 21`).
- `count` is the number of peaks so far (`line 22`).

```
1  #include "ecv.h"
2
3  ghost
4  (
5      bool isPeak(const int* array v, const int i)
6      pre(i in v.indices)
7      returns(forall k in 0 .. (i - 1) :- v[k] <= v[i])
8  )
9
10 int peaks(const int* array v, const int n)
11 pre(v.lwb == 0; v.lim == n; n > 0)
12 returns((those x in v.indices :- isPeak(v, x)).count)
13 {
14     int imax = 0; int count = 1;
15     for (int i = 1; i < n; ++i)
16     writes (imax; count; i)
17     keep
18     (
19         i in 0 .. n;
20         imax in 0 .. (i - 1);
21         forall k in 0 .. (i - 1) :- v[imax] >= v[k];
22         count == (those x in 0 .. (i - 1) :- isPeak(v, x)).count
23     )
24     decrease(n - i)
25     {
26         if (v[i] >= v[imax]) {
27             imax = i;
28             ++count;
29         }
30     }
31     return count;
32 }
```

Algorithm 3.14: Peaks of an array

3.4.4 Positive partial sums and total zero

The following algorithm checks if all the partial sums in an array are positive or zero, and the sum of all the elements is zero. This is a good example of how we can need different ghost functions in different parts of the specification and verification process.

```
1
2  #include "ecv.h"
3
4  ghost
5  (
6      bool positivePartialSum(const short* array v, const short i)
7      pre(v.lwb == 0; i < v.lim)
8      returns(forall k in 0 .. i :- + over v.all.take(k + 1) >= 0)
9  )
10
11 ghost
12 (
13     bool ghostPPSAndsumZero(const short* array v)
14     pre(v.lwb == 0; v.lim <= maxof(short))
15     returns(+ over v.all == 0 && positivePartialSum(v, v.lim - 1))
16 )
17
18 bool PPSAndsumZero(const short v[], const short n)
19 pre
20 (
21     v.lim == n; v.lwb == 0;
22     n > 0
23 )
24 returns(ghostPPSAndsumZero(v))
25 {
26     long sum = v[0];
27     bool ok = v[0] >= 0;
28
29     for(short i = 1; i < n && ok; ++i)
30     keep
31     (
32         i in 1 .. n;
33         sum == + over v.all.take(i);
34         ok == positivePartialSum(v, i - 1)
35     )
36     decrease(n - i)
37     {
38         sum += v[i];
39         ok = ok && sum >= 0;
40     }
41
42     return ok && sum == 0;
43 }
```

Algorithm 3.15: Positive partial sums and total zero

3.5 Sort algorithms

In this section we specify and verify two well-known sort algorithms, both with quadratic performance complexity: bubble sort and selection sort.

3.5.1 Bubble sort

The algorithm below sorts the array elements using the bubble sort method. Neither the precondition nor the postcondition should be unknown for us. The precondition defines the typical limits and the postcondition requires that the array is ordered (by using the predefined `isndec()` function, that we have already seen in Algorithm 3.13).

This is the first time we face nested loops: we have to define an invariant, as well as `writes` and `decrease` clauses for the external loop and others for the internal loop.

The external invariant (`lines 9 - 14`) indicates the limits of the loop control variable `i` and requires that all the processed elements are greater than those to their left (that is, they are in increasing order).

For the internal invariant (`lines 19 - 25`) we indicate: the limits of the internal loop control variable, that all the elements to the right of the current element are greater than it and we repeat part of the external invariant. Repeating the external invariant is needed because, since this loop is inside the other, it is forced to keep it. However, we observe that it is not necessary to repeat the limits of the variable `i`. The reason is in `line 18`: the variable `i` is not overwritten in this loop. Consequently, it must keep the limits indicated above. If this algorithm had overwritten `i`, then, the limits should have been indicated again.

```
1  #include "swap.cpp"
2
3  void bubblesort(int v[], const int n)
4  pre(n == v.lim; v.lwb == 0)
5  post(v.all.isndec())
6  {
7      for(int i = 0; i < n - 1; ++i)
8      writes(i; v.all)
9      keep
10     (
11         n > 0 => i in 0 .. (n - 1);
12         forall k in 0 .. (i - 1); t in k .. (n - 1)
13         :- v[k] <= v[t]
14     )
15     decrease(n - i)
16     {
17         for (int j = n - 1; j > i; --j)
18         writes(j; v.all)
19         keep
20         (
21             j in i .. (n - 1);
22             forall t in j .. (n - 1) :- v[j] <= v[t];
23             forall k in 0 .. (i - 1); t in k .. (n - 1)
24             :- v[k] <= v[t]
25         )
26         decrease(j)
```

```
27          {
28              if (v[j] < v[j - 1]) {
29                  swap(v[j], v[j - 1]);
30              }
31          }
32      }
33  }
```

Algorithm 3.16: Bubble sort

3.5.2 Selection sort

This is a version of the selection sort algorithm, where the part of the array that is being ordered is on the right side of the array.

The specification of this algorithm is exactly the same as the specification of the bubble sort, illustrating once more that the specification does not depend on the implementation of the algorithm.

The verification of selection sort is very similar to the one of bubble sort and has the same difficulties, but once we achieve one of them, we can get the other in small time, changing mainly the indices of the invariant. The external invariant requires that every visited element, has to be greater than or equal to the elements on its left (`lines 13 - 14`). The internal invariant requires the same condition as the external invariant and that the current element to be sorted (`v[n - 1 - i]`) is greater than or equal to the elements on its left.

```
1   #include "ecv.h"
2
3   void selectionsort(int v[], const int n)
4   pre(v.lim == n; v.lwb == 0)
5   post(v.all.isndec())
6   {
7       for(int i = 0; i < n - 1; i++)
8       writes(i; v.all)
9       keep
10      (
11          n > 0 => i in 0 .. (n - 1)
12                   &&
13                   (forall k in (n - i) .. (n - 1)
14                   :- v[k] >= v.all.take(k).max())
15      )
16      decrease(n - i)
17      {
18          for(int j = 0; j < n - 1 - i; j++)
19          writes(j; v.all)
```

```
20          keep
21          (
22              j in 0 .. (n - 1 - i);
23              forall k in (n - i) .. (n - 1)
24                  :- v[k] >= v.all.take(k).max();
25              j > 0 => v.all.take(j).max() <= v[n - 1 -i]
26          )
27          decrease(n - j)
28          {
29              if (v[j] > v[n - 1 - i]) {
30                  int aux = v[j];
31                  v[j] = v[n - 1 - i];
32                  v[n - 1 - i] = aux;
33              }
34          }
35      }
36  }
```

Algorithm 3.17: Selection sort

3.6 Multidimensional arrays

We call a "multidimensional" array an array of arrays or an array of other multidimensional arrays. In this section we use 2D arrays, which are arrays of arrays. The next examples illustrate the verification of nested loops. They define invariant and decrease clauses for both the external and the internal loops.

3.6.1 Initialize a 2D array

In this algorithm we initialize the elements of a 2D array with a given value. The first issue we need is to define the 2D array itself. Once again, the syntactic sugar of C/C++ can't be used (`int[][]`), so we define the 2D array with help from the alias `tVector`.

The function receives a 2D array with `n` rows and `m` columns, while `X` is the given value to initialize the elements of the 2D array.

We notice that not only the limit and the lower bound of the 2D array must be defined (let's remember that it is still an array), but all the limits and lower bounds of every `tVector` in the 2D array. In other words, we must define the limit and the lower bound of every array and multidimensional array.

As we did for an array in the sorting algorithms, we must indicate that all the elements of the 2D array will be overwritten. We do this with the `2Darray.all` declaration in the `writes` clauses of the loops. Also, part of the invariant of the external loop needs to be repeated in the internal loop, as we saw in Algorithm 3.16.

The postcondition requires that all the elements of the 2D array must be equal to `X`,

so the external loop invariant ensures that for every row, all the previous rows are equal to `X`; and the inner loop invariant ensures that for every element of a row, all the previous elements are equal to `X`.

```
#include "ecv.h"

const int N = 10000;
typedef int tVector[N];

void initialize(tVector* array array2D, const int n, const int m,
    const int X)
pre
(
    n == array2D.lim; array2D.lwb == 0;
    forall k in array2D.indices
    :- (array2D[k].lwb == 0 && m == array2D[k].lim)
)
post
(
    forall k in array2D.indices
    :- (forall t in array2D[k].indices :- array2D[k][t] == X)
)
{
    for (int i = 0; i < n; ++i)
    writes (i; array2D.all)
    keep
    (
        i in 0 .. n;
        forall k in 0 .. (i - 1)
        :- (forall t in array2D[k].indices :- array2D[k][t] == X)
    )
    decrease (n - i)
    {
        for (int j = 0; j < m; ++j)
        writes (j; array2D.all)
        keep
        (
            j in 0 .. m;
            forall k in 0 .. (i - 1)
            :- (forall t in array2D[k].indices
                :- array2D[k][t] == X);
            forall t in 0 .. (j - 1) :- array2D[i][t] == X
        )
        decrease(m - j)
        {
            array2D[i][j] = X;
        }
    }
}
```

Algorithm 3.18: Initialize a 2D array

3.6.2 All 2D array elements are equal

This algorithm checks if all the elements of a 2D array are equal to a given value. Here we find another way of defining a 2D array: using the common syntax of an array in *eCv* twice.

This way, `int* array` is an array and `int* array* array` is an array of arrays (that is, a 2D array). We prefer this way because the maximum number of rows doesn't depend on a constant defined in the source code as it does in Algorithm 3.18.

The rest of the specification is very similar to the specification of Algorithm 3.18.

```
#include "ecv.h"

bool allEqual(const int* array* array array2D, const int n,
    const int m, const int X)
pre
(
    m >= 0; array2D.lwb == 0; n == array2D.lim;
    forall row in array2D.all :- (row.lwb == 0 && row.lim == m)
)
post
(
    result  =>
    (
        forall row in array2D.all
        :- (forall elem in row.all :- elem == X)
    )
)

{
    bool ok = true;
    for(int i = 0; i < n; ++i)
    writes(i; ok)
    keep
    (
        i in 0 .. n;
        ok =>
        (
            forall k in 0 .. (i - 1)
            :- (forall elem in array2D[k].all :- elem == X)
        )
    )
    decrease(n - i)
    {
        for (int j = 0; j < m; ++j)
        writes(j; ok)
        keep
        (
            j in 0 .. m;
            ok =>
            (
                forall k in 0 .. (i - 1)
                :- (forall elem in array2D[k].all :- elem == X)
            );
            ok => (forall t in 0 .. (j - 1) :- array2D[i][t] == X)
        )
        decrease(m - j)
        {
            if (array2D[i][j] != X) {
                ok = false;
            }
        }
    }
```

```
53
54      return ok;
55  }
```

Algorithm 3.19: All 2D array elements are equal

3.7 Other iterative algorithms

3.7.1 Fibonacci

This is an algorithm that calculates an element of the Fibonacci sequence. Its specification requires defining a ghost function, since we have to ensure the system that all the operations will fit in ints. For this reason, we indicate in the precondition:

- that the required element fits in an `int`.
- that any sum of two of the previous elements fits in an `int`. This is needed because *eCv* can't verify inductive properties. So, even if `ghostFib(x) + ghostFib(x)` fits in an `int`, *eCv* has no way to know that `ghostFib(x - 1) + ghostFib(x - 2)` fits in an `int` too.

Recursive calls in the specification are only allowed in a `returns` clause. Then, in order to express the value of an element of the Fibonacci sequence we use the ghost function `ghostFib()`.

To finish the proof, it is required that the variable `i` is smaller than or equal to the maximum value of an `int`. We use an `assume` clause to avoid its verification, so *eCv* takes it for granted.

```
1
2   #include "ecv.h"
3
4   ghost
5   (
6       integer ghostFib(const integer n)
7       pre(n >= 0)
8       decrease(n)
9       returns(n <= 1 ? 1 : ghostFib(n - 1) + ghostFib(n - 2))
10  )
11
12  int fibonacci(const int n)
13  pre
14  (
15      n >= 0;
16      forall i in 0 .. n
17          :- ghostFib(i) in minof(int) .. maxof(int);
18      forall i in 0 .. (n - 1) ; j in 0 .. (n - 2)
```

```
        :- (ghostFib(i) + ghostFib(j)
        in minof(int) .. maxof(int))
)
decrease(n)
returns(ghostFib(n))
{
    if (n <= 1) {
        return 1;
    }

    int n1 = 1, n2 = 1, f = 2;
    for (int i = 2; i < n; ++i)
    keep
    (
        i in 2 .. n;
        n2 == ghostFib(i - 2);
        n1 == ghostFib(i - 1);
        f == ghostFib(i)
    )
    decrease(n - i)
    {
        n2 = n1;
        n1 = f;
        f = n1 + n2;

        assume(i < maxof(int));
    }

    return f;
}
```

Algorithm 3.20: Fibonacci

3.7.2 Is prime number

This algorithm checks if a given number is prime, by checking if it is divisible by any number from 2 to itself (excluding the last one). We give the definition of a prime number in a ghost function.

```
#include "ecv.h"

ghost
(
    bool ghostIsPrime(const int n)
    pre(n >= 0)
    returns
    (
        n >= 2
        &&
        (forall k in 2 .. (n - 1) :- n % k != 0)
    )
)

bool isPrime(const int n)
pre(n >= 0)
returns(ghostIsPrime(n))
```

```
18 {
19     bool prime = n >= 2;
20
21     for(int i = 2; prime && i < n; ++i)
22     keep
23     (
24         n >= 2 => i in 2 .. n;
25         n >= 2 => prime ==
26             (forall k in 2 .. (i - 1) :- n % k != 0);
27         n < 2 => !prime
28         )
29         decrease(n - i)
30         {
31             prime = n % i != 0;
32         }
33
34     return prime;
35 }
```

Algorithm 3.21: Is prime number

3.7.3 Greatest common divisor

Here we make an algorithm to calculate the greatest common divisor (gcd) of two positive integer numbers. We define the gcd by using the Euclidean recursive algorithm for the gcd, defined in a ghost function. Then, we make an iterative implementation of the algorithm.

We treat `b` as the smallest value between the two given numbers. If not, we just swap both `a` and `b` (`lines 21 - 25`).

In the invariant, we indicate some known properties (`lines 32 - 34`) of the gcd. We could combine `line 32` and `line 33` using a logical disjunction, but we prefer to divide them because it is easier to read and because *eCv* makes the disjunction internally anyway. But, if some of the properties are not fulfilled, then *eCv* indicates what statement is not being fulfilled. If this happens with the statements combined, then we don't know what of both statements is not being fulfilled.

```
1  #include "ecv.h"
2
3  ghost
4  (
5      unsigned gcdGhost(const unsigned a, const unsigned b)
6      pre(a >= b)
7      decrease(b)
8      returns(b == 0 ? a : gcdGhost(b, a % b))
9  )
10
11 unsigned gcd(unsigned a, unsigned b)
12 decrease(b)
13 post
14 (
15     old a >= old b => result == gcdGhost(old a, old b);
16     old a <=  old b => result == gcdGhost(old b, old a)
```

```
)
{
    unsigned aux;

    if (a < b) {
        aux = a;
        a = b;
        b = aux;
    }

    while(b != 0)
    keep
    (
        a >= 0;
        b in 0 .. a;
        b != 0 => gcdGhost(a, b) == gcdGhost(b, a % b);
        b != 0 => gcdGhost(old a, old b) == gcdGhost(b, a % b);
        b == 0 => gcdGhost(old a, old b) == a
    )
    decrease(b)
    {
        aux = a;
        a = b;
        b = aux % b;
    }

    return a;
}
```

Algorithm 3.22: Greatest common divisor

Chapter 4

Recursive algorithms

In this chapter we show the verification of some recursive algorithms. We have chosen the implementation of two mathematical functions and the binary search to illustrate the process.

4.1 Mathematical functions

4.1.1 The factorial

Like in the Algorithm 3.20, the algorithm for calculating the factorial needs a ghost function for defining the limits of the element calculated by the function.

```
1  #include "ecv.h"
2
3  ghost
4  (
5      integer ghostFactorial(const integer n)
6      pre(n >= 0)
7      decrease(n)
8      returns(n == 0 ? 1 : n * ghostFactorial(n - 1))
9  )
10
11 int factorial(const int n)
12 pre
13 (
14     n >= 0;
15     ghostFactorial(n) in minof(int) .. maxof(int)
16 )
17 decrease(n)
18 returns(ghostFactorial(n))
19 {
20     if (n == 0) {
21         return 1;
22     }
```

```
23
24      return n * factorial(n - 1);
25  }
```

Algorithm 4.1: Factorial

4.1.2 Pow

This algorithm calculates the exponentiation from a number to another by using a divide-and-conquer strategy and the *eCv* own definition of pow (`_ecv_pow`).

```
1   #include "ecv.h"
2
3   int pow(const int base, const int exp)
4   pre
5   (
6       exp >= 0;
7       base != 0 || exp != 0;
8       base != 0 =>
9           (forall i in 0 .. exp :-
10          base _ecv_pow i in minof(int) .. maxof(int))
11  )
12  decrease(exp)
13  returns(base _ecv_pow exp)
14  {
15      int res;
16
17      if (exp == 0) {
18          res = 1;
19      } else if (base == 0) {
20          res = 0;
21      } else {
22          res = pow(base, exp / 2);
23          res = res * res;
24
25          if (exp % 2 != 0) {
26              res = res * base;
27          }
28      }
29
30      return res;
31  }
```

Algorithm 4.2: Pow

4.2 Binary search

We already verified an iterative implementation of the binary search in Algorithm 3.5, but this time we make it recursive.

The precondition expresses:

- limits for the parameters (`lines 7 - 9`).
- the sum of any couple of numbers from `left` to `right` is not going to exceed `maxof(int)` (`line 10`). It is not necessary to do the same for `minof(int)` because all these numbers have to be greater or equal to zero.
- the array is ordered (`line 11`).
- the discarded part on the left is less than the `key` (`line 12`).
- the discarded part on the right is greater than the `key` (`line 13`).

On the other hand, the postcondition expresses that `result` has to be one of the array indices (`line 18`) and that either `v[result]` is the searched value (`line 19`) or all the elements from the beginning until (`result - 1`) are less than the `key` (`line 21`) and all the element from (`result + 1`) until the end (`line 22`), meaning that the `key` wasn't found.

```
1  #include "ecv.h"
2
3  int binarySearch(const int* array v, const int left,
4      const int right, const int key)
5  pre
6  (
7      v.lwb == 0;
8      right in v.indices;
9      left in 0 .. right;
10     forall i in left .. right :- i + right <= maxof(int);
11     forall i in v.indices; j in i .. (v.lim - 1) :- v[i] <= v[j];
12     forall i in 0 .. (left - 1) :- v[i] < key;
13     forall i in (right + 1) .. (v.lim - 1) :- key < v[i]
14 )
15 decrease(right - left)
16 post
17 (
18     result in v.indices;
19     (v[result] == key)
20     ||
21     ((forall i in 0 .. (result - 1) :- v[i] < key)
22         && (forall i in (result + 1) .. (v.lim - 1) :- key < v[i]))
23 )
24 {
25     int mid = (right + left) / 2;
26
27     if(left == right) {
28         return mid;
29     } else if (v[mid] == key) {
30         return mid;
31     } else if (v[mid] < key) {
32         return binarySearch(v, mid + 1, right, key);
33     } else {
34         return binarySearch(v, left, mid, key);
35     }
36 }
```

Algorithm 4.3: Binary search (recursive)

Chapter 5

Conclusions

The main goal of this project was to prove the aptitude of *eCv* to be used as an alternative tool to Dafny by the students of the *Data Structures and Algorithms* course of the *Facultad de Informática* of the *Universidad Complutense*. Its main advantage regarding Dafny (verifying algorithms in a programming language already known by the students) has been checked. Although it can't use part of the syntactic sugar that we are used to in C and C++, at the level the language is used in the subject, there are no significant differences between the subset used by *eCv* and what students use during the course, so it will facilitate its approach.

Moreover, it's a tool that allows to verify algorithms of the complexity of a basic course, even deeper than Dafny, like checking the overflow in numerical operations. This can make the verification more difficult, especially at the beginning, but for being truly formal, these checks must be taken in care.

In addition, besides *eCv* sometimes offers little descriptive messages about the problems that it is having when making a verification, once got some practice, they are very helpful and enough to understand the errors and inaccuracies that we may have committed when making a proper formal specification for the algorithm we are addressing. This way, it makes it easier both the specification and the verification.

As a disadvantage regarding Dafny, *eCv* doesn't allow the use of lemmas and can't use induction, what makes impossible to prove some properties, making necessary to use other system where making the demonstration of these properties, for giving them assumed in *eCv* .

eCv doesn't have extensive documentation either. In fact, usually it feels scarce. This forces to make hundreds of proves to understand basic features of the tool, that finally could be summarized in a line or two.

Initially, we would have liked to verify of data structures, but because of the lack of

experience with *eCv* it hasn't been possible to try this feature that it offers.

In brief, *eCv* shows big potential and can be of big help when making formal verification.

Conclusiones

El principal objetivo de este trabajo era comprobar la aptitud de *eCv* para ser empleado como herramienta alternativa a Dafny por los estudiantes del curso de *Estructuras de Datos y Algoritmos* de la Facultad de Informática de la Universidad Complutense. Su principal ventaja con respecto a Dafny (verificar algoritmos en un lenguaje de programación ya conocido por los alumnos) ha quedado probada. A pesar de que carece de parte del azúcar sintático al que estamos familiarizados en C y C++, al nivel al que es empleado el lenguaje en la asignatura, no existen diferencias significativas entre el subconjunto de estos lenguajes empleado por *eCv* y lo que usan los alumnos en el transcurso de la asignatura, por lo que facilitará su acercamiento.

Por otra parte, es una herramienta que permite verificar algoritmos de la complejidad de un curso básico, e incluso en mayor profundidad que Dafny, como las comprobaciones de desbordamiento entre operaciones numéricas. Esto puede dificultar las verificaciones, especialmente al principio, pero para ser verdaderamente formales, estas comprobaciones han de ser tenidas en cuenta.

Adicionalmente, si bien *eCv* ofrece en ocasiones mensajes poco descriptivos sobre los problemas que está teniendo a la hora de realizar una verificación, una vez se obtiene un poco de práctica, son de mucha ayuda y suficientes para comprender los distintos errores o imprecisiones que hayamos podido cometer a la hora de realizar una especificación formal apropiada del algoritmo que estemos tratando. De este modo, se facilita tanto la especificación como la verificación.

Como desventaja respecto a Dafny, *eCv* no permite el uso de lemas ni es capaz de emplear la inducción, lo que hace imposible que pueda probar ciertas propiedades, siendo necesario emplear otro sistema en el que realizar la demostración de estas propiedades, para darlas asumidas a *eCv* .

eCv tampoco disfruta de una extensa documentación. De hecho, a menudo se siente muy escasa. Esto obliga a realizar cientos de pruebas para entender características básicas de la herramienta, que finalmente se podrían haber resumido en una línea o dos.

Inicialmente, se hubiese deseado realizar también verificaciones sobre estructuras de datos, pero dada la falta de experiencia con *eCv* no ha sido posible probar esta característica que ofrece.

En resumen, *eCv* muestra un enorme potencial y puede ser de gran ayuda a la hora de realizar verificaciones formales.

Bibliography

[1] José Luis Balcázar. *Programación Metódica*. Madrid [etc]: McGraw-Hill, D. L., 2003.

[2] David Crocker. Miscellaneous. http://eschertech.com/articles/index.php.

[3] David Crocker. Proving C and C++ programs correct. http://eschertech.com/articles/index.php.

[4] Rubén Rafael Rubio Cuéllar. *Verificación de algoritmos y estructuras de datos en Dafny*. Universidad Complutense de Madrid, 2016.

[5] Anne Kaldewaij. *Programming: the derivation of algorithms*. New York: Prentice-Hall, 1990.

[6] Escher Technologies Ltd. Escher C/C++ verifier 7.0: Reference manual. http://eschertech.com/product_documentation/ecvReference/eCv_Manual.pdf, 2017.

[7] Narciso Martí Oliet, Clara María Segura Díaz, J. A. Verdejo López. *Especificación, derivación y análisis de algoritmos: ejercicios resueltos*. Madrid: Pearson/Prentice Hall, 2006.