

From Graphical Loop Invariants to Formal Verification

Auteur : Grosjean, Antoine

Promoteur(s) : Fontaine, Pascal; Donnet, Benoît

Faculté : Faculté des Sciences appliquées

Diplôme : Master : ingénieur civil en informatique, à finalité spécialisée en "management"

Année académique : 2024-2025

URI/URL : <http://hdl.handle.net/2268.2/23376>

Avertissement à l'attention des usagers :

Tous les documents placés en accès ouvert sur le site le site MatheO sont protégés par le droit d'auteur. Conformément aux principes énoncés par la "Budapest Open Access Initiative"(BOAI, 2002), l'utilisateur du site peut lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces documents, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale (ou prévue par la réglementation relative au droit d'auteur). Toute utilisation du document à des fins commerciales est strictement interdite.

Par ailleurs, l'utilisateur s'engage à respecter les droits moraux de l'auteur, principalement le droit à l'intégrité de l'oeuvre et le droit de paternité et ce dans toute utilisation que l'utilisateur entreprend. Ainsi, à titre d'exemple, lorsqu'il reproduira un document par extrait ou dans son intégralité, l'utilisateur citera de manière complète les sources telles que mentionnées ci-dessus. Toute utilisation non explicitement autorisée ci-avant (telle que par exemple, la modification du document ou son résumé) nécessite l'autorisation préalable et expresse des auteurs ou de leurs ayants droit.

From Graphical Loop Invariants to Formal Verification



Montefiore Institute of Electrical Engineering and Computer Science
Faculty of Applied Sciences
University of Liège

Master's thesis carried out to obtain the degree of Master of Civil
Engineering in Computer Science

by Antoine Grosjean

Academic supervisors : Pr. Pascal Fontaine
 Pr. Benoît Donnet

Academic year 2024-2025

Abstract

Logic and mathematics are essential for understanding and verifying computer programs, yet many first-year computer science students struggle with these subjects.

This Master's thesis represents a first step toward automating the verification of simple C programs without requiring students to master formal logic. It demonstrates the feasibility of developing and implementing a tool that uses Loop Graph Invariants to verify computer programs written in C.

To this end, a comprehensive process was developed and tested on ten programming problems covering the content of the introductory computer programming course taught at the University of Liège. All benchmarks were eventually verified.

The methodology implemented relies on using the Dafny programming and verification language. It also involves a code translator that converts a C program into a program following the Dafny syntax, as well as an Embedding Model for French developed by La Javaness, used to extract the semantics of French sentences.

Although the process is not fully automated, it paves the way for future research on the subject and the eventual realization of a complete system.

Keywords: Verification, Invariant, Dafny, Loop, C, Graphical Loop Invariant, Predicate, Automation, Logic, Correctness.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my two supervisors, Professors Benoit Donnet and Pascal Fontaine. I am deeply grateful for their assistance and availability throughout this project. Their insightful comments and advice significantly improved the final outcome of this work, and I am thankful to them.

Thanks also to Ms. Géraldine Brieven for providing documents that allowed me to base this work on exercises that are truly representative of the programming problems students face.

I would also like to thank my girlfriend. For over a year and a half, she listened to me talk about the problems I encountered, watched me work through entire evenings, and supported me at every critical moment during the completion of this work and throughout the rest of my studies. Your support has meant everything to me.

Thanks also to my mom, dad, sister, and friends, who have supported me since the beginning of my studies. Without them, I certainly would not be where I am today. Thank you for being there for me. You mean more to me than I could ever express.

Finally, I am grateful to everyone who has helped, supported, or advised me in one way or another.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 1.1 | Context | 1 |
| 1.2 | Motivation and Objective | 1 |
| 1.3 | Overview | 2 |
| 2 | Theoretical Background | 3 |
| 2.1 | Loops and Invariants | 4 |
| 2.1.1 | Formal Loop Invariant | 4 |
| 2.1.2 | Graphical Loop Invariant | 9 |
| 3 | Programming Languages and Software | 15 |
| 3.1 | Dafny | 16 |
| 3.1.1 | Overview | 16 |
| 3.1.2 | Basics of Dafny Utilization | 16 |
| 3.1.3 | Example of Dafny Program | 22 |
| 3.2 | GLIDE | 22 |
| 3.2.1 | User Interface | 24 |
| 3.2.2 | Blank GLI | 24 |
| 4 | Methodology and Process | 26 |
| 4.1 | Programming Patterns | 28 |
| 4.1.1 | Assessing Properties for Sequences of Numbers | 28 |
| 4.1.2 | Operations on Array | 29 |
| 4.1.3 | Manipulating the Digits of a Number | 30 |
| 4.1.4 | Drawing on the Terminal | 31 |
| 4.1.5 | Common Specificities | 32 |

| | | |
|----------|---|-----------|
| 4.2 | Parsing GLI | 33 |
| 4.2.1 | Structure Type | 33 |
| 4.2.2 | Bars | 34 |
| 4.2.3 | Text Zones | 36 |
| 4.2.4 | Automation with JSON Parsing | 37 |
| 4.3 | Process from GLI to Dafny Syntax | 40 |
| 4.3.1 | Writing C Code from GLI | 40 |
| 4.3.2 | Translation of C to Dafny Syntax | 44 |
| 4.4 | Getting Dafny Annotations | 50 |
| 4.4.1 | Standardization | 50 |
| 4.4.2 | Dictionnary Solution | 53 |
| 5 | Results | 57 |
| 5.1 | Complete Process | 58 |
| 5.1.1 | Research of Maximum | 58 |
| 5.2 | Compilation of Results | 64 |
| 5.3 | Limitations | 72 |
| 5.3.1 | Single or Two Nested Loop Program | 72 |
| 5.3.2 | Strict Compliance with GLI | 72 |
| 5.3.3 | Extraction of Variable Names | 73 |
| 5.3.4 | Integration Steps | 73 |
| 5.4 | Code Files | 74 |
| 6 | Conclusion | 75 |
| 6.1 | Future Work | 76 |
| A | Basic C Functionalities | 77 |
| | Bibliography | 81 |

List of Figures

| | | |
|------|--|----|
| 2.1 | Code structure of <code>findMax</code> function | 6 |
| 2.2 | Complete code of <code>findMax</code> function | 8 |
| 2.3 | First graphical view | 9 |
| 2.4 | Loop invariants for finding a maximum in an unsorted array | 10 |
| 2.5 | Templates of Graphical Loop Invariants | 11 |
| 2.6 | Rules for building a GLI and maximum example | 13 |
| 3.1 | Phases of verification in VS Code | 17 |
| 3.2 | Syntax of quantifier | 18 |
| 3.3 | Syntax of methods and functions | 19 |
| 3.4 | Syntax of pre- and postconditions | 20 |
| 3.5 | Syntax of invariant | 20 |
| 3.6 | Syntax of variant | 21 |
| 3.7 | Syntax of predicate | 21 |
| 3.8 | Syntax of lemma | 22 |
| 3.9 | Full code of <code>FindMax</code> method | 23 |
| 3.10 | GLIDE user interface | 24 |
| 3.11 | Blank GLI | 25 |
| 4.1 | Big picture of the implemented process | 27 |
| 4.2 | Drawings of a square and an isosceles triangle of height = 4 | 32 |
| 4.3 | Terminal states | 34 |
| 4.4 | Lower bound placement | 35 |
| 4.5 | Representation of a GLI as a picture and a JSON file | 38 |
| 4.6 | General pattern of an iterative loop | 41 |
| 4.7 | GLI manipulations to deduce C code | 42 |

| | | |
|------|---|----|
| 4.8 | Complete piece of code with zone labels | 43 |
| 4.9 | Example of an AST of a piece of code in C | 45 |
| 4.10 | AST generated by pycparser | 46 |
| 4.11 | Translated piece of code using DafnyGenerator | 50 |
| 5.1 | Potential GLI designed by a student | 59 |
| 5.2 | Completed Dafny program | 63 |
| 5.3 | Prompt to extract variables from sentences | 73 |

List of Tables

| | | |
|-----|--|----|
| 3.1 | Basic operations in Dafny | 18 |
| 4.1 | Values assigned to GLI types | 37 |
| 5.1 | Compilation of results for Problems 1-4 and 6-10 | 72 |

Chapter 1

Introduction

1.1 Context

The following research work was conducted within a specific context. That means a specific work environment, a particular field of application, and some basic constraints or hypotheses. More precisely, this research and its results will be used in introductory computer science courses, specifically the first course of the bachelor's program in computer science at the University of Liège, Introduction to Computer Programming (INFO0946-1 [9]). In this course, several learning unit contents are approached such that: “[...] • basic syntax and semantics of the C language; • simple algorithms (linear run of an array, cumulative mathematical operations, binary search, introduction to sorting problems) [...]” [9]. These two points give a good overview of the context of this work. We will use the basic syntax of the C language and simple algorithms like mathematical operations and sorting problems. Our research builds on that conducted by Pr. B. Donnet and PhD student G. Brieven, who use the Graphical Loop Invariant (GLI) theory in their courses and practical work.

1.2 Motivation and Objective

The initial motivation for developing an alternative and more educational theory for using loop invariants in program verification came from the observation that first-year computer science students have uneven foundations in logic and mathematics. In particular, at the University of Liège, no assessment is carried out before student enrollment. As a result, when it comes time to tackle loops and their invariants, many students cannot grasp them properly due to their gaps in logic and mathematics. In response to this problem, a visual approach to invariants has emerged. The aim was to abstract from mathematics and formal logic to enable as many people as possible to understand and use loop invariants when creating computer programs.

This work continues along the same lines by attempting to link the graphical and formal methods. Our ultimate goal is to implement a function in a GLI design tool that

allows Graphical Loop Invariants (GLI) to be used to verify C programs. Therefore, this work aims to establish the feasibility of a system that can automatically verify a program written in C using the associated GLI. We want to show that the development and implementation of such a system are viable and feasible projects.

1.3 Overview

This document details the various components of the research work carried out. We will begin by establishing the theoretical foundations on which our research is based (see Chapter 2). In the following chapter (see Chapter 3), we will present the tools used throughout the work that enabled us to obtain our results. We will continue with Chapter 4, addressing each of the different parts that comprise the methodology we implemented before summarizing all the results we obtained in Chapter 5. Finally, one will outline the conclusions of the work in the last chapter (see Chapter 6).

Chapter 2

Theoretical Background

In Computer Science, loops are particularly poorly mastered by new CS1 students (i.e., students in their first year of study). Yu. A. Cherenkova et al. found indeed that “[...] conditionals and loops prove particularly problematic [...]” [7]. However, there is a theoretical tool that allows us to build correct programs, and in particular, correct loops. Unfortunately, this formal companion, the loop invariant, which will be defined in the next section, struggles to find its place in students’ toolboxes. Although extremely useful for building a computer program that will be executed without error, a lack of mathematical and logical knowledge often prevents students from correctly understanding how the formal loop invariant works. That is unfortunate since the question of the correctness of computer programs is one of the most crucial issues programmers face. A new vision of this tool has emerged to tackle this issue and make the theory of the so-called “formal” invariant more accessible, as expressed by mathematical and logical formulas. In this vision, we promote the invariant graphically with the help of drawings.

In this work, and for the formal side, we will focus on the method based on Floyd-Hoare’s logic [10, 13]. As for the graphical aspect of invariants, the work carried out by Prof. B. Donnet et al.[5, 18] at the Montefiore Institute, University of Liège, will be our starting point.

Let us look at the theoretical foundations on which loop invariants are based. For the remainder of this work and the sake of brevity, the expression Formal Loop Invariant will be abbreviated as FLI, and that of Graphical Loop Invariants, which will be introduced in Subsection 2.1.2, as GLI.

2.1 Loops and Invariants

2.1.1 Formal Loop Invariant

Inductive Definition and Properties

As the introduction of the current chapter mentions, invariants can be used to check that a program delivers correct results. This correction is called partial because the validity of the results is subject to the program's termination condition. Indeed, programs, particularly those containing loops, may never reach the terminal state; in other words, they never end. As the program will never output results, we cannot prove these results to be wrong.

Actually, the definition of an FLI is: “[...] a property of a program loop that is true before (and after) each iteration.” “It is a logical assertion, [...] expressed by formal predicate logic and used to prove properties of loops and by extension algorithms that employ loops (usually correctness properties).” [26]. An FLI is thus a boolean expression that must be verified (i.e., true) at each iteration (i.e., each time the guard or loop condition is evaluated) of the loop.

Referring to Floyd–Hoare theory and logic, the FLI, defined in this way, allows us to prove specific properties of a program via the Hoare triplet proof of the form

$$\{P\} S \{Q\},$$

where S is executable code and P and Q are predicates (i.e., boolean functions) concerning variables, input, and output data. We call them precondition P and postcondition Q [3].

The triplet $\{P\} S \{Q\}$ is shown to be valid if and only if, when the predicate P is true before the execution of the statement S , then after the execution of S , Q is true. When S contains a loop, our FLI will give us the mandatory tool to assert the validity of Hoare's triplet. One way of visualizing the usefulness of an invariant in demonstrating a triplet is to subdivide S into four parts: *init*, *cond*, *body*, and *end*.

Require: P

Ensure: Q

1: *init*

2: **while** *cond* **do**

3: *body*

▷ S is the lines 1–5

4: **end while**

5: *end*

The invariant, which we will call I , does not appear directly, but is created to reason on the loop's validity. I will be correct if it meets these two properties:

- i) After every execution of *init*, if the required properties P were true, we will reach a

state where I holds [11].

- ii) Whenever I and $cond$ hold before the execution of $body$, then I holds also in the state after the execution of $body$ [11].

With these properties, we get the definition of an inductive FLI. If the execution of $init$ leads to a state where the invariant I is satisfied, and if any number of executions of $body$ maintain the invariant (as long as $cond$ remains true), then I will be satisfied in the state resulting from the execution of $init$ followed by the while loop.

Formally, proving the triplet $\{P\} S \{Q\}$ is now divided into three steps. With our invariant I , first, we need to ensure that

$$\{P\} \text{ init } \{I\}$$

is valid. Then, whatever the number of iterations of $body$,

$$\{I\} \text{ while } cond \text{ do } body \{I \wedge \neg cond\}$$

must remain valid. Finally, after the last iteration of $body$,

$$\{I \wedge \neg cond\} \text{ end } \{Q\}$$

should also be valid. The first two steps formally express the properties of an FLI:

- i) $\{P\} \text{ init } \{I\}$ is the initiation property and states that initialization ensures the invariant [11, 26].
- ii) $\{I\} \text{ while } cond \text{ do } body \{I \wedge \neg cond\}$ is the inductiveness (or conduction) property and states that the loop preserves the invariant [11, 26].

However, we can only conclude the loop's partial correctness without proving termination. To show its total correctness, we must prove that it will eventually end in a terminal state. We will introduce a loop variant to show that. This loop variant is a non-negative and monotonic decreasing quantity that plays a crucial role in proving termination. It is based on the loop's iterations and specific properties of the invariant, so it decreases with each iteration of the loop. Suppose the variant takes values in a well-founded ordered set (e.g., the positive integers). In that case, non-negativity combined with strict monotonic decrease ensures a finite number of iterations and, therefore, loop termination.

One last important thing to bear in mind is that the loop invariant can be either used to demonstrate *a posteriori* (i.e., after the code has been written) that the algorithm works correctly or can be used *a priori* to write code that is correct by construction. In the latter case, the invariant will help build the correct code on the first attempt. We will be discussing this use (i.e., *a priori* approach, inspired by Dijkstra [8]) throughout this research project because it avoids the need for trial and error to eliminate all errors from a program and presents a more pedagogical aspect of learning programming.

Practical example

Let us apply the above theory to an algorithm, here to the C function `findMax` (Figure 2.1), which returns the index of the maximal value inside an unsorted array.

```

1  int findMax(int a[], int length)
2  {
3      init;
4      while (cond)
5      {
6          body;
7      }
8      end;
9  }
```

Figure 2.1: Code structure of `findMax` function

The precondition $\{P\}$ is that the array contains at least one element, and that `length` corresponds to the size of `a`. The array must contain at least one element; otherwise, the maximal value is undefined. We therefore have

$$\{ |a| > 0 \wedge |a| = \text{length} \}.$$

As for the postcondition $\{Q\}$, the value of `index` must lie in the interval $[0, \text{length})$, the element at `a[index]` must be the largest of all the elements of `a`, and the array `a` must be preserved. Otherwise, one could rewrite the array with a single value, which will thus become the maximum of the array. That gives us, for $\{Q\}$,

$$\{ 0 \leq \text{index} < |a| \wedge \forall k. (0 \leq k < \text{length} \implies a[\text{index}] \geq a[k]) \\ \wedge \text{unchanged}(a) \}.$$

Then, with some intuition, we decide to define the FLI I as follows:

$$\{ 0 \leq i \leq \text{length} \wedge 0 \leq \text{index} < \text{length} \\ \wedge \forall k. (0 \leq k < i \implies a[\text{index}] \geq a[k]) \\ \wedge \text{unchanged}(a) \}.$$

It comprises four clauses which state that, at each iteration, `a[index]` must be the maximal value (i.e., third clause). The first two clauses ensure i and `index` are within the valid index range. The fourth clause prevents modification of the array.

1. Initialization.

The initiation property is satisfied if, after execution of `init`, one sets i and `index` to

0. Indeed, the Hoare triple

$$\begin{aligned}
 & \{ |a| > 0 \wedge |a| = \text{length} \} \\
 & \quad \text{init} \\
 & \{ 0 \leq i \leq \text{length} \wedge 0 \leq \text{index} < \text{length} \\
 & \quad \wedge \forall k. (0 \leq k < i \implies a[\text{index}] \geq a[k]) \\
 & \quad \wedge \text{unchanged}(a) \}
 \end{aligned}$$

will be valid. The first two clauses of the invariant follow immediately from the assignments. Since there is no $k < i = 0$, the third clause holds vacuously. The fourth clause is preserved as long as we do not modify any value in the array.

2. *Induction.*

At each iteration of the loop, if we find an element larger than the one at `index`, we update `index` to the index of this new maximum (which is i inside the loop). This operation preserves the third clause of our invariant,

$$\forall k. (0 \leq k < i \implies a[\text{index}] \geq a[k]),$$

and can be managed by an `if` statement:

$$\text{if } a[i] > a[\text{index}] \text{ then } \text{index} = i.$$

The first two clauses must be preserved by the loop guard (i.e., *cond*) and by the rest of the loop body. Thus, we prevent i from exceeding `length` by choosing the loop guard

$$\text{while } (i < \text{length}) \text{ do } \{ \dots \}.$$

We also ensure each iteration examines the next element by incrementing i . Hence, the Hoare triple

$$\begin{aligned}
 & \{ 0 \leq i \leq \text{length} \wedge 0 \leq \text{index} < \text{length} \\
 & \quad \wedge \forall k. (0 \leq k < i \implies a[\text{index}] \geq a[k]) \\
 & \quad \wedge \text{unchanged}(a) \} \\
 & \quad \text{while } (i < \text{length}) \text{ do } \text{body} \\
 & \{ 0 \leq i \leq \text{length} \wedge 0 \leq \text{index} < \text{length} \\
 & \quad \wedge \forall k. (0 \leq k < i \implies a[\text{index}] \geq a[k]) \\
 & \quad \wedge \text{unchanged}(a) \\
 & \quad \wedge \neg(i < \text{length}) \}
 \end{aligned}$$

remains valid throughout the loop iterations.

3. *Postcondition.*

When exiting the loop, since $\neg(i < \text{length})$, we have $i \geq \text{length}$. Together with the first clause of the invariant ($0 \leq i \leq \text{length}$), it follows that $i = \text{length}$. Consequently, the final Hoare triple

$$\begin{aligned}
 & \{ 0 \leq i \leq \text{length} \wedge 0 \leq \text{index} < i \\
 & \quad \wedge \forall k. (0 \leq k < i \implies a[\text{index}] \geq a[k]) \\
 & \quad \wedge \text{unchanged}(a) \\
 & \quad \wedge \neg(i < \text{length}) \} \\
 & \quad \text{end} \\
 & \{ 0 \leq \text{index} < \text{length} \wedge \forall k. (0 \leq k < \text{length} \implies a[\text{index}] \geq a[k]) \\
 & \quad \wedge \text{unchanged}(a) \}
 \end{aligned}$$

is valid, where end corresponds to returning index.

4. Termination.

We have shown that if the loop terminates, the result is correct (partial correctness). It remains to be proven that the loop indeed terminates. Introduce the loop variant

$$\text{length} - i.$$

This variant is non-negative and strictly decreases on each iteration (since i is incremented until it reaches length). Therefore, it must eventually reach zero after a finite number of iterations, ensuring that the loop (and hence the function) terminates.

All those steps illustrate a well-constructed FLI's usefulness in building a correct algorithm. In our case, the FLI comprises three clauses that led to the findMax function's algorithm shown in Figure 2.2.

```

1  int findMax(int a[], int length)
2  {
3      int i = 0;
4      int index = i;
5      while (i < length)
6      {
7          if (a[index] < a[i])
8          {
9              index = i;
10         }
11         i += 1;
12     }
13     return index;
14 }
```

Figure 2.2: Complete code of findMax function

To start thinking graphically from an FLI, the following picture (Figure 2.3), proposed by Carlo A. Furia et al. (*Fig 1. The loop as a computation by approximation* [11]), is a pretty good representation of the steps described above. The *Previous state* is our precondition, the *Exit condition* is met when $\neg cond$ is true, and the termination is proved by the finite amount of *body*

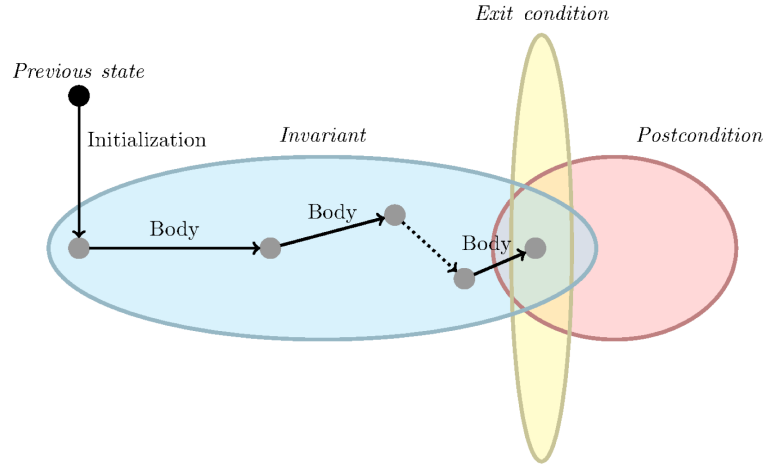


Figure 2.3: First graphical view

2.1.2 Graphical Loop Invariant

Overview

This section presents the theory of the Graphical Loop Invariant (GLI). However, before we do, let us take a quick look at the reasons that have led some people to view invariant theory in a completely different light. As mentioned above, mastering FLI requires a solid background in logic and mathematics. Indeed, students lacking the necessary mathematical and logical background would not easily understand the previous section. Most often, one cannot expect students attending a CS1 course to have this prerequisite knowledge already, and that is an important issue for them to reason rigorously on their first algorithm involving loops. They then turn to testing to correct their first programs through trial and error. This is how the idea of representing invariants emerged, not as Boolean expressions, but rather as evolutionary drawings.

In the same way as for FLIs, in this work, we consider GLIs according to the *a priori* approach, just as in previous work by Liénardy et al.[18] and Brieven et al.[5]. In the latter, we use the invariant as a tool for constructing the program (i.e., a constructive approach) and not as a means of verifying (i.e., asserting that a program works correctly) an already written program.

Considering our intent of abandoning complex logical and mathematical concepts and our constructive approach of invariants, let us discuss how the graphical methodology differs from the formal one presented by Dijkstra. To best reflect the original concept (i.e.,

formal loop invariants), the image (i.e., GLI) must contain the same information as the FLI. That includes constants, variables, and data structures, as well as their associated constraints (domain, bounds, etc.) and all interrelationships among these elements, which must be preserved.

To be consistent and better understand the links with the so-called “formal” invariant (i.e., FLI), we will reuse the same example of finding a maximum in an array a.

Input: an unsorted array a and its length

Output: the index of the maximal value

Figure 2.4a shows how the example (i.e., finding the maximum value in an array) should be solved through the corresponding GLI. We start by drawing a rectangular shape to represent the array of the problem. Then, we give it a name: a, to be able to refer to it. To stay as general as possible, we state the array is composed of N elements and label them with integers representing the indices. A vertical red line (the *Dividing Line*) is added to describe the program’s state after several iterations. It splits the array into two distinct regions. The blue area on the left includes the elements already compared with the current maximum value’s index stored in the variable index. This variable acts as an *Accumulator*, tracking the progress. The green area on the right contains elements that have yet to be compared against the current maximum. The element immediately to the right of the *Dividing Line* is labeled i, which serves as the *Iterator* traversing the array from index 0 to N. The black-bold lines define the range. They act as boundaries for the *Dividing Line*, in such a way that if we move the *Dividing Line* from the left to the right, we immediately notice that the *Iterator* i goes from 0 to N. Of course, the variables i and index must be used in the code.

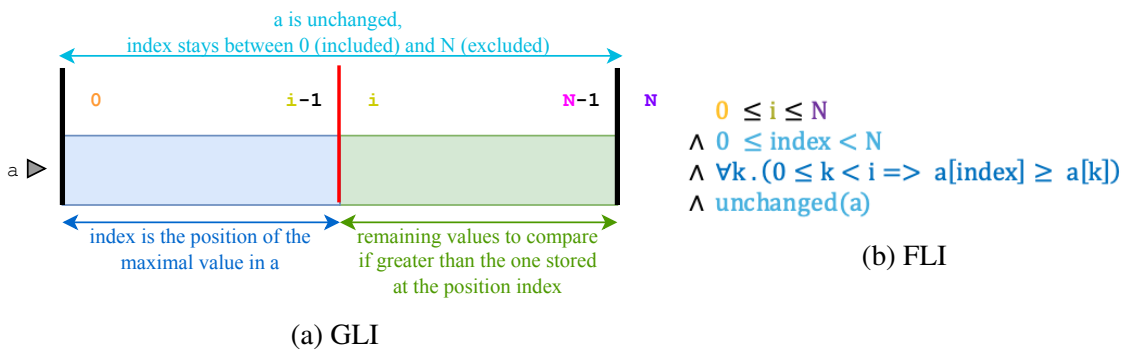


Figure 2.4: Loop invariants for finding a maximum in an unsorted array

Following the example presented above, it is clear that many of the elements that held back students lacking the necessary logical background are no longer apparent in the GLI. Indeed, when we compare the GLI in Figure 2.4a with the FLI shown in Figure 2.4b, we can see that logical and mathematical notions such as \forall , \wedge , or \in have been replaced by sentences and graphical representations. We also introduced color coding, which is described in part concerning the construction of a GLI between the two types of invariants to allow better visualization of equivalent elements.

Patterns

Arrays are not the only structures covered by the CS1 course that GLIs can represent. There are others, and this section introduces them. We will look at the characteristics of each pattern. To maintain consistency and ease of understanding, we will keep the color scheme introduced in the previous example and detail it in the next section. This color scheme will be maintained throughout the rest of the document.

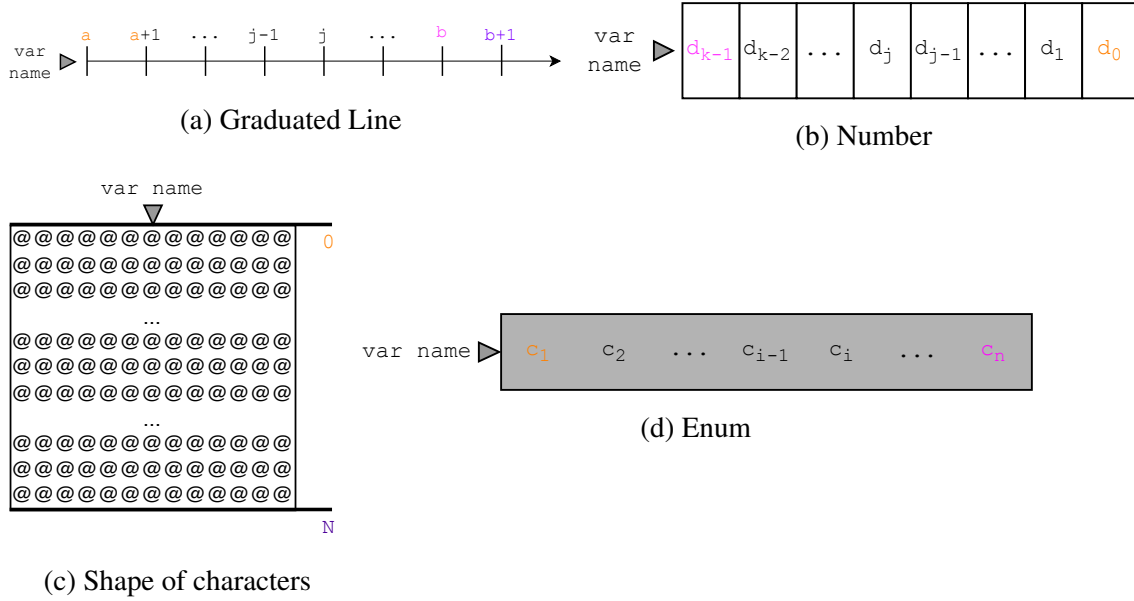


Figure 2.5: Templates of Graphical Loop Invariants

- *Array*

The array pattern illustrates the example described in the previous section. This pattern is a standard visualization of an array containing N elements. In this case, we use a solid rectangular shape to represent the continuity of the data. Cells of interest in the array, such as the first and last, are highlighted by their indexes, 0 and $N-1$ (see Figure 2.4a). Note that the size of the array, represented by the index N , to the right of the upper boundary (i.e., above a blank), is present to highlight that the cell $a[N]$ is not accessible. The array's name is placed before it to specify how to access the structure.

- *Graduated Line*

The graduated line (Figure 2.5a) is the most basic pattern in the collection. It represents problems involving reasoning with integers or naturals (e.g., compute the sum of the first N naturals). This pattern consists of an oriented straight line intersected by equidistant vertical segments. The orientation of the straight line, represented by the arrow at its end, marks the ascending order in which the values appear. The vertical segments, placed at regular intervals, indicate the constant offset between values. The `var name` label specifies which set of numbers appears on the line (e.g., \mathbb{N} or \mathbb{Z}).

- *Number*

When, in the course of algorithms, we come to manipulate the digits of a number (e.g., calculating the number of zeros in the decimal representation of a number), then the number pattern comes into play. One can use it for all representations, whether decimal, hexadecimal, binary, or other. In Figure 2.5b, we represent a number with k digits by a sequence of d_j digits. The most significant digit is placed on the left (d_{k-1}) and the least significant on the right (d_0). The direction in which the digits are evaluated may differ occasionally. In Figure 2.5b, we have chosen d_0 as the minimum bound and d_{k-1} as the maximum bound.

- *Shape of characters*

This pattern seems less valuable than the others when it comes to programming. It draws simple geometric shapes using ASCII characters in a terminal. Figure 2.5c shows a square, but other shapes, such as triangles, trapezoids, or even hourglasses, are possible. The name of the figure is given in `var name`, while the vertical dimension is delimited by `N`. This pattern introduces a crucial programming concept: two-dimensional arrays. The drawn area can be considered as a set of lines that define a matrix together. However, that is not a proper matrix because we are working with characters; we will talk instead about a single string (one-dimensional array) of characters containing the `'\n'` character, allowing you to return to the line.

- *Enum*

This parameter is quite similar to the array parameter and the number parameter. However, unlike the array, it is not a modifiable object but simply a sequence of elements (e.g., numbers, characters, etc.). What is more, unlike a number, there is no link between the various elements listed. For example, if they are numbers, then the distance between them may not be constant. The structure is identified by `var name`. This pattern works well in conjunction with *Shape of characters*. It allows us to iterate over the characters on the same line.

Construction of a Graphical Loop Invariant

Finding the right loop invariant, whether formal or graphical, is the most difficult and important step. There are several ways of approaching the problem, such as starting from the precondition and trying to reach the postcondition or reasoning by induction. With the graphical invariant technique, the recommended method is to graphically apply the constant relaxation technique [11]. In other words: “[...] replacing an expression (that does not change during program execution - e.g., some n) from the postcondition by a variable i , and use $i = n$ as part or all of the Stop Condition.” [5]. Six rules (see Figure 2.6) have been established to provide a framework for this practice [5, 9]. They fall into two categories: syntactic (i.e., how to draw the GLI —*Rules 1-4*) and semantic (i.e., how to give meaning to the GLI —*Rules 1, 5, 6, 5+6*).

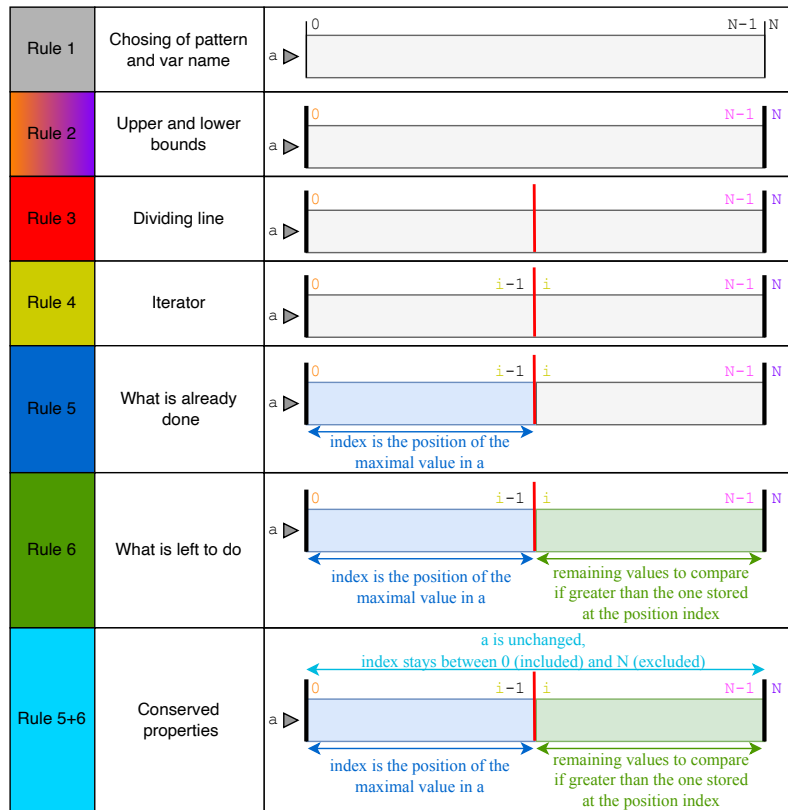


Figure 2.6: Rules for building a GLI and maximum example

Let us review the different rules, using our example of finding the maximum of an array. The first step is to establish the result to be achieved.

Output: $a[\text{index}]$ is the maximal value of the array

Based on the latter, we can apply *Rule 1*, *2* directly. Depending on the data structure used (in this case, an array), we choose the most appropriate pattern (*Rule 1*). It may happen that, for a given problem, we are playing with several structures and, therefore, several GLIs and potentially different patterns. That is where *var name* comes into its own. It allows us to find our way around when our problem requires several invariants, and to avoid mixing everything up when passing from GLI to code. Next, we determine the upper and lower bounds and the size of our structure (*Rule 2*). This step is essential, especially with arrays, as it highlights the limits of our problem. In this way, we can prevent future errors, such as a buffer overflow.

The next step, *Rule 3*, is the heart of the graphical invariant method. It consists of adding the *Dividing Line*. The *Dividing Line* plays two essential roles. First, it divides the structure into two zones, allowing the visualization of what has been done (*Rule 5*) and what remains to be done (*Rule 6*). Note that we are not limited to only one *Dividing Line*. One can add as many as needed to create as many zones as wanted. Second, it introduces the possibility of dynamically and visually modifying the GLI. Thanks to that, we can see how the program evolves over the iterations of the loop. Once the *Dividing Line* is in place, *Rule 4* is applied, and it is time to choose which side to place the *Iterator* on. Positioning

the iteration variable (i.e., *Iterator*) is possible on both sides (i.e., left or right). However, in general, we prefer to place it on the right. In this way, the variable indicates the next element to be processed. Furthermore, the initial state (i.e., *Dividing Line* on lower bound) and terminal state (i.e., *Dividing Line* on upper bound) are better represented, as you can see directly which value the *Iterator* will take.

Now that the zones have appeared, it is time to make sense of them. The *Rule 5* describes what previous iterations have already achieved. In fact, the blue zone label is strongly inspired by the final result but adapted to an intermediate state. This rule asks: “To achieve such a result, what should I have achieved by x% of the process?”. In our example, to get the maximum of the array at the end, we need to know, at each step, the maximum of the part of the array we have already analyzed. This step encourages students to think about how the loop works and which operations to include. In some cases, the *Rule 5* creates *Accumulator* variables to store the results of past iterations (e.g., to store a partial sum).

The *Rule 6*, on the other hand, describes what remains to be done. While it may seem less valuable than the others, as it has no equivalent in FLI, it does have certain advantages. First, as in the previous step, it stimulates thinking about the actions to be carried out in the loop. Then, it enables the visualization of the algorithm’s progress. The green zone is, in fact, total at the outset and gradually shrinks until it finally disappears completely, revealing that the loop has completed its mission. At the same time, this gradual reduction makes the size of the *To do* zone a good indicator of the Loop Variant (e.g., $N-i$ in our example).

Finally, *Rule 5+6* is optional (i.e., not always used) and describes a global property that must be maintained throughout iterations. In our case, the fact that `index` must be part of the range of index values in the `a` array.

A final step, sometimes referred to as *Rule 7*, consists of checking the consistency between the GLI and the deduced code (e.g., do not forget to mention all the variables that appeared during the previous rules).

Chapter 3

Programming Languages and Software

This chapter will examine this project’s various software and programming languages. As explained in the introduction (see Section 1.1), students begin by learning to program in the C language. They are taught the graphical loop invariant method to help them build their first `while` or `for` loops in this programming language. So, in this research work, we will concentrate on applying the C language. As C is one of the best-known languages in computer science, we will not go back over its specific features. Just bear in mind that we are in the context of an introductory course, so the algorithms presented will remain relatively simple and will not go into all the subtleties of C programming. A brief summary of the C functionalities will nevertheless be given in Appendix A.

In the course of this work, we also came across the use of Python. For the same reasons as for C, we will not go back over the basic principles of this language. However, since we have introduced some of Python’s particular features and libraries, a presentation of them will be given in Section 4.3 and Section 4.4.

Finally, this chapter looks at two main programming languages and software. Section 3.1 introduces the Dafny [24] verification language, combined with the Visual Studio Code IDE. It relies on the works of Herbert et al.[12], Koenig [14], and Leino [17, 14], and on the lecture of Sonnex and Drossopoulou [27]. Section 3.2 introduces the CAFÉ 2.0 software [4] and its GLI online design tool, the *Graphical Loop Invariant Drawing Editor* (GLIDE) [5].

3.1 Dafny

3.1.1 Overview

Dafny is a programming language designed to make it easy to write correct computer programs. This so-called hybrid language combines functional programming features, such as recursion, with those of object-oriented programming. Dafny is designed to verify algorithms formally (i.e., using pre- and postconditions, loop invariants), automatically, and completely. In other words, Dafny ensures the total correctness (see Section 2) of a program (i.e., its termination and the satisfaction of its specifications). A program written (and verified) in Dafny is guaranteed to be free of runtime errors and valid with respect to the programmer’s expectations (i.e., following the formal specifications provided—e.g., invariants, preconditions, etc.).

To achieve that, the programmer provides the Dafny language with high-level indications (i.e., *Annotations* in the code). For instance, such an indication can be

ensures $0 < \text{rtn} < N$,

that ensures that the value `rtn` is strictly within the interval $]0, N[$. The theorem prover behind the Dafny language is Z3 [21]. The way it works is the following. First, the Dafny program is translated into an intermediate verification language, Boogie 2 [1]. The translation is such that if the Boogie program is verified, one can conclude that the Dafny one is correct. Then, some first-order logic conditions are generated from the Boogie tool and given to Z3, an SMT (i.e., satisfiability modulo theories) solver. When the power of Z3 is insufficient, the programmer has to contribute. That can be done by subdividing the verification of a complete program into several verifications of sub-parts of the program or by introducing lemmas (i.e., a minor property demonstrated for use in a larger proof). As Koenig and Leino put it so well: “Dafny lifts the burden of writing bug-free *code* into that of writing bug-free *Annotations*.” [14]

3.1.2 Basics of Dafny Utilization

Environment

There are several ways of writing and executing (i.e., checking) code in Dafny. We have opted to use the Visual Studio Code IDE. Microsoft has developed both, so that is undoubtedly the smoothest way to run Dafny programs. The current installation requires several downloads. Everything is clearly explained on the dedicated Dafny documentation page ¹. One will find all the instructions for each operating system and purpose. Once the installation is done, one will be in a comfortable work environment to start using Dafny.

¹url: <https://dafny.org/latest/Installation>

Verification of Code

Figure 3.1 represents the different possible states of a code in Dafny. In the event of a problem, one can immediately distinguish what has been verified from the errors (e.g., index out of range in Figure 3.1c and missing semicolon in Figure 3.1d) encountered by the *Verifier* (i.e., red squiggly underlining like a spell checker). This check is automatic and does not require going through the command terminal. However, one can manually check a file by entering the command:

```
$ dafny verify filename.dfy
```

To execute a program, we use:

```
$ dafny run filename.dfy
```

The difference between both commands is that `verify` checks the correctness of the code, while `run` executes the instructions of the program (e.g., computing a sum, displaying some results, etc.).

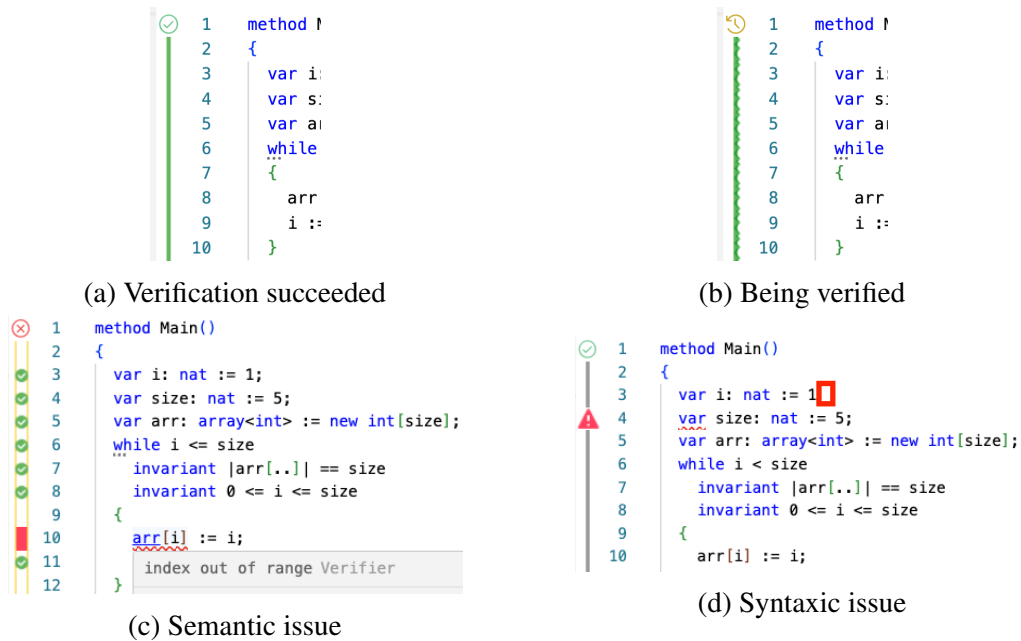


Figure 3.1: Phases of verification in VS Code

Operations

Dafny's arithmetic, logic, and comparison operations are similar to those of the C language, as seen in Table 3.1. However, there are some differences. Assignment is made via the symbol `:=` rather than `=`, and we note the introduction of implication (`==>`) and equivalence (`<==>`), which will be used in particular in invariants.

| Category | Operator | Description |
|------------|----------|---|
| Arithmetic | + | Addition |
| | - | Subtraction |
| | * | Multiplication |
| | / | Division (integer division if both operands are integers) |
| | % | Modulo |
| Comparison | == | Equal to |
| | != | Not equal to |
| | < | Less than |
| | <= | Less than or equal to |
| | > | Greater than |
| | >= | Greater than or equal to |
| Logical | && | Logical AND |
| | | Logical OR |
| | ==> | Logical implication |
| | <==> | Logical equivalence (if and only if) |
| | ! | Logical NOT |
| Unary | - | Unary negation (for numbers) |
| Assignment | := | Assignment |

Table 3.1: Basic operations in Dafny

Quantifiers

Dafny also adds the feature of using universal, \forall , and existential, \exists , quantifiers. The latter is handy for expressing properties over a set of values. The syntax of quantifiers is shown in Figure 3.2. It can be seen in Figure 3.2a that we introduce a temporary bound

```
forall k :: 0 <= k < a.Length ==> a[index] >= a[k]
```

(a) Universal quantifier \forall

```
exists i, j :: 0 <= i < j < a.Length && a[i] > a[j]
```

(b) Existential quantifier \exists

Figure 3.2: Syntax of quantifier

variable k which is separate from the quantified property with a pair of colon symbols ($::$). Here, the property states that for each k in the index interval of the array a , the value of $a[\text{index}]$ is greater than or equal to the value of $a[k]$. That means $a[\text{index}]$ is one of the maximal values of the array. It is thus the Dafny equivalence to the following formal expression (see Section 2.1.1):

$$\forall k. (0 \leq k < |a| \implies a[\text{index}] \geq a[k])$$

Methods and Functions

Dafny methods are pieces of code that are imperative and executable when called. They can be seen as C functions, but the term ‘function’ in Dafny is reserved for another use. Methods and functions are declared as follows. Here, in Figure 3.3a, the method is

```
method FindMax(a: array<int>, length: nat) returns (index: nat)
{
  | //...
}
```

(a) Method’s declaration

```
function power(base: nat, exp: nat): nat
{
  if exp == 0 then 1
  else base * power(base, exp - 1)
}
```

(b) Function’s declaration

Figure 3.3: Syntax of methods and functions

called `FindMax`, takes two arguments (`a` and `length`) whose types are specified via the syntax `var name: type`. The return variable (called *out parameter*) and its type are also contained in the declaration. The body of the method contains a series of statements (e.g., `if` statement, `while` loop, etc.). Functions operate differently from methods. They are semantically similar to mathematical functions. That means they cannot write to memory and consist of a single expression (e.g., an `if then else` statement). In Figure 3.3b, the function `power` calculates the result of base^{exp} .

Pre- and Postconditions

With pre- and postconditions, we are really starting to see the advantages of the Dafny language. The power of Dafny indeed lies in its ability to add *Annotations* that specify properties to be respected. In the case of pre- and postconditions, these are input and output conditions. A precondition is designated by the keyword `requires` and a postcondition by `ensures`. An example of the `FindMax` method is shown in Figure 3.4. Thanks to these *Annotations*, we can describe Hoare’s triplet (i.e., $\{P\} S \{Q\}$) presented in Section 2.1.1 in Dafny code. If we come back to our example of the search for the maximum, the postcondition $\{Q\}$ is formally expressed by

$$\{ 0 \leq \text{index} < |a| \wedge \forall k. (0 \leq k < \text{length} \implies a[\text{index}] \geq a[k]) \\ \wedge \text{unchanged}(a) \},$$

whose the first two clauses correspond to the two *Annotations* ensures in Figure 3.4.

```

method FindMax(a: array<int>, length: nat) returns (index: nat)
  requires a.Length > 0
  requires a.Length == length
  ensures 0 <= index < a.Length
  ensures forall k :: 0 <= k < a.Length ==> a[index] >= a[k]
{
  //...
}

```

Figure 3.4: Syntax of pre- and postconditions

Loops

There are two types of loops in Dafny: for loops and while loops. As with the C language, for loops are just syntactic sugar and can be expressed as while loops. So, for the sake of generality, we will only be using while loops throughout this work.

Loop Invariants and Variants

Loop invariants allow us to tell the Verifier that we want a property to be preserved after each pass (i.e., each execution) through a loop. These are *Annotations*, like pre- and postconditions; their syntax is shown in Figure 3.5. When the *Annotation*

```

while (i < length)
  invariant 0 <= i <= length
  invariant 0 <= index < length
  invariant forall k :: 0 <= k < i ==> a[index] >= a[k]
{
  //...
}

```

Figure 3.5: Syntax of invariant

invariant is used, the property defined afterward must be checked on entering the loop and preserved (i.e., maintained) at each iteration. Dafny distinguishes between these two conditions, and an error message specific to each case is transmitted in the event of non-compliance (i.e., “could not be proved on entry” and “could not be proved to be maintained by the loop”²).

Dafny also takes care of proving loop termination (i.e., no infinite number of iterations). To do that, we use the *Annotation* decreases (see Figure 3.6a.) However, in the case of sufficiently simple loops, Dafny can assume the loop variant from the loop keeper on its own. In that case, a small note will appear under the keyword while specifying which variant Dafny has chosen (see Figure 3.6b).

²error messages from Dafny *Verifier*

| | |
|---|---|
| <pre>while (i < length) decreases length - i</pre> <p>(a) Manual variant</p> | <pre>while (i < length) decreases length - i Resolver</pre> <p>(b) Automatic variant</p> |
|---|---|

Figure 3.6: Syntax of variant

Predicates

Dafny predicates are functions that return a Boolean value. They are handy for improving the readability of code. They also enable programmers to build a library of general predicates and use them in different contexts. For example, checking that a value is the largest in an array can be expressed via a predicate, using the keyword `predicate` to declare it. In our maximum search example (see Figure 3.7), one can notice the `subLength` argument. We introduce it to make the predicate as general as possible. So, when `subLength = a.Length`, we check the entire length of the array `a`, and when `subLength = i` we check from index 0 to index `i-1`. The reads *Annotation* allows the

```
predicate Max(a: array<int>, subLength: nat, index: nat)
  requires 0 <= subLength <= a.Length
  requires 0 <= index < a.Length
  reads a
{
  forall k :: 0 <= k < subLength ==> a[index] >= a[k]
}
```

Figure 3.7: Syntax of predicate

predicate to access the memory area where the `a` array is stored. By default, the function’s reading frame (i.e., all the memory locations the function is allowed to read) is empty, and you need to mention explicitly what a function can access. The reason for this limitation is to avoid race conditions. A similar *Annotation*, `modifies`, can be applied to methods to authorize data modification in memory, particularly in arrays.

Lemmas

Sometimes, the code may be too complicated to be verified immediately. Indeed, to prove the validity of a program, Dafny starts again from the foundations of first-order logic, which may require too much time or computing power. However, the programmer can help Dafny by proving some specific properties. These demonstrations can be proved by induction, contradiction, etc. They are declared with the keyword `lemma` and take the kind of structure shown in Figure 3.8.

Others

There are a few other Dafny features we have used in this project. In particular, we used sequences (`seq<T>`) for their practicality in mathematical demonstrations. For

```
lemma L(...)
  requires P
  ensures Q
{
  if !Q {
    ...
    assert !P;
    assert false;
  }
}
```

(a) Proof by contradiction

```
lemma NatSuccPositive(n: nat)
  requires n >= 0
  ensures n + 1 > 0
{
  if !(n + 1 > 0) {
    assert n + 1 <= 0;
    assert n <= -1; // contradiction with n >= 0
    assert false;
  }
}
```

(b) Example of proof by contradiction

Figure 3.8: Syntax of lemma

example, subsequences can be considered via `seq[i..j]` (elements from index `i` included to `j` excluded). We also used multisets (`multiset<T>`), which do not care about the order but track how many times each element occurs. They are instrumental in sorting algorithms to check that one is only swapping elements in an array. Finally, we used `old(e)`, which returns the value of `e` at the beginning of the method. It allows us to compare the before and after state, which is, for example, useful in the case of proving a method that swaps elements in an array.

3.1.3 Example of Dafny Program

What has been presented in Section 3.1.2 is, of course, only an overview of the possibilities offered by the Dafny language. For an exhaustive list of what it can do, please refer to the Dafny Reference Manual [24]. It is available online ³ and updated regularly.

Nevertheless, with everything we have covered, we can start proving the algorithms seen in the Introduction to Computer Programming course. Figure 3.9 is a good example. It brings together many of the features discussed above. For more precision, the invariants, preconditions, and postconditions are described formally and graphically in Section 2.1.1 and Section 2.1.2.

3.2 GLIDE

CAFÉ 2.0 (standing for “Correction Automatique et Feedback des Étudiants”⁴) [4] is a learning tool developed to assist students in learning programming. This online tool aims to help students adopt an active and regular programming practice and to provide them with personalized feedback on their work. In this context, the *Graphical Loop Invariant Drawing Editor* (GLIDE) was developed and added to CAFÉ.

³url: <https://dafny.org/latest/DafnyRef/DafnyRef>

⁴which means “Automated Assessment and Feedback for Students”

```

1  predicate IsUnchanged(a: seq<int>, b: seq<int>)
2  {
3      a == b
4  }
5
6  predicate Unsorted(a: array<int>)
7      reads a
8  {
9      exists i, j :: 0 <= i < j < a.Length && a[i] > a[j]
10 }
11
12 predicate Max(a: array<int>, subLength: nat, index: nat)
13     requires 0 <= subLength <= a.Length
14     requires 0 <= index < a.Length
15     reads a
16 {
17     forall k :: 0 <= k < subLength ==> a[index] >= a[k]
18 }
19
20 method FindMax(a: array<int>, length: nat) returns (index: nat)
21     requires Unsorted(a)
22     requires a.Length > 0
23     requires a.Length == length
24     ensures 0 <= index < a.Length
25     ensures Max(a, a.Length, index)
26     ensures IsUnchanged(a[..], old(a[..]))
27 {
28     var i: int := 0;
29     index := i;
30
31     while (i < length)
32         decreases length - i
33         invariant 0 <= i <= length
34         invariant 0 <= index < length
35         invariant Max(a, i, index)
36         invariant IsUnchanged(a[..], old(a[..]))
37     {
38         if a[i] > a[index] {
39             index := i;
40         }
41         i := i + 1;
42     }
43 }

```

Figure 3.9: Full code of FindMax method

3.2.1 User Interface

GLIDE offers students a working environment in which they can develop their Graphical Loop Invariants. Figure 3.10 shows the user interface of the design tool. All the tools present on the sidebar (*Dividing Line*, lower and upper bounds, zones selector, eraser, etc.) enable the user to apply the rules seen in Section 2.1.1 and repeated in Figure 2.6. In this way, users can build their invariants from scratch and manipulate them as they see fit.

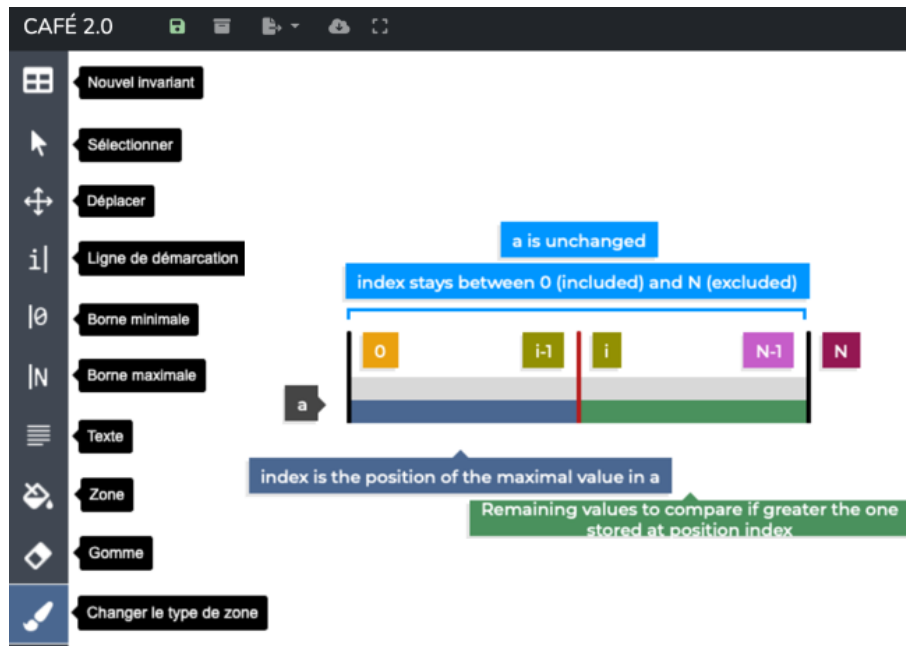


Figure 3.10: GLIDE user interface

3.2.2 Blank GLI

CAFÉ also offers a system of challenges in which students are asked to think about algorithmic problems (e.g., programming a function determining whether a number is prime). During these challenges, students receive a GLI as a canvas to fill in, also known as a Blank GLI (see Figure 3.11). There are two types of boxes to fill. Red boxes are for expressions (variables, constants, left blank, etc.) with freely selectable content. Conversely, green boxes are designed to contain predefined labels chosen from the list on the left (containing around fifty labels). This way of using Graphical Loop Invariants tends to standardize the final output and opens the door to automatic correction.

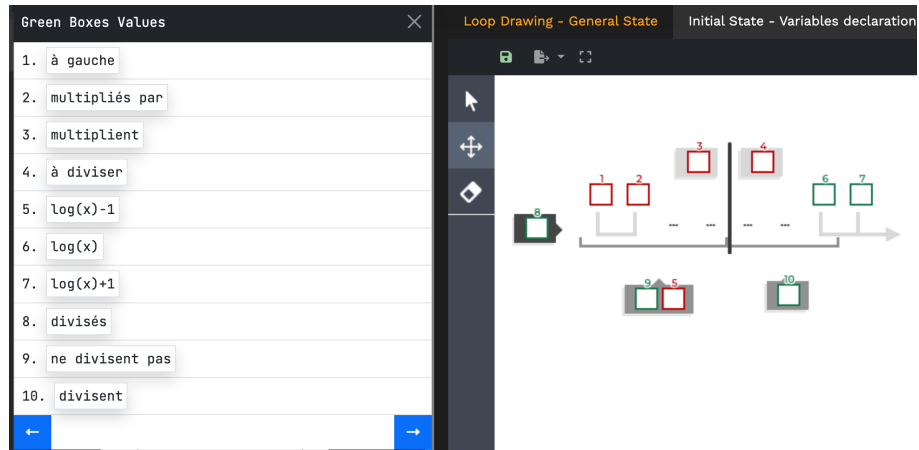


Figure 3.11: Blank GLI

Chapter 4

Methodology and Process

This chapter presents the methodology and thought process implemented during this research work. In order to achieve the objective of this work (i.e., to prove the feasibility of automatic verification of Graphical Loop Invariants), our methodology was the following. For each programming challenge, we proceeded as follows. We started building a suitable GLI for the studied problem. Thanks to this GLI, we built code to solve the programming challenge using the C language. Then, we transformed the C syntax into Dafny syntax. At the end, we added the Dafny *Annotations* (i.e., invariant, preconditions, postconditions, etc.). We finally came up with a way to automate the translation from C to Dafny and the addition of *Annotations*. In this chapter, we will retranslate and explain the key stages of this process.

First, as stated in Chapter 1, the content of the INFO0946-1 course [9] is limited to an introduction to computer programming. The programming covered concepts (i.e., arrays, sequences of numbers, etc.) are restricted to specific programs (i.e., challenges submitted to students) and GLI patterns (see Section 2.1.2) with certain influencing specifications. We are not trying to demonstrate the feasibility of every possible GLI and associated program. We limit ourselves to the programming challenges presented in Section 4.1. Section 4.2 then describes which information contained in a GLI can be systematically extracted and transformed into more formal predicates. As we want to achieve automatic verification of programs thanks to GLI, we must develop a systematic method of parsing GLI. Section 4.3 covers two essential points. The first is using a GLI to write code (in our case, in the C language). We indeed chose for *a priori* use of invariant to help build a totally correct program on the first attempt. The second step is to check that this code is correct by translating it into Dafny syntax. Finally, in Section 4.4, we will look at the most sensitive point of our methodology, the translation from loop invariant clauses described by sentences into formal clauses, implemented in first-order logic, by mathematical or computerized functions.

For the sake of clarity, a big picture of the process with related sections for each step can be observed in Figure 4.1 below. That picture can be used as a road map of the current and the next chapters.

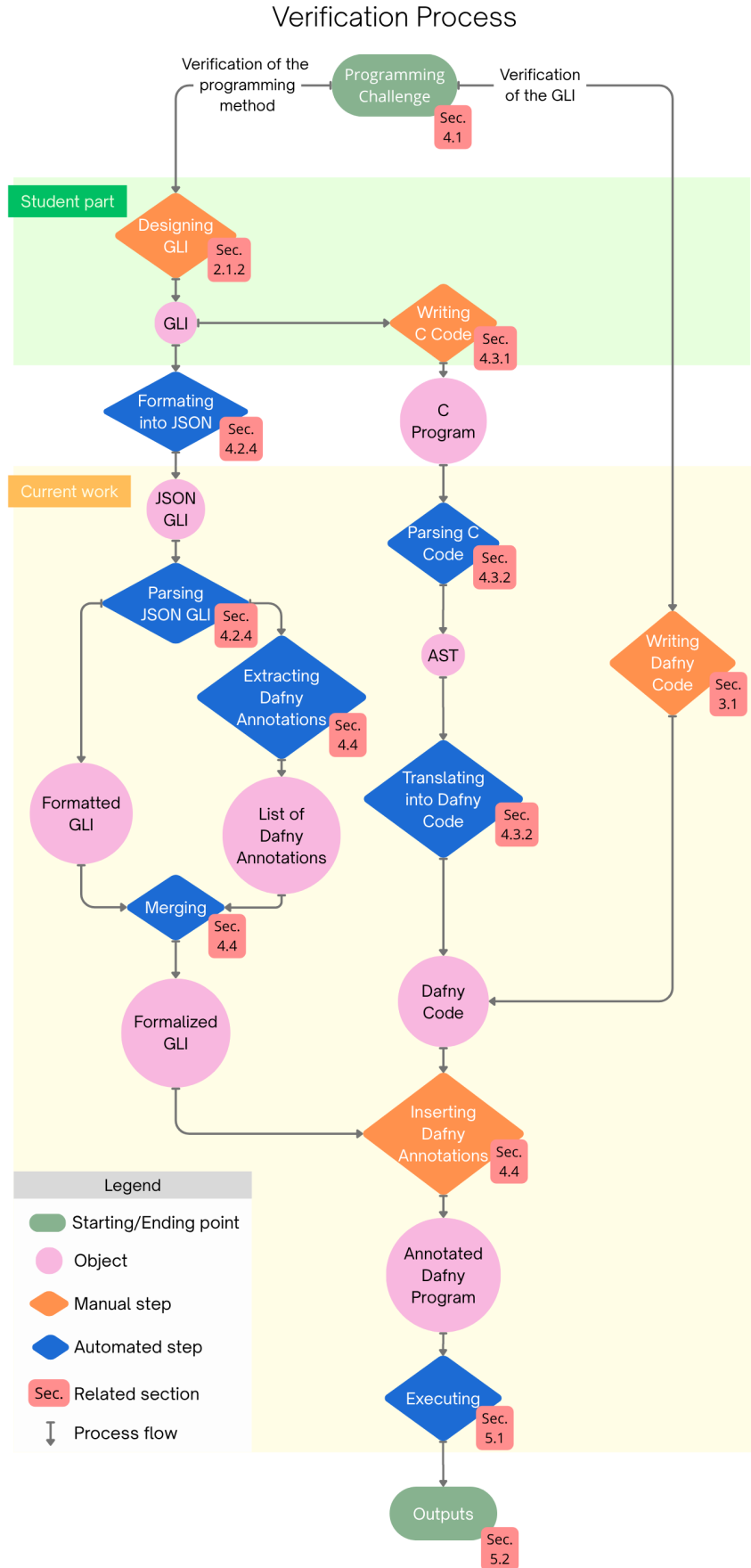


Figure 4.1: Big picture of the implemented process

4.1 Programming Patterns

This section will review the different types of programming problems covered in the CS1 course (i.e., Introduction to Computer Programming [9]), which has been analyzed as part of this work. The list of programming challenges that can be proposed to students as part of the course is very long. Consequently, we have not been able to cover this list exhaustively. There are, in fact, many possible variations of statements. However, we have tried to be as representative as possible by choosing a set of problems with specific features that can be found in other types of issues. For each analyzed problem, we highlighted the key points to solve to be able to verify it. We must also note that these programming patterns match the GLI patterns introduced in Section 2.1.2.

4.1.1 Assessing Properties for Sequences of Numbers

Manipulating sequences of numbers is one of the types of problems that students can be confronted with. The main aim is to demonstrate a property or to calculate a result. In the first case, we might ask, for example, to display the prime numbers among a sequence of numbers. In the second case, we might want to calculate the sum of n natural primes, or the product of integers between a and b , with $a < b$. Sequences of numbers also allow us to code programs with nested loops (i.e., `while` loop inside another `while` loop).

An example is the display of prime numbers. In reality, these are two sub-problems. The first is to run through a sequence of numbers (first `while` loop). The second is to check whether each number n encountered is prime, i.e., its only divisors are 1 and n (second `while` loop).

Selected Problems

The programming challenges for which the feasibility of an automatic GLI check has been analyzed are as follows:

1. Calculate the sum of the first natural numbers.
2. Calculate the product of integers in the interval $[a, b]$.
3. Check whether a number is prime.
4. Count the number of occurrences of a digit in a given sequence of numbers.

Specificities

The study of challenges 1 to 4 has revealed several recurring themes in this type of problem that require particular attention.

First, there is the concept of upper and lower bounds. That is present in every problem. We always have to traverse a set of elements. That may be, for example, the n

natural primes (Problem 1) or the divisors of a number (Problem 3). These different runs have a beginning, an end, and a lower and an upper limit.

Second, some problems must be reduced to several sub-problems (SP). In the case of Problem 4, on the one hand, we need to be able to traverse a given sequence and keep track of the overall number of occurrences of the digit d (SP1). On the other hand, for each element of the sequence, we need to count the local number (i.e., only in that element) of occurrences of the digit d (SP2). So, here two nested loops appear.

4.1.2 Operations on Array

Arrays are fundamental structures in programming. So, in an introductory course to computer programming, it is more than necessary to cover this important topic. Using and manipulating arrays allows access and play (i.e., read and write values) with memory. While that is a crucial asset, it is also a dangerous game. Indeed, a program manipulating arrays needs to draw our attention to read and write operations. The problems that students may encounter are varied. They may have to manipulate a single array to determine its maximum value or to sort it. Other challenges may involve several arrays, such as inserting one array inside another.

Selected Problems

The programming challenges for which the feasibility of automatic GLI verification has been analyzed are as follows.

5. Determine the index of the maximum value of an array.
6. Sort an array by implementing the Selection Sort algorithm.
7. Combine two integer arrays, inserting one of the two arrays at a given position.

Specificities

We identified several programming specificities after looking at challenges 5 to 7, as in Section 4.1.1.

As with the first four challenges, boundary issues are involved in the three studied problems (Problem 5, 6, and 7). If, with integer sequences, it was necessary to specify the maximum and minimum bounds, that is also the case for manipulating arrays. The only difference is that these bounds must absolutely lie between zero and the dimensions of the array(s) being traversed. Indeed, the array size plays a significant role in determining the limits. Without this precaution, there are two risks. The first is errors when reading values from memory. It is essential to prevent a program from reading values from memory locations that have not been allocated. The second is errors when writing values to memory. As with read operations, a program cannot write values to memory outside its allocated

areas. Fortunately, both operation types can be controlled by carefully selecting the upper and lower bounds.

The second point of attention, linked to problems involving arrays, concerns the integrity of the data contained in these arrays. We want to ensure that the data is only modified if the statement requires it (e.g., incrementing or decrementing the values of an array). Otherwise, ensuring that the program has not modified the data is essential. In the case of finding the maximum value of an array (Problem 5), we could imagine a program writing a single value to all the indexes of the array and returning any of the indexes. The index returned would then be that of the maximum value of the array, but for all that, we could not say that we had solved the problem correctly. Similarly, we do not want a sorting algorithm (Problem 6) to replace the values of an array with a sequence of increasing numbers.

4.1.3 Manipulating the Digits of a Number

Playing with the digits of a number is another type of challenge that can be proposed to students of the Introduction to Computer Programming course. Whether the digits are in decimal or binary form, this kind of exercise is interesting because it highlights another way of traversing a set of elements. Unlike arrays, where iterating over elements amounts to incrementing or decrementing a variable to go through the array's indexes, with numbers, we will be using the computer operations modulo, integer division, and powers instead. Integer division (or Euclidean division) is a mathematical operation in which one integer (the dividend, D) is divided by another (the divisor, d). The result of this operation is an integer (the quotient, q) and a natural number (the remainder, r). Mathematically, that gives $D/d = q + r$. In computing, this operation often ignores the remainder (i.e., $D/d = q$). So, we use the modulo operation, which returns the remainder of the whole division. That is denoted $a \bmod b$ or $a \% b$ (e.g., $12 \bmod 10 = 2$). Those two tools can be used to select one or more digits of a number to check certain properties (e.g., even or odd digit) or to calculate certain results (e.g., number of occurrences of a particular digit).

Selected Problems

The programming challenges for which the feasibility of an automatic GLI check has been analyzed are as follows:

8. Determine the reverse of a natural number (i.e., reverse the digits of the number).
9. Count the number of odd digits/occurrences of a digit in a natural number.

Specificities

We came across several issues when we studied Problems 8 and 9.

Once more, as in all previous programming challenges, the issue of bounds came

up again. As with arrays, the length of numbers (i.e., the number of digits) often acts as an upper bound for the iteration variable. We thus need to access this length. Given that the C language does not contain in its basic features a function to get the number of digits (i.e., the length) of a number, giving that length as an argument with the number itself is a worth-considering solution.

Then, as mentioned above, integer division, the modulo operation, and powers are used to step through the digits, making up a number. For example, we can access the third digit of an integer $d_3d_2d_1d_0$ (the first digit being the rightmost) through the operations $d_3d_2d_1d_0/10^2 \bmod 10$. We use integer division and powers to calculate $d_3d_2d_1d_0/100 = d_3d_2$ and the modulo operation to recover the required digit $d_3d_2 \bmod 10 = d_2$. In order to verify the GLIs associated with this type of program, we need to verify these types of operations.

4.1.4 Drawing on the Terminal

Drawing shapes in the terminal is another problem that students may have to solve. It may seem less valuable than the others, but it is not. This kind of challenge trains students to think in two dimensions. It is a way of introducing multi-dimensional arrays. Moreover, the shapes to be drawn always require determining mathematical definitions to define them.

Selected Problems

The programming challenges for which the feasibility of automatic GLI verification has been analyzed are as follows:

10. Draw a triangle/inverted triangle/hourglass in the command terminal using characters.

Specificities

The problems encountered (i.e., Problem 10) are always very similar (i.e., only the shape to be drawn changes).

Of course, once again, boundaries are a central issue in this kind of challenge. In the case of drawings, we are trying to create a string of ASCII characters, which will form the desired shape when printed in the terminal. We, therefore, need to establish the dimensions of the figure to be drawn. However, only a few dimensions are supplied in advance. The rest must be calculated based on the geometric properties of the figure in question. In these problems, the scale used is the character. For example, if one wants to draw a square, and the character is @, there is only the need of specifying how many '@' the square measures in height to define the square. In the case of a triangle, if they specify that it must be isosceles, then only the height about the side of different length is required (see Figure 4.2).



Figure 4.2: Drawings of a square and an isosceles triangle of height = 4

In addition to the notion of drawing limits, there is also the notion of shape limits. Using mathematical equations, we need to define which character should be in which position in the string. That is the regularity of the drawn shapes that makes it possible. The properties of the shapes, combined with the dimensions supplied as input, can be used to delimit zones in the character string. The right character can then be assigned to the correct zone. Therefore, the mathematical definition of a figure must be carefully considered if we want it to be correctly displayed.

4.1.5 Common Specificities

In analyzing problems 1 to 10, we noted that each type of problem had specific features. Even within the same group, variations can be from one challenge to another. However, some concepts came up again and again.

The first concept is about establishing the boundaries of the problem. Establishing the boundaries of the problem is an essential step. Thanks to these limits, we can then iterate on all the elements (numbers in a sequence, elements in an array, or digits in a number) and check that the program works correctly. One way or another, the GLI must contain this information about the upper and lower bounds of the problem. We must automate access to this information and check if these boundaries fit well with the context of the problem.

Once the limits of the problem have been defined, we need to find a way of verifying that the evaluated property is correctly computed during program iterations. That is a common specificity of every problem, too. This verification can be a mathematical formula, a logical expression, or a computer function. In Problem 1, for example, the mathematical formula $\frac{n \times (n+1)}{2}$ is used to calculate the sum of the first n natural numbers. In the case of prime numbers (Problem 3), we will use the logical expression:

$$n > 1 \wedge \neg (\exists d \in \{2, \dots, \lfloor \sqrt{n} \rfloor\} : n \bmod d = 0),$$

which is true only when n has no divisors other than 1 and itself. Computer functions are helpful when no logical expression or mathematical function can be found to describe the property being evaluated. In the case of Problem 4, we will need to be able to calculate the product of sequences, while in Problem 8, we will need to check that one number is the inverse of another. The special feature of these verification functions, whether

mathematical, logical, or computational, is that they must work for subsequences of the initial sequence of elements. Indeed, as we saw in Chapter 2, the invariant must hold before and after each iteration for a program to be verified by a loop invariant. So, for every problem, whatever it may be, we need to determine these verification functions. Particular attention must be paid to the correctness of these functions, as they are the basis for the correctness of the program. After verification, it can be said that the code produces a result that follows the verification functions. If these functions produce the wrong result at the outset, then the program, although considered to produce a correct result, will be erroneous.

4.2 Parsing GLI

As we saw in Section 2, the construction of a graphical invariant is based on a series of steps (i.e., construction rules). Each construction step provides the invariant with components essential to proper functioning and use (e.g., lower bound, structure type, etc.). This section will discuss what valuable information for GLI verification can be extracted from these components. As we saw in the previous section (Section 4.1), the verification of different types of programs is subject to specific criteria, such as compliance with bounds, preservation of data integrity, or the ability to verify program actions. All the elements required to meet these criteria are, by definition, represented in the GLI. Otherwise, the GLI alone could not establish the program's total accuracy. There are three main parts to a GLI, each providing a share of the data needed to verify the program. These three parts are the GLI pattern, the bars, and the text boxes.

4.2.1 Structure Type

The GLI pattern tells us which structure is used by the associated program. Each pattern plays an important role in determining the preconditions (i.e., conditions to be met before entering the loop) and how they are constructed.

- **Sequences of numbers**

The use of sequences implies that numbers must be ordered, often ascending. In the case of an interval $[a, b]$, the precondition $a \leq b$ must be satisfied.

- **Digits of numbers**

Calculating using a number's digits presupposes that the number's size (i.e., its number of digits) is strictly greater than zero. Indeed, a number without any digits can not be manipulated.

- **Arrays**

As with the digits of a number, when working with arrays, it can be assumed that they are not empty. In other words, their length is strictly greater than zero. Indeed, some functions, such as finding the index of the maximal value, will not be able to

return any value if the array size is null. Furthermore, with arrays, other conditions can be imposed. For example, the array may be initially sorted or unsorted. It may also have to contain a particular value, such as a terminating character.

- **Drawings**

In the case of drawings, the pattern specifies the symbols used to represent the figures.

Therefore, we need to extract information on the type of GLI used to build the preconditions and start the program in compliance with the initial conditions.

4.2.2 Bars

The bars answer two vital questions. What is the *Iterator* used to go through the different elements of a set, and what are the lower and upper bounds of the loop?

Iterator

The *Iterator* is indicated by the *Dividing Line*. That is the iteration variable in the box to the right or left of the *Dividing Line*. The other box is labeled $Iterator - 1$ or $Iterator + 1$, depending on whether you have chosen to put the iteration variable to the right or left of the *Dividing Line*, respectively. The *Iterator* plays a crucial role in the loop. Its value determines the progress of the program. It increases (or decreases) with each iteration until the final state is reached. In a GLI, this variation in value is visually represented by a shift of the *Dividing Line* (in red) to the right or left until the *To Do* zone (green zone) has completely disappeared. Figure 4.3 shows both processes. The situation for finding the maximum of an array (Problem 5) is shown in Figure 4.3b, and the situation for finding the number of odd digits of a number (Problem 9) is shown in Figure 4.3a. Once in the final state, the value of *Iterator* represents the loop's stop condition. In our two examples, the iteration variables are named i and are equal to $N + 1$ and N , respectively, the number of digits in the number n and the size of the array a .

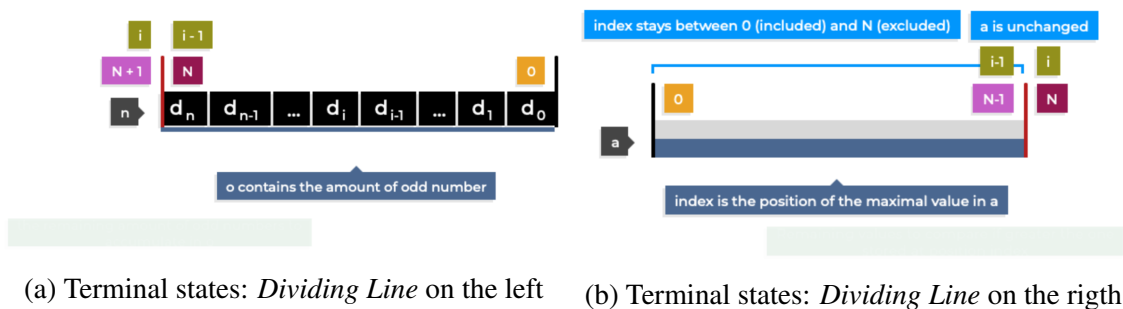


Figure 4.3: Terminal states

By analogous reasoning, we can determine the value of this *Iterator* for the initial state. To do that, we shift entirely the *Dividing Line* to the opposite side and make the

Done (blue zone) disappear.

The start and end values of the iteration variable then allow us to do two things. First, we can use them to deduce the code of the loop body (see Section 4.3.1). Then, they will enable us to establish one of the invariant clauses in Dafny (i.e., one of the *Annotations*).

Lower and Upper Bounds

The loop bounds are the values contained in the boxes at the ends of the GLI. The yellow boxes represent the lower bounds, and the pink/purple boxes represent the upper bounds. The reason for using a second color (the purple) for the upper bounds is to highlight the size of the structure described by the GLI (e.g., the number of elements in a sequence, the size of an array, the number of digits in a number, etc.). This size is also the value of the *Iterator* after the loop exit (i.e., after the increment or decrement that reached the loop stop condition).

While we established above that these bounds were used to define the range of values the iteration variable takes, they can also help delimit a domain for the *Accumulator*. We saw in Section 2.1.2 that this variable keeps track of the intermediate results calculated by the loop. In Figures 4.3a and 4.3b, these accumulation variables are labeled *o* and *index*. Their final value is often trivial. At the end of the loop, an *Accumulator* contains the desired value of the problem. That can be the number of odd digits or the maximum value *index*. However, bounds, especially the minimum bound, can be used to define the initial value of an accumulation variable. Depending on the position of the bound on the graphical invariant, the *Accumulator* will not have the same value on the first iteration of the loop. A good example is the multiplication of integers within an interval $[a, b]$ (Problem 2).

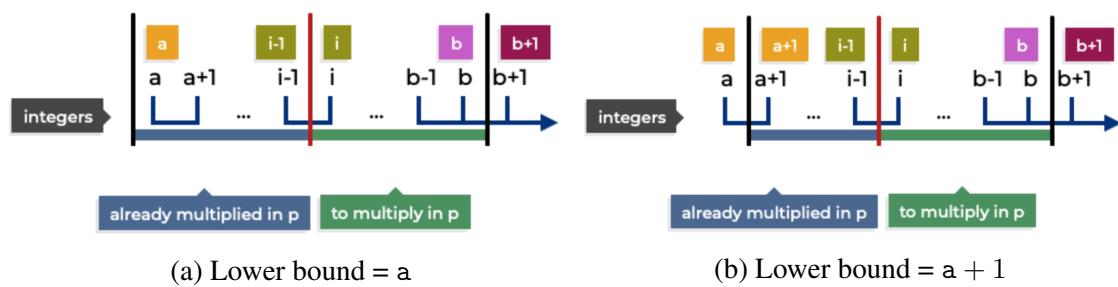


Figure 4.4: Lower bound placement

Figure 4.4 compares two cases. First, the case where the lower bound is a and second, the case where the lower bound is $a+1$ (the right-hand value is considered, as the *Iterator*, i , is to the right of the red line). In the first (Figure 4.4a), the accumulation variable p will have an initial value of 1 (i.e., the neutral element of the multiplication). In this way, it can be multiplied by $a, a+1, \dots, b$ to obtain the result. Whereas in the second case (Figure 4.4b), p will already be equal to a when entering the loop. One can observe that the second situation is equivalent to the first, but one iteration ahead. We will

avoid getting ahead of ourselves in loop iterations to keep as much generality as possible, preferring to define the lower bound as in Figure 4.4a.

This last example shows that the choice of bounds can influence the values of different variables in a program. Therefore, bounds will be essential during verification.

4.2.3 Text Zones

Text boxes are the components that give meaning to the GLI. They describe the loop's actions during iterations. In Section 2.1.2, we described them as the semantic elements of the image. Therefore, the content of these text zones is crucial for GLI verification. There are four types of text zone: the `var name` label, *Done* zone (in blue), *To Do* zone (in green), and *Conserved* zone (in light blue). Each zone provides information on the conditions the program must meet at different points in its execution.

- **Label** `var name`

This label defines either the type of value accepted by the GLI for number sequences (e.g., a sequence of integers or a sequence of natural numbers) or it represents the name of the structure on which we iterate (e.g., an array labeled `arr`, or a character form labeled `fig`). This text field allows us to refer to structures and their values using the right label and type. We need it to refer to structures in predicates and postconditions.

- ***Done* zone**

This zone contains one of the most essential information in the GLI. The *Done* zone uses words and phrases to describe the properties the program must check at each iteration. This text zone is responsible for checking the intermediate results that lead, at the loop exit, to the postcondition check. Thanks to this text field, we can assert a loop's correctness. Hence, we absolutely need to retrieve its contents to deduce the predicates to be checked. Moreover, the contained text must be sufficiently precise. The GLI adds a layer of abstraction that allows us to detach ourselves from logical and mathematical notations, replacing them with sentences. However, these sentences must be sufficiently precise to deduce more formal predicates that will help prove the loop's correctness. For example, in the case of finding the maximum of an array (Problem 5), depending on whether we are looking for the maximum value or its position in the array, the sentence will be slightly different. When searching for the value, the text may be "max is the maximal value of `arr`.". When searching for the position of the value, we prefer "index is the position of the maximal value in `arr`.". Although these two sentences describe a similar situation, the predicates deduced will be different. Verification might be impossible if the sentences are erroneous. We must thus pay particular attention to the semantics of the *Done* zone.

- ***To Do* zone**

This text field is of less interest for automatic verification. As we saw in the section

on GLI construction, the *To Do* zone has no formal equivalent. It is, however, of pedagogical interest, as it visualizes the loop's progression and facilitates the deduction of the Loop Variant. We will still use it in the GLI verification process to ensure the student indicates a sentence consistent with the *Done* zone.

- ***Conserved zone***

Like the *Done* zone, this text box contains properties that must be respected before, during, and after passing through the loop. The significant difference with the properties described in the *Done* zone is that those in the *Conserved* zone do not depend on the number of iterations already performed. In other words, they are properties that apply to all elements at each iteration. In the example of finding the maximum of an array (Problem 5), the property that requires the array not to be modified applies to all array elements, regardless of the loop's progress. Conversely, the maximum value position property index applies only to the part of the array already analyzed. For reasons identical to those mentioned for the *Done* zone, we absolutely must retrieve the contents of the *Conserved* zone to deduce the predicates to be checked. Likewise, the sentences contained in the zone must describe the predicates to be extracted with sufficient precision.

4.2.4 Automation with JSON Parsing

As we saw in the previous sub-section, a GLI contains a great deal of information essential for verifying a program. One must store the GLI image in a format that a computer program can easily manipulate to retrieve all this data. That is possible since CAFÉ 2.0 uses the current JSON format for storing GLIs. A concrete example is shown in Figure 4.5. It shows the JSON file in Figure 4.5b associated with the GLI of Problem 5 (finding the maximum of an array) in Figure 4.5a.

| Type | Value |
|---------------------|-------|
| Array | 0 |
| Number | 1 |
| Sequence of numbers | 2 |
| Enum | 3 |
| Matrix | 4 |
| Shape of characters | 5 |

Table 4.1: Values assigned to GLI types

The only field not currently present in the JSON format used by CAFÉ 2.0 is the type field. It can, however, be easily integrated into the current format. This field takes as its value an integer ranging from 0 to 5, depending on the type of GLI used (see Table 4.1. In our example, the 0 value represents an array.

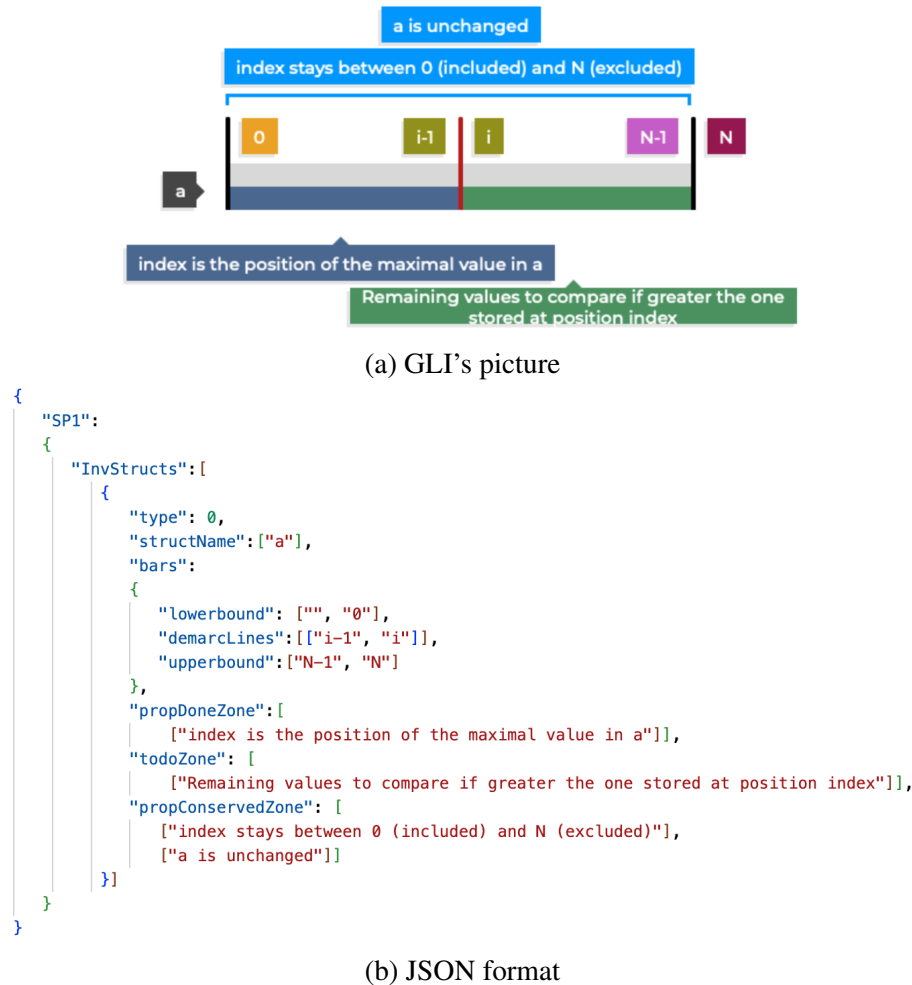


Figure 4.5: Representation of a GLI as a picture and a JSON file

GLI JSON Parser

To automate the analysis and extraction of GLI data, we have to write a program, for example, in Python, which will retrieve all the values contained in the GLI components. Suppose we use Blank GLI instead (see Section 3.2.2), then the analysis is even easier, as the JSON file is reduced to a simple assignment of values to boxes identified by a number.

For the purposes of the tests we wanted to conduct during this research project, we developed a program in Python capable of analyzing GLI in JSON format. This program, `parse_json`, implements two classes created for this purpose: `GLI` and `Annotation`. Here are their descriptions:

```

class Annotation:
    precondition: list[Tuple[Optional[Predicate], float]]
    postcondition_conserved: list[Tuple[Optional[Predicate], float]]
    postcondition_done: list[Tuple[Optional[Predicate], float]]
    invariant_iterator: list[Tuple[Optional[Predicate], float]]
    invariant_conserved: list[Tuple[Optional[Predicate], float]]
    invariant_done: list[Tuple[Optional[Predicate], float]]

```

```

class GLI:
    var_name: str
    type: int
    lower_bound: str
    upper_bound: str
    size: str
    iterator: list[str]
    done_text: list[str]
    todo_text: list[str]
    conserved_text: list[str]
    annotations: Annotation

```

The program takes the JSON file containing the GLI(s) as input and returns a dictionary containing one or more initialized GLI(s).

The value of type depends on the GLI pattern (see Table 4.1).

The iterator field contains the name of the iteration variable. However, this variable can be placed on either side of the *Dividing Line*. This piece of code allows us to determine which side the student has chosen to put their *Iterator*.

```

bars = struct.get("bars", {})
demarc = bars.get("demarcLines", [])
if not demarc or not demarc[0]:
    continue
demarc_pair = demarc[0]

iterator_var = next((v for v in demarc_pair if v.isidentifier()), None)
if iterator_var is None:
    continue
gli.iterator = [iterator_var]
idx = demarc_pair.index(iterator_var)

```

Then, using the position of the iteration variable relative to the *Dividing Line*, the program can fill in the lower_bound and size fields with the values (i.e., numbers or variable names) corresponding to the respective values of iterator (i.e., the iteration variable) at the initial and terminal positions (see Figure 4.7a and 4.7c). The upper_bound field, on the other hand, is simply the value located on the other side of the upper bound bar (i.e., size - 1).

The text areas (i.e., *Done*, *To Do*, *Conserved* area, and var_name) are represented by the fields var_name, done_text, todo_text, and conserved_text.

Finally, the annotations section will contain the Dafny *Annotations* extracted from the GLI (see Section 4.4).

4.3 Process from GLI to Dafny Syntax

Now that we have clearly defined the type of programs for which we wish to establish the feasibility of automatic verification of their GLIs and what information we own thanks to the analysis of GLIs, this section deals with the second and third parts of our methodology. These involve writing a program in C from a GLI (see Section 4.3.1) and then translating the program from C into Dafny (see Subsection 4.3.2).

These two steps are essential if we want to determine whether a student has correctly used the GLI to construct the loop. The aim is, firstly, to translate the C program into Dafny and secondly, to transform the GLI into Dafny *Annotations* (i.e., preconditions, postconditions, invariants). Once the translated code has been associated with the deduced *Annotations*, if the verification is successful, we can then assert that the program is consistent with the GLI. Of course, that will encourage us to think that the student has actually used the GLI to build his program, but we will not be able to state that with any certainty. On the other hand, if we aim to verify the GLI, we will assume that we have a correct Dafny program and associate the *Annotations* from the GLI with it. That will be discussed in more detail in the next section.

In the two subsections below, we will focus on building a loop in C based on a GLI and automating the translation from C to Dafny.

4.3.1 Writing C Code from GLI

The programming methodology used to move from a GLI to a C program is that used in the article “*Graphical Loop Invariant Based Programming*” by Brieven et al. [5]. This process is based on the general schema of a code containing a loop. This schema is shown in Figure 4, and is also taken from the work of Brieven et al. (“*Fig. 4. Loop zones and logical assertions. [..]*” [5]). Red boxes stand for the initial and final states, and blue boxes mark sections of the code with instructions to execute. The orange diamond represents the evaluation of the Boolean expression at each iteration. It is assumed that this evaluation cannot modify the truth value of the Loop Invariant. In other words, operations that cause edge effects (e.g., $--i$, $i++$, etc.) are excluded from the Boolean expression. The green ovals contain logical assertions, while the black arrows indicate the code’s direction. Just after Start and just before End are the Input and Output boxes. They indicate preconditions (e.g., a is not empty, and $length$ is equal to the length of a in the example provided in Chapter 2) and postconditions (e.g., $index$ is the position of the maximal value in a) that must be satisfied (i.e., true). Finally, the labels “yes” (i.e., Loop Condition holds) and “no” (i.e., Loop Condition does not hold) express the result of Loop Condition evaluation.

Figure 4.6 shows four parts that need to be completed: Zone 1, Zone 2, Zone 3 (with instructions), and Loop Condition (with a boolean expression). For the program to be correct, the content of these parts must use the logical assertions that precede them and guarantee those that follow. Completing each part is a step in the construction of the code.

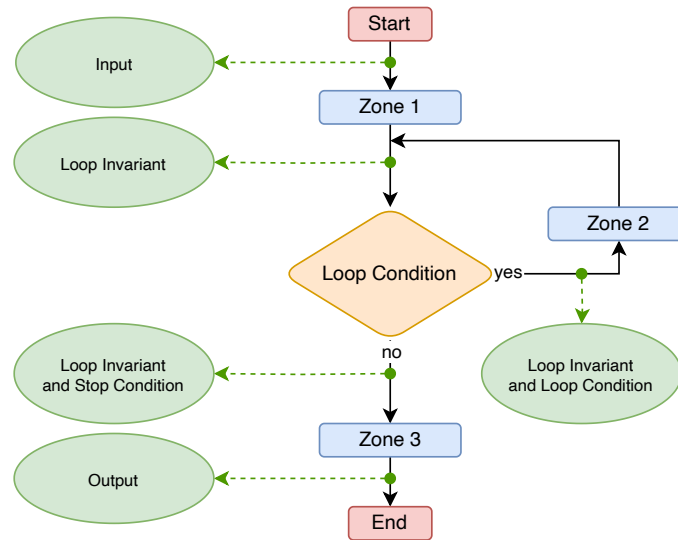


Figure 4.6: General pattern of an iterative loop

There is no set order for completing these steps, but it is generally easiest to proceed as follows:

1. Initializing the variable using the initial state (Zone 1).
2. Establishing the Stop Condition using the final state and deducing the Loop Condition.
3. Determining the instructions to progress the program, i.e., get closer to the final goal (Zone 2).
4. Deducing the last instructions (i.e., after the loop) to satisfy postconditions (Zone 3).

Each of these steps involves manipulations of the GLI. We can also note that this process is similar to the formal method discussed in Section 2.1.1, which is based on Hoare's Triplet [3, 13].

Zone 1

How to use the GLI to define the initial state was discussed in Section 4.2.2. Shift the *Dividing Line* completely to the left. The result is shown in Figure 4.7a. We can immediately deduce the initial value of the iteration variable *i* (i.e., the *Iterator*). As for the *index* variable, we know from the GLI (see Figure 2.4a) that it contains the position of the maximum of the part of the array already traversed. As we are in the initial state, we have not analyzed any elements in the array, so there is no known maximum yet. We therefore decide to initialize *index* to 0, assuming that the first element is the maximum. Note that the program works regardless of the initial value of *index* chosen, as long as it remains within the range of the array indexes. These two initializations are expressed in C below.

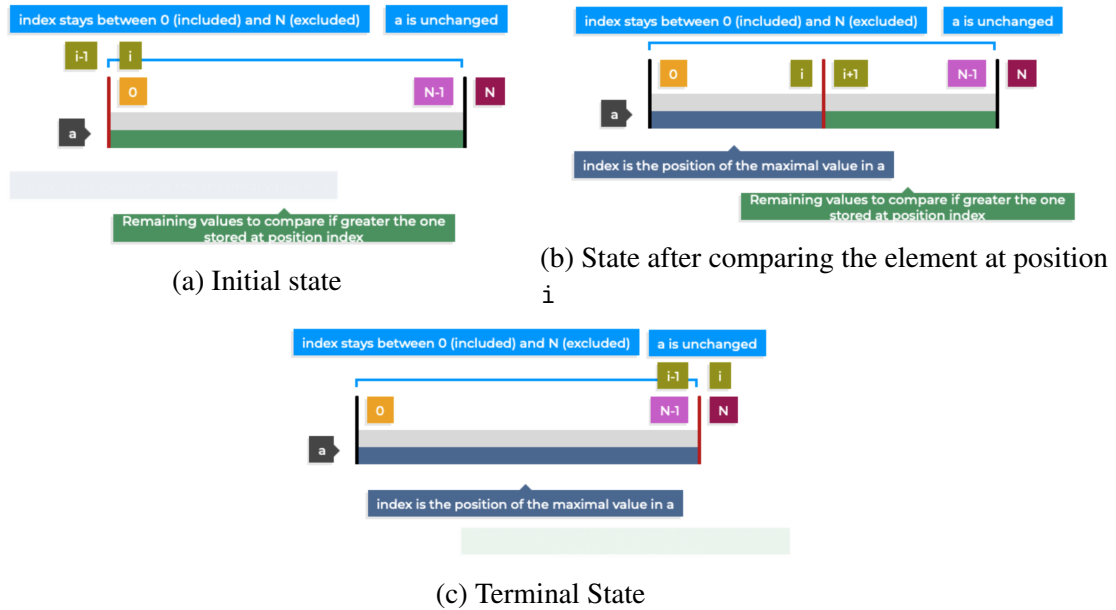


Figure 4.7: GLI manipulations to deduce C code

```

1 int i = 0;
2 int index = 0;

```

Stop and Loop Condition

Section 4.2.2 also detailed how, by shifting the *Dividing Line* completely to the right, we can determine the value of i at the end of the loop and deduce the Stop Condition. In our example, we can see from Figure 4.7c that the value of i is N , and therefore, the Stop Condition is $i = N$. The condition to be met to remain in the loop (i.e., the Loop Condition) is logically $i \neq N$, the negation of the Stop Condition. However, we prefer to use the stronger Loop Condition $i < N$ to better represent the state of the variable i when in the loop (i.e., $i < N$), and when out of it (i.e., $i \geq N$). The C code associated with this Loop Condition is:

```

1 while(i < N)

```

Zone 2

Determining the Loop Body instructions is the most delicate of the four steps. It is less systematic than the others and is often based on intuition acquired over time. However, a few things can give us food for thought. Firstly, we know that the GLI and the Loop Condition are true. We can, therefore, use these assumptions to try to improve the program. In the case of our example, the program progresses when we compare an element of the

array with the maximum of the elements previously analyzed (see *To Do* zone’s label in Figure 4.7b “Remaining values to compare [...]”). That suggests a comparison operation between the element at position *i* with that at position *index* held in memory. Then, depending on the result of this comparison, we will have to update the *index* variable. Indeed, if the value at position *i* is greater than that at position *index*, it becomes our new maximum, and so we assign position *i* to *index*. Otherwise, *index* remains the position of the maximum value, and we leave its value as is. This double case suggests an *if* statement that assigns *i* to *index* if the value at position *i* is greater than that at position *index*. Finally, Zone 2 always ends with instructions that guarantee the maintenance of the GLI. Since we have analyzed one additional element of the array, the GLI looks like the one shown in Figure 4.7b, whereas the initial GLI is shown in Figure 2.4a. We assign thus the value *i*+1 to *i* so that the GLI is restored (i.e., we increase *i*). The C code that summarizes this step is as follows:

```

1 if (a[i] > a[index]){
2     index = i;
3 }
4 i++;

```

Zone 3

In our example, Zone 3 is empty because the value of *index* at the output of the loop is the value we are looking for. However, that is not necessarily the case for all programs. Brieven gave the example of the mean computation, which, after the loop, requires dividing the sum by the number of elements before outputting the result.

Complete piece of code

Figure 4.8 shows the recap of all steps and indicates which part corresponds to which step.

```

int findMax(int a[], int N) Start
{
    int i = 0;
    int index = 0; Zone 1
    while (i < N) Loop Condition
    {
        if (a[i] > a[index])
        {
            index = i;
        }
        i += 1; Zone 2
    }
    return index; End
}

```

Figure 4.8: Complete piece of code with zone labels

4.3.2 Translation of C to Dafny Syntax

Once the C code has been written, it can be translated into Dafny. This step can, of course, be carried out by hand by a person responsible for correcting the GLIs and programs supplied by the students. Dafny’s syntax is, indeed, not very different from that of the C language, and a person with little experience in Dafny could easily do the translation. However, for our purpose of automatic verification, translating C code without external intervention is preferable. Bearing this in mind, we looked for research that fulfills that purpose. We soon came across research by Sriya S., Lavanya L., Aditi M. Manohar, and N. S. Kumar: “*Verification of C Programs using Annotations*” [23]. This article considers that ensuring the reliability of a computer program is essential and encourages the use of formal verification to demonstrate the correctness of a program as soon as it is compiled. The authors illustrate this approach by developing a C to Dafny translator using comments to replace Dafny *Annotations*. Their results show that such a translator is feasible. However, for the research work in progress, we did not have access to the source code of their translator ¹. However, we still wanted to be able to understand and illustrate how such a program works. So, we implemented a prototype Python translator based on the `pycparser` library. [2] library. Our Dumb Translator and the `pycparser` library will be briefly detailed below to understand the key points of the transition from C to Dafny. We will not go into the details of the program, which was not designed to be a final product usable by others.

Why Dafny ?

Before translating from C to Dafny, we feel that explaining why we chose this verification language is essential.

First of all, Dafny is a relatively easy-to-understand verification language. Although the process of verifying a program relies on complex formal methods, Dafny manages to make it accessible thanks to its system of *Annotations* (i.e., see Section 3.1.2 for `ensures`, `requires`, `invariant`, etc.). This abstraction layer means programmers can quickly grasp the language and write their first programs.

Moreover, the language is backed up by clear, comprehensive documentation. *The Dafny Reference Manual* [24] lists all the language’s features in great detail and provides examples for each part, significantly speeding up the learning process. The development team is also very active and regularly updates Dafny.

Another point that makes Dafny easy to use is its compatibility with the VSCode IDE. Following extensive feedback from the community complaining about the UI/UX ², the developers have implemented a visual feedback system [20] directly available from the VSCode interface. This system adds *Gutter Icons* and enhances positive (i.e., what

¹We chose not to request it, as the aim is studying the feasibility of automatic verification.

²User Interface/User Experience

has already been partially or fully verified) and negative (i.e., what has blocked Dafny’s verification) hover feedback [20]. All these additions really help users to learn from their mistakes and thus accelerate the rapid assimilation of how the language works.

Finally, the basic syntax of Dafny is similar in many respects to the basic syntax (i.e., syntax covered in a CS1 course such as INFO0946-1 [9]) of the C language. As we considered translating C code into Dafny code, these similarities were decisive in our choice of verification language.

Pycparser

The pycparser library is a Python library that provides a complete C parser (ISO C99), generating an Abstract Syntax Tree (AST) for browsing, analyzing, and transforming code.

In computer science, an AST is a representation of the structure of a piece of code. An AST uses a tree representation to store a representation of the syntactic structure of a computer language, such as C. Internal nodes represent operators. In contrast, leaves (i.e., external nodes) contain operands [25]. Figure 4.9 shows an example of an AST.

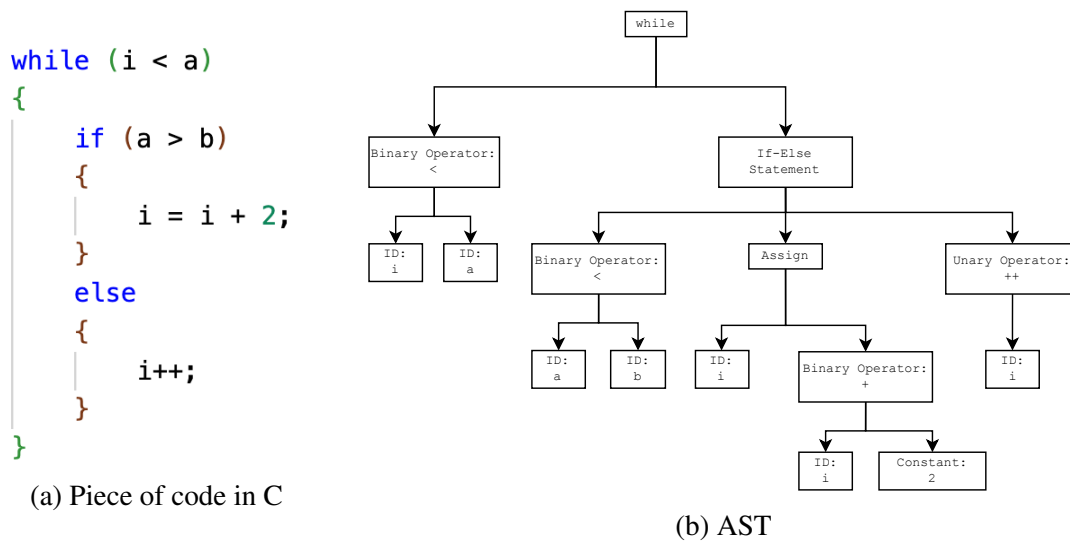


Figure 4.9: Example of an AST of a piece of code in C

The library parser pycparser is open source, and the complete code is available on the GitHub page ³ of its creator Eli Bendersky. The operation and the structure of the AST are explained in detail. Figure 4.9b shows the AST corresponding to the example shown in Figure 4.9a.

In addition to a parser, this library also contains a C code generator: CGenerator. It uses the AST created by the parser to generate the corresponding C code (i.e., the generator performs the inverse operation of the parser). For our work, we have modified this code generator to generate code in the Dafny language. The various points of divergence

³url: <https://github.com/eliben/pycparser>

```

While:
  BinaryOp: <
    ID: i
    ID: a
  Compound:
    If:
      BinaryOp: >
        ID: a
        ID: b
      Compound:
        Assignment: =
          ID: i
          BinaryOp: +
            ID: i
            Constant: int, 2
        Compound:
          UnaryOp: p++
          ID: i

```

Figure 4.10: AST generated by pycparser

between C and Dafny that have been addressed are presented below.

DafnyGenerator

As we have seen, this work is limited to an introduction to programming. Given that, the C syntax students use remains basic (i.e., they do not use all the advanced features of the language, and their programs are built with simple syntax). In this context, we implemented the Dumb Translator, *DafnyGenerator*. However, although C and Dafny have similarities in syntax, there are still some significant differences that we have to address.

- **Functions Declarations** C function declarations are not the same as in Dafny. To begin with, while Dafny functions exist, they do not have the same meaning as C functions. As discussed in Section 3.1.2, the Dafny object corresponding to the C function is the method. Figure 3.3a illustrates how to declare a method. *DafnyGenerator* then retrieves all the elements (i.e., function name, parameter names, types, and return variable type) that make up the C function declaration and uses them to write the corresponding method declaration.

```

1  'method ' + name + '(' + param + ')' ' + 'returns (rtn: ' + typ + '
    )\n'

```

- **Types Declarations**

Types are not the same in C and Dafny. We have, therefore, implemented a dictionary that associates the closest Dafny types with C types. In Dafny, types are declared after variables, and a colon separates the variable from its type.

```

1  types_dict = {
2      'unsigned': 'nat',
3      'long': 'int',
4      'float': 'real',

```

```

5      'double': 'real',
6      'void': ''
7      ...
8  }

```

• Variables Declarations

Unlike C, in Dafny syntax, the declaration of variables differs according to whether they are arguments to a method or declared in the body. The keyword `var` is added when the variable is declared in the method's body. DafnyGenerator therefore differentiates between the two cases.

```

1  if self.param:
2      s += self._generate_type(n.type)
3  else:
4      s += 'var ' + self._generate_type(n.type)

```

• Assignments

The assignment operator in Dafny is `:=`. DafnyGenerator therefore replaces all `=` (i.e., the assignment operator in C) with `:=` during assignments.

• Return Statements

As outlined above and in Section 3.1.2, the return variable is declared in the method declaration. That is an essential difference from C, where only the return type is specified in the function declaration. We, thus, added an extra step when the C program returns a value. First, we decided to systematically declare a return variable of the same type as the C function. This variable is `automatic_return_variable_name`. As this name can no longer be used in the function code, we opted for a long, very explicit name that is unlikely to be chosen by students. Then, at the time of the return instruction, the following expression is evaluated, and the result is assigned to `automatic_return_variable_name`. In the case of a function of type `void`, we remove the return value from the method declaration.

```

1  if n.expr:
2      s += 'automatic_return_variable_name := ' + self.visit(n.expr)
3      + ';\n'
4  s += self._make_indent() + 'return;'

```

• Loops

In Section 3.1.2, we explained that only `while` loops would be used in Dafny programs in this work, as a `for` loop could always be replaced by a `while` loop. However, given that students may use `for` loops, DafnyGenerator performs the syntax transition if it encounters a `for` loop during its traversal of the AST. It then

moves any variable declaration before the loop and adds the increment (or decrement) at the end of the code block. Similarly, DafnyGenerator can transform `do while` loops into `while` loops. To do that, it places a first copy of the loop body before the `while` keyword, and a second copy is placed after the keyword.

• Arrays

Arrays in Dafny are comparable to static arrays in the C language. In fact, an object of type `array<T>` in Dafny must have its size initialized at compile time. DafnyGenerator will, therefore, not accept a student dynamically allocating memory to create a variable-size array. The translator is, however, able to understand the following two syntaxes:

```
1  int t1[3];
2  int t2[] = {0, 0, 0};
```

Yet, Dafny cannot handle assigning values when declaring an array. To solve the problem, the translator replaces array initialization with the declaration of an array of equivalent size and initializes each element individually.

```
1  for i, expr in enumerate(n.init.exprs):
2      init += name + '[' + str(i) + ']:= ' + (self._visit_expr(expr))
        + '; '
3  s += ' := new ' + typ + '[' + str(len(n.init.exprs)) + ']; \n'
4  s += self._make_indent() + init[:-2]
```

• String Literals

The Dafny language we use in this work has no equivalent to C pointers. The closest structure is the `array<T>` type presented above. The only pointers that can be correctly translated to Dafny are `char *`, also known as *String Literals*. The Dafny type that supports such objects is `string`. That is a sequence of characters (i.e., type `seq<char>`). Like *String Literals*, an object of type `string` is immutable; modifying it is prohibited because it invokes undefined behavior. DafnyGenerator makes the difference between this declaration:

```
1  char *s = "Hello, world!";
```

which will be translated by an immutable type `string`. Whereas this one:

```
1  char s[] = "Hello, world!";
```

which becomes a `array<char>` (i.e., an array of characters) of length 15, each element of which has been initialized.

- **Syntactic Sugar**

Syntax shortcuts allowed by the C language, such as `+=` or `++`, are not supported by the Dafny language. However, they can all be replaced by equivalent complete expressions. Indeed, the three lines of code below are semantically equivalent:

```
1   i += 1;  
2   i++;  
3   i = i + 1;
```

DafnyGenerator replaces the first two lines with the third. It should be noted, however, that in the case of `i++`, the Dafny code will not be able to evaluate the value of `i` before incrementing it, as the C language does. Therefore, instructions such as `arr[i++]` should be avoided.

Undefined Functions

We have covered almost all the points of divergence between C and Dafny that our Dumb Translator can handle. One crucial fact remains: the use of external libraries. C programs often use `include` from the standard C library (e.g., `stdio.h`). Although there is a standard library in Dafny, it does not contain the same functions as those available in standard C libraries. That leaves us with two options. Either we implement the C to Dafny translation of the standard functions in DafnyGenerator, and the others will have to be implemented in Dafny. Alternatively, we decided to implement all the functions that could be used in the students' programs in Dafny. We have decided to opt for this second option in this work. We are dealing with simple C programs that do not require many calls to external functions. The use of functions such as `pow()`, `sqrt()`, or `floor()` will therefore require functions or methods to have been implemented beforehand. It will be up to the programmer to ensure the names and parameters match between C and Dafny declarations. These Dafny methods/functions can be collected in a single file and imported into the new file containing the translation using the `include "MyProgram.dfy"` instruction.

There is one special case: the `printf` function. That function can be used for problems involving form drawing. As loop verification is at the heart of the objective of this work, we need to constantly keep track of what is calculated by the loops. However, displaying intermediate results does not allow us to keep track of them. We, therefore, prefer to store the elements to be displayed and return them at the end of the loop. In this way, the part of the program that prints them out can be separated from the part to be checked.

Translated Piece of Code

Figure 4.11 shows the result of the translation of the C program (see Figure 4.8) corresponding to the example of finding the maximum of an array.

```

1  method findMax(a: array<int>, N: int) returns (automatic_return_variable_name: int)
2  {
3      var i: int := 0;
4      var index: int := 0;
5      while (i < N)
6      {
7          if (a[i] > a[index])
8          {
9              index := i;
10         }
11         i := i + 1;
12     }
13
14     automatic_return_variable_name := index;
15     return;
16 }

```

Figure 4.11: Translated piece of code using DafnyGenerator

4.4 Getting Dafny Annotations

We know henceforth how to retrieve the information contained in a GLI (see Section 4.2.4) and how to translate a C program, built from a GLI, into Dafny (see Section 4.3.2). However, we lack a crucial element to use the GLI to verify the C program. This element is *Dafny Annotations*. These enable the Dafny language to assert or deny the correctness of a computer program. Section 3.1.2 presented the various existing *Annotations* (i.e., `reads/modifies` for accessing arrays, `requires` for preconditions, `ensures` for postconditions, and `invariant` for loop invariants).

The formalization of the GLI involves extracting the *Annotations* necessary to verify the program's structure. The current problem is that this information is contained in the GLI in an informal form (i.e., images, colors, and words). In addition, in order to automate verification, we also need to define precisely where in the Dafny code we want the formal predicates to be inserted. Therefore, the goal is to formalize the GLI data and locate where we need to add it in order to use it in the Dafny code.

This section will discuss the two steps necessary for formalizing the GLI. The first will consist of standardizing the *Annotations* and their locations in the form of Dafny predicates. The second step will address the solution we have chosen to analyze the components of the GLI, retrieve its semantics, and use them to deduce the formal predicates hidden behind it.

4.4.1 Standardization

As we extended our methodology to different programming challenges, we realized that if we wanted to prove the feasibility of automatic verification of C programs using GLIs, we would need to establish regularity in using formal predicates.

With this goal in mind, we defined a standard schema to be completed using predicates to prove a program's validity. We also chose to standardize the way we declare our predicates to simplify the integration of variables extracted from GLIs.

Predicates Patterns

The pattern we have established is a template to be completed using predicates. This pattern is represented by the pseudo code below (the while loop in italics between square brackets is not always present):

Require: Challenge preconditions

Require: GLI_{ex} preconditions

Ensure: All Conserved zone properties

Ensure: GLI_{ex} Done zone

```

1: initex
2: while condex do
3:   Invariant  $GLI_{ex}$  Iterator
4:   Invariant  $GLI_{ex}$  conserved properties
5:   Invariant  $GLI_{ex}$  Done zone
6:   ...
7:   [initin
8:     while condin do
9:       Invariant  $GLI_{in}$  Iterator
10:      Invariant  $GLI_{in}$  conserved properties
11:      Invariant  $GLI_{in}$  Done zone
12:      ...
13:    end while
14:    endin]
15:   ...
16: end while
17: endex

```

When analyzing the pseudo code, we immediately notice that two GLIs can be involved in a single program. In that case, one will not insert predicates from different GLIs in the same locations in the Dafny code. To clearly define which predicates are to be inserted in which positions, we need to establish which GLIs they come from.

First, it should be noted that in the context of our work, a GLI defines a while loop and that we do not deal with problems involving more than two nested loops. When working with multiple GLIs, there are two possible scenarios. These GLIs are either associated with two nested loops (i.e., one loop inside another loop's body) or successive loops. The case of successive loops can easily be transposed back to manipulating a single GLI and a single loop by subdividing the problem into two subproblems. For example, Problem 7 (i.e., inserting one array into another at a given position) is divided into two loops and, therefore, into two successive GLIs. The first loop shifts the first array's elements by the second array's length from the given position. The second loop then inserts the elements of the second array into the newly created space. In the case of two nested loops, we have added a notion of hierarchy between the different GLIs associated with the

loops. In this way, we can differentiate between the outer GLI (i.e., describing the loop that contains the other) and the inner GLI (i.e., describing a loop inside the body of the other). Thanks to this hierarchy, whether we are working with a single GLI or two GLIs describing nested loops, we can determine the positioning of the predicates from each GLI.

This way of structuring predicates induces two characteristics that must have predicates from the *Done* zones of GLI_{in} and GLI_{ex} . The first is that the postcondition of the GLI_{in} must guarantee the verification of the GLI_{ex} . The second is that the instructions in the outer while loop must guarantee that the necessary preconditions for the GLI_{in} will be met.

Thanks to this pattern, we can consider automating the insertion of Dafny *Annotations*. Once we are able to extract formalized predicates from a GLI, we will know where in the Dafny code they should be inserted.

Predicate Declarations

After standardizing the insertion of predicates, it was essential to standardize their declaration. We therefore defined explicit labels that follow a regular pattern, as well as a list of arguments based on the information stored in the GLIs. This way, if other predicates are added in the future, the consistency of the code can easily be maintained.

Currently, the labels assigned to predicates take the form:

Is/Has + desc [+ UpTo]

For example, `IsIndex`, `IsMaxPosUpTo`, or `HasNoDivisorUpTo`. The suffix `UpTo` is used for predicates related to *Done* areas to describe their partial verification of the property (i.e., the property is verified for the part of the problem that has already been processed). This standard schema makes it easy to understand what the predicate does without reading its description or implementation.

Regarding implementation, the list of arguments that can be passed to predicates is strictly limited to the variables described in the GLI. In general, the various components of the GLI will be provided as arguments for all predicates. One will use them to verify their relationships when the predicates are called. If one wants to use a variable other than those that make up the GLI structure (i.e., lower and upper bounds, size, etc.), its label and nature must be specified in the area where it is mentioned. For example, in the sentence “index is the position of the maximal value in array a,” we find the label ‘index’ and the nature ‘position in the array a’.

The body of the predicate declaration is also standardized in the sense that it includes all the elements necessary to verify a property. For example, the predicate `IsProductUpTo` correctly calculates the partial product and then compares it with the variable that is supposed to contain the value of that product.

Thanks to this triple standardization (i.e., labels, arguments, and bodies of predicates),

we obtain a scalable list of formal predicates that can be used to prove a program's correctness.

4.4.2 Dictionnaire Solution

The issue addressed in this section concerns the transition from natural and visual language (i.e., words or images) to formal language (i.e., logical predicates). In order to select the predicates we have constructed, we have developed a description-based approach. The idea is to associate each Dafny predicate with a clear description of the property it verifies. It is then up to the programmer to ensure that the descriptions correspond only to the related predicates. That allows us to build a dictionary whose keys are the names of the predicates as presented in the previous section and whose values are the descriptions of these predicates.

Once this dictionary has been created, we can establish a link between the sentences contained in the text areas (i.e., semantic areas) of the GLI and the descriptions of the predicates. We will then be able to extract the meaning of the words the student has used in their GLI and deduce the formal predicates to be placed in the code.

Predicate Class

We have created a `Predicate` class to contain the name, descriptions, and list of arguments for each predicate we have implemented. Our Python class is as follows:

```

1  class Predicate(TypedDict):
2      name: str
3      descriptions: List[str]
4      args: List[str]
5      variables: List[str]
```

We have left the possibility of opening several descriptions for a single predicate. That will be useful when analyzing the similarity between the GLI semantic boxes and the descriptions. Here is an example of the use of this `Predicate` class for the predicate used in the case of searching for the maximum of an array (Problem 5):

```

1  "name": "IsMaxPosUpTo",
2  "descriptions": [
3      "VARIABLE_ est la position de la valeur maximale",
4      "VARIABLE_ est la position de la valeur maximale du tableau",
5      "la position de la valeur maximale est VARIABLE_"
6      "VARIABLE_ est la position de la plus grande valeur du tableau"
7      "la position de la plus grande valeur est VARIABLE_"
8  ],
9  "args": ["a: seq<int>", "size: nat", "lower_bound: nat", "upper_bound:
```

```
nat", "i: nat", "variable: nat"],
```

The descriptions are in French, as the course “Introduction to Computer Programming,” which provides the context for this research, is taught in that language. Students, therefore, fill in their GLIs with sentences in French. Of course, the result would be similar for GLIs containing descriptions in English.

Predicate Matcher

There are many ways to extract sentences’ semantics. Syntactic structure analysis, pattern matching, and statistical approaches all recover the meaning hidden behind a language’s words.

For this work, we opted for a method using deep neural networks and transformers. More specifically, our approach attempts to contextualize the words used by the student. In a nutshell, a model from the BERT family (Bidirectional Encoder Representations from Transformers) and, more precisely, the Sentence-BERT family (i.e., a system using advanced vector representations that produces more robust embeddings suitable for tasks like paraphrase detection) is used to calculate similarity scores between the sentences from the GLI and the descriptions of the predicates [19]. The Python module we use is SentenceTransformers [22]. It provides access to over 10,000 pre-trained Sentence Model Transformers and is available from the Hugging Face website. We will not go into detail about how this approach works, as it is beyond the scope of this research paper. That said, the field of sentence-pair regression tasks such as semantic textual similarity (STS) remains very interesting. Further research on the subject could undoubtedly enhance this part of our work.

Our Python class PredicateMatcher implements, using the Lajavaness/sentence-camembert-large model [16], a match function that compares a sentence provided as an argument to all of our predicate descriptions. It then returns the predicate associated with the highest similarity score it has calculated.

```
1 sent_emb = self._normalize(
2     self.model.encode(self._preprocess(sentence))
3 )
4 scores = np.dot(self.predicate_embeddings, sent_emb).tolist()
5 best_idx = int(np.argmax(scores))
6 best_score = scores[best_idx]
```

The similarity between two vectors (i.e., vectorized sentences) is usually evaluated by calculating the cosine similarity, which is mathematically described by the following equation:

$$\text{cosine_sim}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|}.$$

We calculate the similarity score in two steps in the match function. First, we normalize the two vectors using `_normalize`, and then we perform the scalar product of the two embeddings using `np.dot`. The associated predicate is finally found using the index of the best score. Since it is a cosine, the score is in the interval $[-1, 1]$. A score of 1 represents a perfect match between two sentences, and a score of -1 represents a total divergence between their respective semantics.

It is worth noticing that if we are in the case of Blank GLI (i.e., GLI to be completed with elements from a list, see Section 3.2.2), the cosine similarity will be equal to 1. If we indeed put the correct answers among the descriptions of the predicates, we will get exact matches.

Models

The model we chose for our tests is `Lajavaness/sentence-camembert-large` [16]. That is an Embedding Model specialized for French and developed by La Javaness [15].

We did not have enough data to make a meaningful comparison between several models. That explains why there is no comparison in this report. Many others could, indeed, be just as suitable. Since our goal was to demonstrate the feasibility of the automatic verification process, we did not spend much time choosing a model. We still compared a few using the Python script `evaluate_predicates` and opted for the one that provided the best results. However, as these tests were not thorough, we would not venture to claim that our model is the best of those available. Moreover, in the context of a course taught in English, if we wanted to give students the option of completing their GLIs in English, a multilingual model, or even a model pre-trained on English-language texts, would be a better choice.

The model we chose combines the advantage of being available free online with the fact that it is light enough to be used locally (i.e., we downloaded it). It is also accurate enough to provide results that demonstrate the feasibility of the semantic recognition method. However, in order to help it somewhat, we decided to highlight the names of the variables in the semantic zone phases (i.e., *Done* zone, *Conserved* zone, etc.). One does the highlighting by enclosing the variables in curly brackets. In this way, a function can prepare the sentences to be encoded by replacing the names of these variables with the token `VARIABLE_`. That standardizes the sentences that could be passed to the `match` function, making it easier for the model to compare the sentences with the predicate descriptions. We took the opportunity to retrieve the variable labels and include them in our `Predicate` class. This addition significantly increases the similarity scores between the tested sentences and the predicate descriptions. If we used larger, better-trained models, this pre-processing would no longer be necessary, and we could allow users complete freedom to enter any sentence they wish. If we ever want to deploy such a verification system, we must consider choosing more powerful models.

False Positive matches

As described above, our method for extracting formal predicates poses a significant problem. This problem concerns false positives. According to our match function, the predicate returned has the highest similarity score. However, there is nothing to ensure that this score is actually high (i.e., close to 1). Therefore, we added a threshold below which two sentences are no longer close enough to assume that they describe the same property.

```
1 if best_score < self.threshold:
2     return None, round(best_score, 4)
3
4 pred_idx = self.desc_pred_map[best_idx]
5 matched_pred = self.predicates[pred_idx]
6 return matched_pred, round(best_score, 4)
```

This minimum similarity threshold prevents an incorrect sentence from a GLI from being translated by a correct predicate. It should be noted that the choice of this threshold is a decisive factor in the verification process. If it is too high, sentences describing correct predicates will not be considered sufficiently close to the descriptions. At the same time, if it is too low, any sentence from the text areas could be associated with a correct predicate. We determined the threshold the same way that we chose our model, by running some tests in the Python script `evaluate_predicates`.

Chapter 5

Results

Now that we have detailed all the steps represented in the big picture (see Figure 4.1), it is time to summarize our process and apply it to our 10 programming challenges (see Section 4.1).

This chapter covers our process in its entirety and presents all the results we obtained. Section 5.1 summarizes the entire process and describes explicitly the case of the example of finding the maximum value in an array, which we have been following since the beginning of this work. Next, Section 5.2 summarizes the results obtained for all the other programming exercises we have addressed in this work. Section 5.3 details the limitations of our method, and Section 5.4 lists all the files available on the GitLab page associated with this work.

5.1 Complete Process

As a reminder, we can verify two things. The first is the verification of a GLI in its entirety. That is possible if we have a verified Dafny program. We must remove the Dafny *Annotations* and try to find them again starting from the GLI. If the GLI allows us to find formal predicates, which will enable Dafny to verify the program, then the GLI is also verified. However, we will not present the results of this specific part of the process, as the second verification option entirely covers it.

The second way to use the system we have developed is to verify a C program derived from a GLI using that same GLI. To do this, let us go through the steps in order. There are actually two paths to the process that ultimately converge.

The first consists of deducing the C code for the program from a GLI. That is the job of the person who built the GLI (i.e., in our case, a student) and is described in Section 4.3.1. Next, the resulting C program is transformed into an AST before being translated into Dafny (see Section 4.3.2). Once translated, only one thing is needed to guarantee the accuracy of the program: Dafny *Annotations* (i.e., formal predicates). Retrieving these *Annotations* is the task of the second part of the process.

We take the original GLI in JSON format, which a program can understand. This JSON file is then parsed, and the essential information is extracted. This important information includes, in particular, the sentences that the student has inserted into the GLI to describe the properties that the GLI must verify for the program to be correct. In order to confirm whether these sentences represent the expected properties, we have created a dictionary that links formal predicates to natural language (i.e., in our case, English) via descriptions. The <key, values> pairs in this dictionary are the names and descriptions of the predicates. These sentences are then passed through a Python script that attempts to extract their semantics by comparing them to the descriptions in our dictionary. Once the name of the predicate is found, it is added to the essential information in the GLI. We are left with a list of elements forming the formalized GLI together.

That is where we find the Dafny code obtained previously. All that remains is integrating the formalized predicates into the Dafny code by following the standard procedure described in Section 4.4.1. That gives us a complete Dafny program that one can verify via the terminal by entering the command:

```
$ dafny verify filename.dfy
```

5.1.1 Research of Maximum

We will apply our method to the problem of finding the maximum value in an array (Problem 5).

1. GLI

Each construction challenge begins with the design of the appropriate GLI. The students carry out this step, following the construction rules detailed in Section 2.1.2. In the case of our example, a student should obtain a GLI similar to the one shown in Figure 5.1.

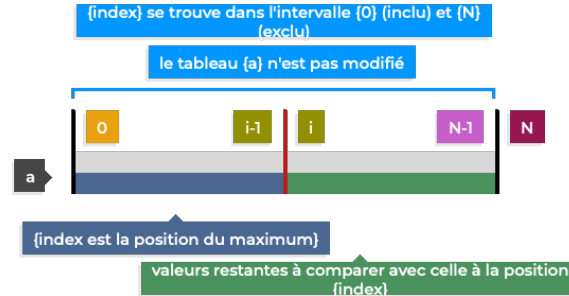


Figure 5.1: Potential GLI designed by a student

2. C program

The student is then asked to derive the C code from the GLI they have drawn. The steps involved in this part were presented in Section 4.3.1. By following them, a student using the GLI in Figure 5.1 will obtain a C program like this one.

```

1  unsigned find_maximum(int a[], unsigned N)
2  {
3      unsigned i = 0;
4      unsigned index = 0;
5      while (i < N)
6      {
7          if (a[i] > a[index])
8          {
9              index = i;
10         }
11         i += 1;
12     }
13     return index;
14 }
```

At this stage, the student's work ends, and our process begins. The goal is to verify the C program using the student's GLI.

3. Dafny code

Once the program has been written, it is passed to the automatic translator, which will parse the code, create an AST to represent its structure, and traverse this AST to generate an equivalent program, but one that uses the Dafny language syntax. Once the translation is complete, we obtain the following Dafny code:

```

1  include "../dafny_prog/benchmarks/predicates.dfy"
2  include "../dafny_prog/benchmarks/help_predicates_functions.dfy"
3
4  method find_maximum(a: array<int>, N: nat) returns (
5      automatic_return_variable_name: nat)
6  {
7      var i: nat := 0;
8      var index: nat := 0;
9      while (i < N)
10     {
11         if (a[i] > a[index])
12         {
13             index := i;
14         }
15         i := i + 1;
16     }
17
18     automatic_return_variable_name := index;
19     return;
20 }

```

We can see that all the slight differences between the syntax of C and Dafny have been considered and modified. We also find the `include` commands, allowing us to use the previously defined formal predicates. All that is missing now are the Dafny *Annotations* to verify whether the program works correctly in the sense of the GLI.

4. Formalized GLI

We therefore need to formalize the GLI constructed by the student. To do this, we will extract the essential information. Figure 5.1 shows a GLI in its pictorial form, but our program `GLI_json_parser` works instead with the JSON format in which GLIs are stored and communicated by CAFÉ 2.0. For our example, this file looks like:

```

{
  "SP1": {
    "InvStructs": [
      {
        "type": 0,
        "structName": ["a"],
        "bars": {
          "lowerbound": ["", "0"],
          "demarcLines": [
            ["i-1", "i"]
          ],
          "upperbound": ["N-1", "N"]
        },
        "propDoneZone": [
          ["{index} est la position de la valeur maximale du tableau {a}"]
        ],
      },
    ],
  },
}

```



```

        "todoZone": [
            ["valeurs restantes à comparer avec celle de la position {index}"]
        ],
        "propConservedZone": [
            ["{index} se trouve dans l'intervalle {0}(inclu) et {N} (exclu)",
            ["{a} n'est pas modifié"]
        ]
    }
}

```

Based on this JSON file, `GLI_json_parser` retrieves the various pieces of information contained in the GLI. Information about the structure of the GLI (i.e., which patterns), about the bars (i.e., lower bound, upper bound, and iteration variable), and about the semantic zones (i.e., *Done*, *To Do*, and *Conserved* zones) is analyzed and assembled to form a formalized GLI. As for the sentences contained in the semantic zones, they are compared to the descriptions of the formal predicates implemented, and if their similarity is deemed sufficient, then these predicates are included in the formalized GLI. In the case of the problem we are currently dealing with, the output is a new JSON:

```

{
  "SP1": {
    "type": 0,
    "var_name": "a",
    "lower_bound": "0",
    "upper_bound": "N-1",
    "size": "N",
    "iterator": ["i"],
    "done_text": [
      "{index} est la position de la valeur maximale du tableau {a}"
    ],
    "todo_text": [
      "valeurs restantes à comparer avec celle de la position {index}"
    ],
    "conserved_text": [
      "{index} se trouve dans l'intervalle {0}(inclu) et {N} (exclu)",
      "{a} n'est pas modifié"
    ],
    "annotations": {
      "precondition": [
        [{"name": "IsArraySize",
          "descriptions": [
            "La longueur du tableau est fournie"
          ],
          "args": [{"struct": "seq<int>", "size": "int"}],
          "variables": []
        }],
        0.9738
      ],
      "postcondition_conserved": [
        [{"name": "IsIndex",
          "descriptions": [
            "l'indice VARIABLE_ est compris entre VARIABLE_ et VARIABLE_",
            "VARIABLE_ se trouve dans les bornes du tableau",
            "l'indice VARIABLE_ est dans l'intervalle VARIABLE_ inclu et VARIABLE_ exclu",
            "VARIABLE_ reste entre VARIABLE_ (inclu) et VARIABLE_ (exclu)"
          ],
          "args": [{"size": "int", "lower_bound": "int", "upper_bound": "int", "variable": "int"}],
          "variables": ["index", "0", "N"]
        }],
        0.9103
      ],
      [{"name": "IsUnchanged",
        "descriptions": [
          "le tableau VARIABLE_ est n'est pas modifié",
          "les l'ordre et valeurs de VARIABLE_ ne sont pas modifiées",
          "l'ordre et les valeurs des éléments de VARIABLE_ ne sont pas modifiées"
        ],
        "args": [{"struct": "seq<int>", "oldStruct": "seq<int>"},
        "variables": ["a"]
      }],
    }
  }
}

```

```

    },
    0.8232]
  ],
  "postcondition_done": [
    [{"name": "IsMaxPosUpTo",
      "descriptions": [
        "VARIABLE_ est la position de la valeur maximale",
        "VARIABLE_ est l'indice de la valeur maximale du tableau",
        "la position de la plus grande valeur est VARIABLE_",
        "VARIABLE_ est l'indice du maximum du tableau",
        "la position du maximum est VARIABLE_"
      ],
      "args": ["struct: seq<int>", "size: nat", "lower_bound: nat", "upper_bound: nat", "i: nat", "variable: nat"],
      "variables": ["index", "a"]
    }],
    0.9247]
  ],
  "invariant_iterator": [
    [{"name": "IsWithinBounds",
      "descriptions": [
        "la variable d'itération est dans dans les bornes"
      ],
      "args": ["size: int", "lower_bound: int", "upper_bound: int", "iterator: int"],
      "variables": []
    }],
    1.0]
  ],
  "invariant_conserved": [
    [{"name": "IsIndex",
      "descriptions": [
        "l'indice VARIABLE_ est compris entre VARIABLE_ et VARIABLE_",
        "VARIABLE_ se trouve dans les bornes du tableau",
        "l'indice VARIABLE_ est dans l'intervalle VARIABLE_ inclu et VARIABLE_ exclu",
        "VARIABLE_ reste entre VARIABLE_ (inclu) et VARIABLE_ (exclu)"
      ],
      "args": ["size: int", "lower_bound: int", "upper_bound: int", "variable: int"],
      "variables": ["index", "0", "N"]
    }],
    0.9103],
    [{"name": "IsUnchanged",
      "descriptions": [
        "le tableau VARIABLE_ est n'est pas modifié",
        "les l'ordre et valeurs de VARIABLE_ ne sont pas modifiées",
        "l'ordre et les valeurs des éléments de VARIABLE_ ne sont pas modifiés"
      ],
      "args": ["struct: seq<int>", "oldStruct: seq<int>"],
      "variables": ["a"]
    }],
    0.8232]
  ],
  "invariant_done": [
    [{"name": "IsMaxPosUpTo",
      "descriptions": [
        "VARIABLE_ est la position de la valeur maximale",
        "VARIABLE_ est l'indice de la valeur maximale du tableau",
        "la position de la plus grande valeur est VARIABLE_",
        "VARIABLE_ est l'indice du maximum du tableau",
        "la position du maximum est VARIABLE_"
      ],
      "args": ["struct: seq<int>", "size: nat", "lower_bound: nat", "upper_bound: nat", "i: nat", "variable: nat"],
      "variables": ["index", "a"]
    }],
    0.9247]
  ]
}
}
}
}

```

5. Insertion of Annotations

Using the standard scheme presented in Section 4.4.1, we can finally include the formal predicates found in the appropriate locations in the Dafny code. The result is shown in Figure 5.2

6. Verification

In the IDE, we can immediately see that the program has been verified thanks to the *Gutter* (see Figure 3.1a) next to the code. However, we can also verify that via the

```

1  include "../dafny_prog/benchmarks/predicates.dfy"
2  include "../dafny_prog/benchmarks/help_predicates_functions.dfy"
3
4  method find_maximum(a: array<int>, N: nat) returns (automatic_return_variable_name: nat)
5  {
6    requires IsArraySize(a[..], N)
7    requires IsNoEmptyArray(a[..], N)
8    ensures IsIndex(N, 0, N-1, automatic_return_variable_name)
9    ensures IsUnchanged(a[..], old(a[..]))
10   ensures IsMaxPosUpTo(a[..], N, 0, N-1, N, automatic_return_variable_name)
11   {
12     var i: nat := 0;
13     var index: nat := 0;
14     while (i < N)
15     {
16       invariant IsWithinBounds(N, 0, N-1, i)
17       invariant IsUnchanged(a[..], old(a[..]))
18       invariant IsIndex(N, 0, N-1, index)
19       invariant IsMaxPosUpTo(a[..], N, 0, N-1, i, index)
20       {
21         if (a[i] > a[index])
22         {
23           index := i;
24         }
25         i := i + 1;
26       }
27     }
28     automatic_return_variable_name := index;
29     return;
30   }
31 }

```

Figure 5.2: Completed Dafny program

command line:

```
$ dafny verify max.dfy
```

We then obtain the verification result.

```
Dafny program verifier finished with 2 verified, 0 errors
```

An error message is displayed if an error occurs (e.g., in the predicate parameters). For example, if we had written `invariant IsWithinBounds(N+1, 0, N-1, i)`, the message would have been:

```

max.dfy(17,26): Error: function precondition could not be proved
|
17 | invariant IsMaxPosUpTo(a[..], N, 0, N-1, i, index)
| ~~~~~

/Users/antoinegrosjean/Desktop/ULG/M2/TFE/git/tfe/dafny_prog/benchmarks/predicates.dfy
(118,19): Related location: this proposition could not be proved
|
118 | requires lb <= i <= size
| ~~~~~

```

7. Remarks

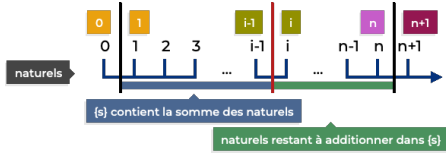
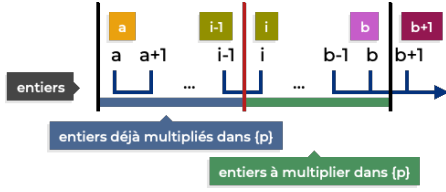
The array must not be empty for the program to be verified. By definition, an empty array has no elements and, therefore, no position to return. Dafny can thus not verify the program because it cannot establish the return value. To avoid this situation, we will assume that the array provided contains at least one element. Therefore, that

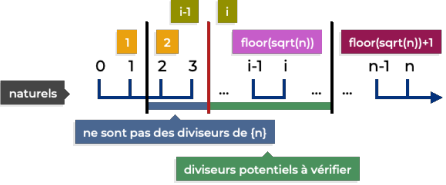
must be guaranteed before calling the function (e.g., via an `if` statement).

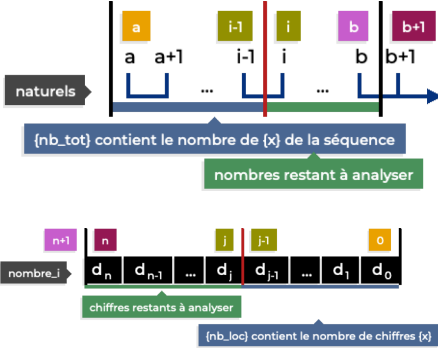
That concludes the results we obtained for the case of searching for the maximum value in an array.

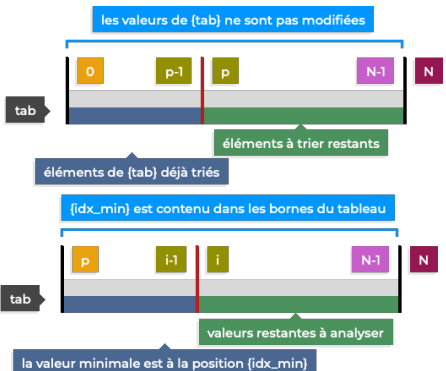
5.2 Compilation of Results

Table 5.1 summarizes the verification process steps for all the problems we tested. It follows the same pattern as the one detailed for Problem 5 (searching for the maximum).

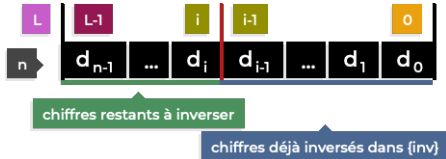
| Problems | Results | |
|----------|---|--|
| 1 | Challenge: Calculate the sum of the first natural numbers. | |
| |  | <pre> int add(unsigned n) { unsigned i = 1; int s = 0; while (i < n + 1) { s += i; i++; } return s; } </pre> |
| | GLI | C program derived from GLI |
| | <pre> {"SP1":{ "type":2, "var_name":"naturels", "lower_bound":"1", "upper_bound":"n", "size":"n+1", "iterator":["i"], "annotations":{ "precondition":[{"name":"IsSequence"}], "postcondition_done":[{"name":"IsSumUpTo"}], "invariant_iterator":[{"name":"IsWithinBounds"}], "invariant_done":[{"name":"IsSumUpTo"}] } } </pre> | <pre> method add(n: nat) returns (int) { var i: nat := 1; var s: int := 0; while (i < n + 1) { s := s + i; i := i + 1; } return s; } </pre> |
| | Formalized GLI overview (JSON) | Automatically translated Dafny code |
| | Output: Dafny verifier finished with 2 verified, 0 errors | |
| | Remarks: none. | |
| 2 | Challenge: Calculate the product of integers in the interval $[a, b]$. | |
| |  | <pre> int prod(int a, int b) { int i = a; int p = 1; while (i < b + 1) { p *= i; i++; } return p; } </pre> |

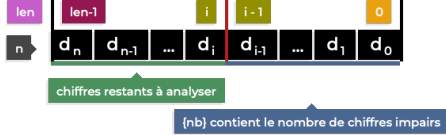
| Problems | Results | |
|----------|---|--|
| | GLI | C program derived from GLI |
| | <pre> {"SP1": { "type": 2, "var_name": "entiers", "lower_bound": "a", "upper_bound": "b", "size": "b+1", "iterator": ["i"], "annotations": { "precondition": [{"name": "IsSequence"}], "postcondition_conserved": [], "postcondition_done": [{"name": "IsProductdUpTo"}], "invariant_iterator": [{"name": "IsWithinBounds"}], "invariant_conserved": [], "invariant_done": [{"name": "IsProductdUpTo"}] } } </pre> | <pre> method prod(a: int, b: int) returns (automatic_return_variable_name: int) { var i: int := a; var p: int := 1; while (i < (b + 1)) { p := p * i; i := i + 1; } automatic_return_variable_name := p; return; } </pre> |
| | Formalized GLI overview (JSON) | Automatically translated Dafny code |
| | Output: Dafny verifier finished with 2 verified, 0 errors | |
| | Remarks: none. | |
| 3 | Challenge: Check whether a number is prime. | |
| | GLI | C program derived from GLI |
| |  <pre> {"SP1": { "type": 2, "var_name": "naturels", "lower_bound": "2", "upper_bound": "floor(sqrt(n))", "size": "floor(sqrt(n))+1", "iterator": ["i"], "annotations": { "precondition": [{"name": "IsSequence"}], "postcondition_conserved": [], "postcondition_done": [{"name": "HasNoDivisorUpTo"}], "invariant_iterator": [{"name": "IsWithinBounds"}], "invariant_conserved": [], "invariant_done": [{"name": "HasNoDivisorUpTo"}] } } </pre> | <pre> int prime(unsigned n) { if (n < 2) { return 0; } for (int i = 2; i < floor(sqrt(n)) + 1; i++) { if (n % i == 0) { return 0; } } return 1; } </pre> <pre> method prime(n: nat) returns (automatic_return_variable_name: int) { if (n < 2) { automatic_return_variable_name := 0; return; } var i: int := 2; while i < (floor(sqrt(n)) + 1) { if ((n % i) == 0) { automatic_return_variable_name := 0; return; } i := i + 1; } automatic_return_variable_name := 1; return; } </pre> |

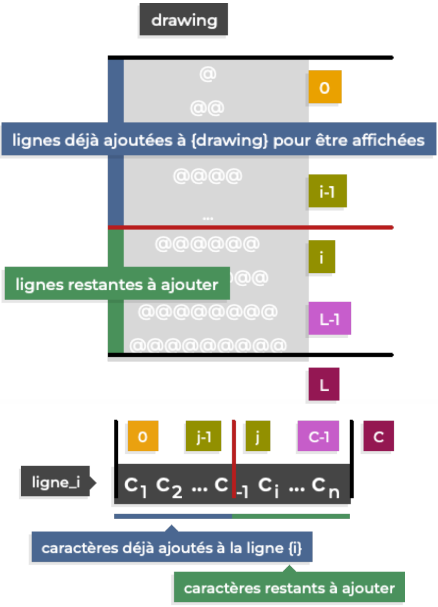
| Problems | Results | |
|----------|---|-------------------------------------|
| | Formalized GLI overview (JSON) | Automatically translated Dafny code |
| | Output: Dafny verifier finished with 2 verified, 0 errors | |
| | Remarks: The prime method differs from the others because it can output two results. It returns 1 if n is prime and 0 otherwise. We must thus compare the value of our predicate <code>HasNoDivisorUpTo</code> with the returned value. <code>HasNoDivisorUpTo</code> must be true if and only if the return value is 1. Therefore, we changed the postcondition schema and added a <code><==></code> check. This solution will be used for all other problems returning several possible values (e.g., a boolean function like <code>prime</code>). | |
| 4 | Challenge: Count the number of occurrences of a digit in a given sequence of numbers. <div style="display: flex; justify-content: space-between; align-items: flex-start; padding: 10px;"> <div style="width: 45%;">  <p>The diagram illustrates the counting process. The top part shows a sequence of natural numbers from 'a' to 'b+1'. A red vertical line is at 'i'. Below, a sequence of digits 'd_n' to 'd_0' is shown, with a red vertical line at 'j'. Labels indicate the current state of counters and the remaining elements to be analyzed.</p> </div> <div style="width: 50%;"> <pre> int count_digit(unsigned a, unsigned b, unsigned x) { unsigned i = a; unsigned nb_tot = 0; while (i < b + 1) { unsigned nb_i = i; unsigned len = lengthOf(nb_i); unsigned nb_loc = 0; unsigned j = 0; while (j < len) { unsigned d = (nb_i / (int)pow(10, j)) % 10; if (d == x) { nb_loc++; } j++; } nb_tot += nb_loc; i++; } return nb_tot; } </pre> </div> </div> <div style="display: flex; justify-content: space-between; margin-top: 10px;"> GLI C program derived from GLI </div> | |

| Problems | Results | |
|----------|--|---|
| | <pre> {"SP1": { "type": 2, "var_name": "naturels", "lower_bound": "a", "upper_bound": "b", "size": "b+1", "iterator": ["i"], "annotations": { "precondition": [{"name": "IsSequence"}], "postcondition_conserved": [], "postcondition_done": [{"name": "IsGlobalCountUpTo"}], "invariant_iterator": [{"name": "IsWithinBounds"}], "invariant_conserved": [], "invariant_done": [{"name": "IsGlobalCountUpTo"}] } }, "SP2": { "type": 1, "var_name": "nb_i", "lower_bound": "0", "upper_bound": "len-1", "size": "len", "iterator": ["j"], "annotations": { "precondition": [], "postcondition_conserved": [], "postcondition_done": [], "invariant_iterator": [{"name": "IsWithinBounds"}], "invariant_conserved": [], "invariant_done": [{"name": "IsLocalCountUpTo"}] } } } </pre> | <pre> method count_digit(a: nat, b: nat, x: nat) returns (automatic_return_variable_name: int) { var i: nat := a; var nb_tot: nat := 0; while (i < (b + 1)) { var nb_i: nat := i; var len: nat := lengthOf(nb_i); var nb_loc: nat := 0; var j: nat := 0; while (j < len) { var d: nat := (nb_i / (pow(10, j) as int)) % 10; if (d == x) { nb_loc := nb_loc + 1; } j := j + 1; } nb_tot := nb_tot + nb_loc; i := i + 1; } automatic_return_variable_name := nb_tot; return; } </pre> |
| | Formalized GLI overview (JSON) | Automatically translated Dafny code |
| | Output: Dafny verifier finished with 2 verified, 0 errors | |
| | Remarks: The C function count_digit requires a function to get the number of digits of a number. We used lengthOf and ensured it had been verified. | |
| 6 | Challenge: Sort an array by implementing the Selection Sort algorithm. | |
| |  | <pre> void selectionSort(int tab[], unsigned N) { for (int p = 0; p < N; p++) { int idx_min = p; for (int j = p; j < N; j++) { if (tab[j] < tab[idx_min]) { idx_min = j; } } int tmp = tab[p]; tab[p] = tab[idx_min]; tab[idx_min] = tmp; } } </pre> |
| | GLI | C program derived from GLI |

| Problems | Results | |
|----------|--|---|
| | <pre> {"SP1": { "type": 0, "var_name": "tab", "lower_bound": "0", "upper_bound": "N-1", "size": "N", "iterator": ["p"], "annotations": { "precondition": [{"name": "IsArraySize"}], "postcondition_conserved": [{"name": "IsPermutationOf"}], "postcondition_done": [{"name": "IsSortedUpto"}], "invariant_iterator": [{"name": "IsWithinBounds"}], "invariant_conserved": [{"name": "IsPermutationOf"}], "invariant_done": [{"name": "IsSortedUpto"}] } }, {"SP2": { "type": 0, "var_name": "tab", "lower_bound": "p", "upper_bound": "N-1", "size": "N", "iterator": ["i"], "annotations": { "precondition": [], "postcondition_conserved": [{"name": "IsIndex"}], "postcondition_done": [], "invariant_iterator": [{"name": "IsWithinBounds"}], "invariant_conserved": [{"name": "IsIndex"}], "invariant_done": [{"name": "IsMinPosUpTo"}] } } } </pre> | <pre> method selectionSort(tab: array<int>, N: nat) { var i: int := 0; while i < (N - 1) { var idx_min: int := i; var j: int := i; while j < N { if (tab[j] < tab[idx_min]) { idx_min := j; } j := j + 1; } var tmp: int := tab[i]; tab[i] := tab[idx_min]; tab[idx_min] := tmp; i := i + 1; } } </pre> |
| | Formalized GLI overview (JSON) | Automatically translated Dafny code |
| | Output: Dafny verifier finished with 2 verified, 0 errors. | |
| | Remarks: The Dafny <i>Annotation</i> modifies [array] must be added for challenges modifying arrays. | |
| 7 | Challenge: Combine two integer arrays, inserting one of the two arrays at a given position. | |
| | | <pre> void insert(int a[], unsigned cap, unsigned L, unsigned pos, int b[], unsigned bLen) { unsigned j = 0; while (j < bLen) { a[pos + j] = b[j]; j++; } } </pre> |
| | GLI | C program derived from GLI |

| Problems | Results | |
|----------|---|--|
| | <pre> {"SP1": { "type": 0, "var_name": "b", "lower_bound": "0", "upper_bound": "bLen-1", "size": "bLen", "iterator": ["j"], "annotations": { "precondition": [{"name": "IsArraySize"}], "postcondition_conservd": [{"name": "IsUnchangedExceptBetween"}], "postcondition_done": [{"name": "IsInsertedInUpTo"}], "invariant_iterator": [{"name": "IsWithinBounds"}], "invariant_conservd": [{"name": "IsUnchangedExceptBetween"}], "invariant_done": [{"name": "IsInsertedInUpTo"}] } } </pre> | <pre> method insert(a: array<int>, cap: nat, L: nat, pos: nat, b: array<int>, bLen: nat) { var j: nat := 0; while (j < bLen) { a[pos + j] := b[j]; j := j + 1; } } </pre> |
| | Formalized GLI overview (JSON) | Automatically translated Dafny code |
| | Output: Dafny verifier finished with 4 verified, 0 errors. | |
| | Remarks: The Dafny <i>Annotation</i> modifies [array] must be added for challenges modifying arrays. The function insert requires that the array has been shifted. We do not show the shift function, but we verify it. That is why there is 4 verified instead of 2. | |
| 8 | Challenge: Determine the inverse of a natural number (i.e., reverse the digits of the number). | |
| |  | <pre> unsigned reverse(unsigned n, unsigned L) { unsigned inv = 0; unsigned i = 0; while (i < L) { unsigned d = n / (int)pow(10, i) % 10; inv = inv * 10 + d; i = i + 1; } return inv; } </pre> |
| | GLI | C program derived from GLI |

| Problems | Results | |
|----------|--|--|
| | <pre> {"SP1": { "type": 1, "var_name": "n", "lower_bound": "0", "upper_bound": "L-1", "size": "L", "iterator": ["i"], "annotations": { "precondition": [{"name": "IsNumberLength"}], "postcondition_conserved": [], "postcondition_done": [{"name": "IsReversedUpTo"}], "invariant_iterator": [{"name": "IsWithinBounds"}], "invariant_conserved": [], "invariant_done": [{"name": "IsReversedUpTo"}]} } } </pre> | <pre> method reverse(n: nat, L: nat) returns (automatic_return_variable_name: nat) { var inv: nat := 0; var i: nat := 0; while (i < L) { var d: nat := (n / (pow(10, i) as int)) % 10; inv := (inv * 10) + d; i := i + 1; } automatic_return_variable_name := inv; return; } </pre> |
| | Formalized GLI overview (JSON) | Automatically translated Dafny code |
| | Output: Dafny verifier finished with 2 verified, 0 errors. | |
| | Remarks: none. | |
| 9 | Challenge: Count the number of odd digits/occurrences of a digit in a natural number. | |
| |  | <pre> unsigned nb_odd(unsigned n, unsigned len) { unsigned nb = 0; unsigned i = 0; while (i < len) { unsigned d = (n / (int)pow(10, i)) % 10; if (d % 2 != 0) { nb++; } i++; } return nb; } </pre> |
| | GLI | C program derived from GLI |
| | <pre> {"SP1": { "type": 1, "var_name": "nb", "lower_bound": "0", "upper_bound": "len-1", "size": "len", "iterator": ["i"], "annotations": { "precondition": [{"name": "IsNumberLength"}], "postcondition_conserved": [], "postcondition_done": [{"name": "IsAmountOddUpTo"}], "invariant_iterator": [{"name": "IsWithinBounds"}], "invariant_conserved": [], "invariant_done": [{"name": "IsAmountOddUpTo"}]} } } </pre> | <pre> method nb_odd(n: nat, len: nat) returns (automatic_return_variable_name: nat) { var nb: nat := 0; var i: nat := 0; while (i < len) { var d: nat := (n / (pow(10, i) as int)) % 10; if ((d % 2) != 0) { nb := nb + 1; } i := i + 1; } automatic_return_variable_name := nb; return; } </pre> |
| | Formalized GLI overview (JSON) | Automatically translated Dafny code |

| Problems | Results | |
|----------|--|---|
| | Output: Dafny verifier finished with 2 verified, 0 errors. | |
| | Remarks: none. | |
| 10 | Challenge: Draw a triangle in the command terminal using characters. | |
| |  | <pre> char *DrawTriangle(int n, char symbol) { int totalChars = 2 * n * n; char drawing[totalChars + 1]; int i = 0; while (i < n) { int j = 0; while (j < 2 * n - 1) { if (j < (n - i - 1) j >= (n + i)) { drawing[2 * n * i + j] = ' '; } else { drawing[2 * n * i + j] = symbol; } j++; } drawing[2 * n * i + j] = '\n'; i++; } drawing[2 * n * i] = '\0'; return drawing; } </pre> |
| | GLI | C program derived from GLI |
| | <pre> {"SP1": { "type": 5, "var_name": "triangle", "lower_bound": "0", "upper_bound": "L-1", "size": "L", "iterator": ["i"], "annotations": { "precondition": [], "postcondition_conserved": [], "postcondition_done": [{"name": " IsTriangleLineUpTo"}], "invariant_iterator": [{"name": " IsWithinBounds"}], "invariant_conserved": [], "invariant_done": [{"name": " IsTriangleLineUpTo"}] } }, "SP2": { "type": 3, "var_name": "ligne_i", "lower_bound": "0", "upper_bound": "C-1", "size": "C", "iterator": ["j"], "annotations": { "precondition": [], "postcondition_conserved": [], "postcondition_done": [], "invariant_iterator": [{"name": " IsWithinBounds"}], "invariant_conserved": [], "invariant_done": [{"name": " IsTriangleLineUpTo"}] } } } </pre> | <pre> method DrawTriangle(n: int, symbol: char) returns (automatic_return_variable_name: array<char>) { var totalChars: int := (2 * n) * n; var drawing: array<char> := new char[totalChars + 1]; var i: int := 0; while (i < n) { var j: int := 0; while (j < ((2 * n) - 1)) { if ((j < ((n - i) - 1)) (j >= (n + i))) { drawing[((2 * n) * i) + j] := ' '; } else { drawing[((2 * n) * i) + j] := symbol; } j := j + 1; } drawing[((2 * n) * i) + j] := '\n'; i := i + 1; } drawing[(2 * n) * i] := '\0'; automatic_return_variable_name := drawing; return; } </pre> |

| Problems | Results | |
|----------|--|-------------------------------------|
| | Formalized GLI overview (JSON) | Automatically translated Dafny code |
| | Output: Dafny verifier finished with 2 verified, 0 errors. | |
| | Remarks: none. | |

Table 5.1: Compilation of results for Problems 1-4 and 6-10

5.3 Limitations

The process we have developed has certain limitations. However, those do not prevent us from achieving the initial objective of this research, namely, to demonstrate the feasibility of automatic verification using GLI in the context of the CS1 introductory programming course [9]. They do not prevent the automatic verification of GLIs and programs using GLIs. However, if further development were to be considered, it would be necessary to reflect on and overcome these various limitations to obtain a result applicable in a broader context (i.e., beyond the subject matter covered in a CS1 course).

5.3.1 Single or Two Nested Loop Program

We have always worked with single or double nested loops in this work. We have not had the opportunity to test programs with three or more levels of loops. At first glance, simple programs could probably fit the standard scheme mentioned in Section 4.4.1, but that would require additional and more thorough testing to verify.

5.3.2 Strict Compliance with GLI

Another limitation is the strict compliance that the programmer must have with the GLI. In order for a GLI to verify a program, one must construct the program in the same way as the GLI from which it originates. Students are, in any case, expected to use invariants constructively (i.e., to use them to construct the program). However, deviating from how the GLI works when writing code is relatively easy. In that case, Dafny may have difficulty proving that the program is correct. That is because Dafny is a language that uses basic logic and mathematics to demonstrate the correctness of programs. Also, some properties that may seem obvious to a human being (e.g., the commutativity and distributivity of multiplication) will require much time and calculations to validate. It is therefore necessary to remain vigilant and not deviate from the functioning of the GLI when deriving the C code.

5.3.3 Extraction of Variable Names

Our process does not fully handle the extraction of variables described in sentences contained in the semantic zones (i.e., *Done*, *To Do*, and *Conserved* zones) of the GLI. We retrieved the labels of these variables by highlighting them (i.e., placing them in curly brackets). Unfortunately, the model we use for extracting the semantics of sentences is not powerful enough to retrieve the meaning of these variables from the context of the sentences. We ran a few tests with ChatGPT and its most advanced model, ChatGPT-4o-mini-high, and the results were much better. Figure 5.3 shows what we received from simple sentences without context outside the question prompt.

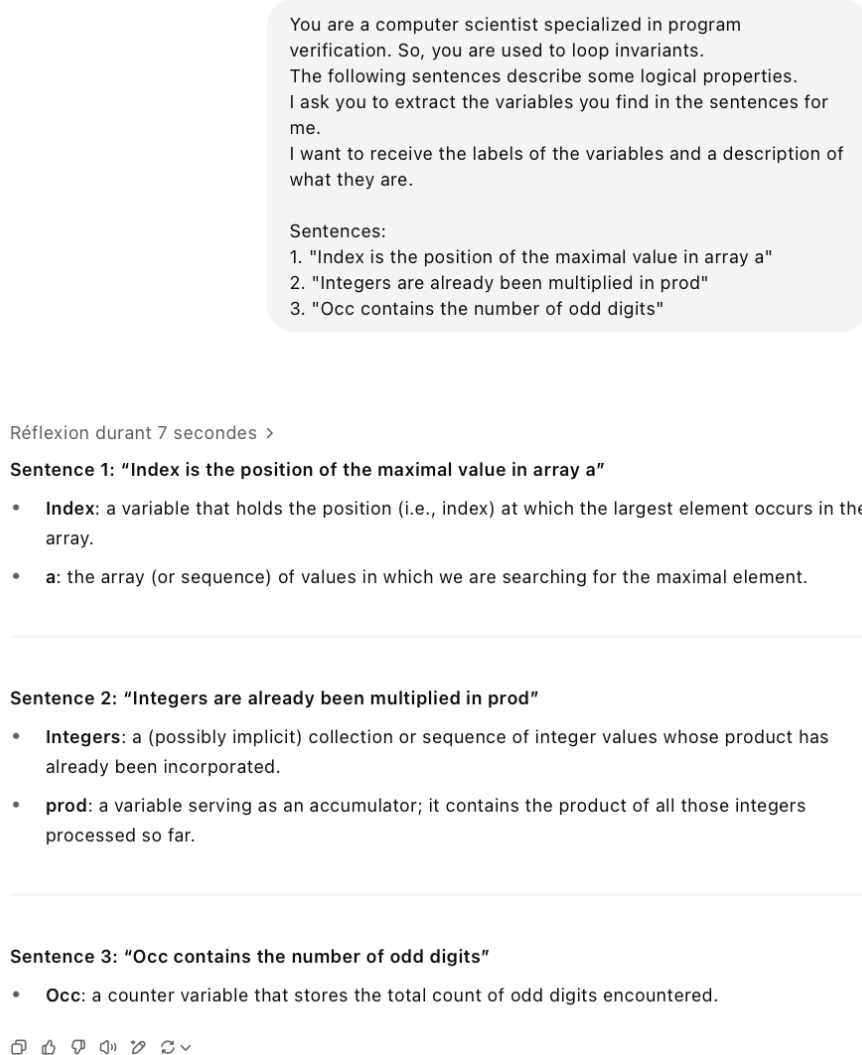


Figure 5.3: Prompt to extract variables from sentences

5.3.4 Integration Steps

One limitation is the lack of automation in the final step of the process: inserting predicates into the Dafny code. We have mitigated that by attempting to simplify the

integration of predicates into the code as much as possible (e.g., insertion patterns, standardization of predicates, etc.). However, two obstacles have prevented the implementation of this final step. The first is that our model has difficulty extracting variables from the semantic areas of the GLI. The second is the use of `pycparser`. We modified the code generator to generate code following the Dafny syntax, but did not change the parser that constructs the AST from the C code. Therefore, the conditions were not optimal for inserting elements with no equivalent in C. The solution proposed by Sriya S et al. in their article “*Verification of C Programs using Annotations*” [23] is to use comments. Unfortunately, `pycparser` cannot handle comments in C. However, if we rely on the results obtained by Sriya S et al., then we can say that it is possible to insert Dafny *Annotations* into code to verify it.

5.4 Code Files

We have not included all the code files in this document for length reasons. However, they are all available on the GitLab page: <https://gitlab.uliege.be/A.Grosjean/tfe>. The `README.md` file provides a quick overview of the various files in the repository. Here is a non-exhaustive list of the main ones.

- `c_prog` (Files: 10, Lines: ± 200)
It contains all the C codes derived from GLI.
- `dafny_prog/benchmarks` (Files: 14, Lines: ± 800)
It contains all the Dafny files. These are not the translated files but the original files with which we worked to develop our methodology and the process.
- `GLI_json` (Files: 10, Lines: ± 500)
It contains all the initial JSON files of the GLIs (i.e., files in the format used by CAFÉ2.0 to store the GLIs)
- `json_parser` (Files: 14, Lines: ± 2200)
It contains the programs used to parse JSON files and generate the formalized GLIs. These formalized GLIs are stored in `json_parser/formalized_GLI` under JSON format.
- `translated_prog` (Files: 10, Lines: ± 300)
It contains the ten Dafny translations of the C programs.
- `translator` (Files: 1, Lines: ± 600)
It contains the `DafnyGenerator` used to translate the `.c` automatically in `.dfy`

Chapter 6

Conclusion

Based on our results, we can say that the research's initial objectives have been met.

We began by detailing the theoretical concepts we used during our research. Based on those, we considered a process that would allow us to verify computer programs using the GLI tool formally. The solution we developed is based on the Dafny programming language, enabling us to verify ten programming challenges. To achieve this result, we divided the process into three significant steps.

The first was translating the program to be verified from C to Dafny. The research work of Sriya et al. [23] showed us that it was feasible in our context (i.e., introduction to C programming). Besides, we were also able to implement a functional translator.

The second step involved parsing the GLI, which involved extracting the information needed to verify the program. In addition to this extraction, we had to transform natural language propositions (e.g., French) into formal propositions using logic and mathematics. The semantic comparison of descriptions made that possible. At the end of this second step, we obtained a formalized version of the GLIs.

Finally, the last part involved inserting the formal elements of the GLI into the program, translated into Dafny. Although this step could not be fully automated, we demonstrated its feasibility by describing a standard schema explaining where to position the logical predicates derived from the GLI.

Therefore, the initial objective of demonstrating that GLIs could be used to prove the correctness of C programs in the Introduction to Computer Programming course has been achieved. The fact that we were able to implement the process steps in a concrete and schematic way is a sign that future work on the subject could lead to a concrete tool that teachers and students in introductory C programming courses can use.

From a strictly personal point of view, in addition to being constructive for further research in the field, carrying out this work has made me aware of the importance of pedagogy in introducing students to the world of computer science. In my opinion, too many students still drop out of this field because they do not understand its concepts, such

as using loops. I believe that efforts to support better students' learning, such as studies on GLI, stand to benefit the entire discipline of computer science.

6.1 Future Work

Examining the limitations would be a good starting point for future work that could extend the results of this research.

In order to provide a truly functional tool, it will be necessary to define an efficient and robust translator from C to Dafny. Such a tool will also require looking into the use of better models for semantic extraction. We did not give many details about such models in our section on the limitations of our method. Still, large language models (LLMs) are a serious solution to consider in order to break down the barrier between words (i.e., natural language) and formal principles of logic and mathematics. Finally, further research will be needed to include as many types of programs as possible within the scope of the system. The aim is to cover enough notions so that students have sufficient knowledge to grasp, on their own, the fundamental concepts hidden beneath the layer of abstraction that is GLIs.

Appendix A

Basic C Functionalities

This appendix contains an extract from the site Quickref [6] containing a summary of the syntax of the C language. The website can be found here: <https://quickref.me/c.html>.

Getting Started

hello.c

```
#include <stdio.h>

int main(void) {
    printf("Hello World!\n");

    return 0;
}
```

Compile `hello.c` file with `gcc`

```
$ gcc hello.c -o hello
```

Run the compiled binary `hello`

```
$ ./hello
```

Output => Hello World!

Variables

```
int myNum = 15;

int myNum2; // do not assign, then
myNum2 = 15;

int myNum3 = 15; // myNum3 is 15
myNum3 = 10;    // myNum3 is now

float myFloat = 5.99; // floating
char myLetter = 'D'; // character

int x = 5;
int y = 6;
int sum = x + y; // add variables

// declare multiple variables
int x = 5, y = 6, z = 50;
```

Constants

```
const int minutesPerHour = 60;
const float PI = 3.14;
```

Best Practices

```
const int BIRTHYEAR = 1980;
```

Comment

```
// this is a comment
printf("Hello World!"); // Can con

/*Multi-line comment, print Hello
to the screen, it's awesome */
```

Switch

```
int day = 4;

switch (day) {
    case 3: printf("Wednesday"); break;
    case 4: printf("Thursday"); break;
    default:
        printf("Weekend!");
}

// output -> "Thursday" (day 4) 🐼
```

While Loop

```
int i = 0;

while (i < 5) {
    printf("%d\n", i);
    i++;
}

NOTE: Don't forget to increment the
variable used in the condition, otherwise
the loop will never end and become an
"infinite loop"!
```

Condition

```
int time = 20;
if (time < 18) {
    printf("Goodbye!");
} else {
    printf("Good evening!");
}

// Output -> "Good evening!"

int time = 22;
if (time < 10) {
    printf("Good morning!");
} else if (time < 20) {
    printf("Goodbye!");
} else {
    printf("Good evening!");
}

// Output -> "Good evening!"
```

For Loop

```
for (int i = 0; i < 5; i++) {
    printf("%d\n", i);
}
```

Break out of the loop Break/Continue

```
for (int i = 0; i < 10; i++) {
    if (i == 4) {
        break;
    }
    printf("%d\n", i);
}
```

break out of the loop when `i` is equal to 4

```
for (int i = 0; i < 10; i++) {
    if (i == 4) {
        continue;
    }
    printf("%d\n", i);
}
```

Example to skip the value of 4

While continue example

```
int i = 0;

while (i < 10) {
    i++;

    if (i == 4) {
        continue;
    }
    printf("%d\n", i);
}
```

Do/While Loop

```
int i = 0;

do {
    printf("%d\n", i);
    i++;
} while (i < 5);
```

While Break Example

```
int i = 0;

while (i < 10) {
    if (i == 4) {
        break;
    }
    printf("%d\n", i);

    i++;
}
```

Strings

```
char greetings[] = "Hello World!";
printf("%s", greetings);
```

access string

```
char greetings[] = "Hello World!";
printf("%c", greetings[0]);
```

modify string

```
char greetings[] = "Hello World!";
greetings[0] = 'J';

printf("%s", greetings);
// prints "Jello World!"
```

Another way to create a string

```
char greetings[] = {'H','e','l','l','o'};

printf("%s", greetings);
// print "Hello!"
```

Creating String using character pointer (String Literals)

```
char *greetings = "Hello";
printf("%s", greetings);
// print "Hello!"
```

NOTE: String literals might be stored in read-only section of memory. Modifying a string literal invokes undefined behavior. You can't modify it!

C does not have a String type, use **char** type and create an **array** of characters

Arrays

```
int myNumbers[] = {25, 50, 75, 100};

printf("%d", myNumbers[0]);
// output 25
```

change array elements

```
int myNumbers[] = {25, 50, 75, 100};
myNumbers[0] = 33;

printf("%d", myNumbers[0]);
```

Loop through the array

```
int myNumbers[] = {25, 50, 75, 100};
int i;

for (i = 0; i < 4; i++) {
    printf("%d\n", myNumbers[i]);
}
```

set array size

```
// Declare an array of four integers
int myNumbers[4];
```

```
// add element
myNumbers[0] = 25;
myNumbers[1] = 50;
myNumbers[2] = 75;
myNumbers[3] = 100;
```

Operators

Arithmetic Operators

```
int myNum = 100 + 50;
int sum1 = 100 + 50; // 150 (100 + 50)
int sum2 = sum1 + 250; // 400 (150 + 250)
int sum3 = sum2 + sum2; // 800 (400 + 400)
```

| | | |
|----|-----------|----------|
| + | Add | $x + y$ |
| - | Subtract | $x - y$ |
| * | Multiply | $x * y$ |
| / | Divide | x / y |
| % | Modulo | $x \% y$ |
| ++ | Increment | $++x$ |
| -- | Decrement | $--x$ |

Assignment operator

| | |
|----------------|------------------|
| $x = 5$ | $x = 5$ |
| $x += 3$ | $x = x + 3$ |
| $x -= 3$ | $x = x - 3$ |
| $x *= 3$ | $x = x * 3$ |
| $x /= 3$ | $x = x / 3$ |
| $x \% = 3$ | $x = x \% 3$ |
| $x \&= 3$ | $x = x \& 3$ |
| $x = 3$ | $x = x 3$ |
| $x \wedge = 3$ | $x = x \wedge 3$ |
| $x >>= 3$ | $x = x >> 3$ |
| $x <<= 3$ | $x = x << 3$ |

Comparison Operators

```
int x = 5;
int y = 3;

printf("%d", x > y);
// returns 1 (true) because 5 is greater than 3
```

| | | |
|----|--------------------------|----------|
| == | equals | $x == y$ |
| != | not equal to | $x != y$ |
| > | greater than | $x > y$ |
| < | less than | $x < y$ |
| >= | greater than or equal to | $x >= y$ |
| <= | less than or equal to | $x <= y$ |

Comparison operators are used to compare two values

| Logical Operators | | | |
|-------------------|-------------|---|---|
| && | and logical | returns true if both statements are true | <code>x < 5 && x < 10</code> |
| | or logical | returns true if one of the statements is true | <code>x < 5 x < 4</code> |
| ! | not logical | Invert result, return false if true | <code>!(x < 5 && x < 10)</code> |

| Bitwise operators | | | |
|-------------------|---|--|--|
| & | Bitwise AND operation, "AND" operation by binary digits | (A & B) will get 12 which is 0000 1100 | |
| | Bitwise OR operator, "or" operation by binary digit | (A B) will get 61 which is 0011 1101 | |
| ^ | XOR operator, perform "XOR" operation by binary digits | (A ^ B) will get 49 which is 0011 0001 | |
| ~ | Inversion operator, perform "inversion" operation by binary bit | (~A) will get -61 which is 1100 0011 | |
| << | binary left shift operator | A << 2 will get 240 which is 1111 0000 | |
| >> | binary right shift operator | A >> 2 will get 15 which is 0000 1111 | |

| Operator Examples | |
|--|--|
| <pre> unsigned int a = 60; /*60 = 0011 1 unsigned int b = 13; /*13 = 0000 1 int c = 0; c = a & b; /*12 = 0000 1100 */ printf("Line 1 -the value of c is c = a b; /*61 = 0011 1101 */ printf("Line 2 -the value of c is c = a ^ b; /*49 = 0011 0001 */ printf("Line 3 -the value of c is c = ~a; /*-61 = 1100 0011 */ printf("Line 4 -The value of c is c = a << 2; /*240 = 1111 0000 */ printf("Line 5 -the value of c is c = a >> 2; /*15 = 0000 1111 */ printf("Line 6 -The value of c is </pre> | |

Data Types

| Basic data types | | | |
|--------------------|--------------|--------------------------------|-------------------------------------|
| char | 1 byte | -128 ~ 127 | single character/alphanumeric/ASCII |
| signed char | 1 byte | -128 ~ 127 | - |
| unsigned char | 1 byte | 0 ~ 255 | - |
| int | 2 to 4 bytes | -32,768 ~ 32,767 | store integers |
| signed int | 2 bytes | -32,768 ~ 32,767 | |
| unsigned int | 2 bytes | 0 ~ 65,535 | |
| short int | 2 bytes | -32,768 ~ 32,767 | |
| signed short int | 2 bytes | -32,768 ~ 32,767 | |
| unsigned short int | 2 bytes | 0 ~ 65,535 | |
| long int | 4 bytes | -2,147,483,648 ~ 2,147,483,647 | |
| signed long int | 4 bytes | -2,147,483,648 ~ 2,147,483,647 | |
| unsigned long int | 4 bytes | 0 ~ 4,294,967,295 | |
| float | 4 bytes | 3.4E-38 ~ 3.4E+38 | |
| double | 8 bytes | 1.7E-308 ~ 1.7E+308 | |
| long double | 10 bytes | 3.4E-4932 ~ 1.1E+4932 | |

| Data types | |
|---|--------------------------------------|
| <pre> // create variables int myNum = 5; // integer float myFloatNum = 5.99; // float char myLetter = 'D'; // string // High precision floating point c double myDouble = 3.2325467; // print output variables printf("%d\n", myNum); printf("%f\n", myFloatNum); printf("%c\n", myLetter); printf("%lf\n", myDouble); </pre> | |
| char | character type |
| short | short integer |
| int | integer type |
| long | long integer |
| float | single-precision floating-point type |
| double | double-precision floating-point type |
| void | no type |

Bibliography

- [1] Mike Barnett et al. “Boogie: A Modular Reusable Verifier for Object-Oriented Programs”. In: *Lecture Notes in Computer Science* 4111 (Jan. 2006). ISSN: 1611-3349.
- [2] Eli Bendersky. *pycparser: Complete C99 Parser in Pure Python*. GitHub. Release v2.22, March 30, 2024. Accessed: 2025-05-25. Mar. 2024. URL: <https://github.com/eliben/pycparser>.
- [3] B. Boigelot. *Introduction à l’informatique*. Accessed: 2025-04-21. 2025. URL: <https://people.montefiore.uliege.be/boigelot/cours/info/slides/info-slides-2024.pdf>.
- [4] Géraldine Brieven, Lev Malcev, and Benoit Donnet. “Practicing Abstraction Skills Through Diagrammatic Reasoning Over CAFÉ 2.0”. In: *2024 IEEE Global Engineering Education Conference (EDUCON)*. 2024, pp. 1–10. DOI: 10.1109/EDUCON60312.2024.10578665.
- [5] Géraldine Brieven et al. “Graphical Loop Invariant Based Programming”. In: *Formal Methods Teaching*. Ed. by Catherine Dubois and Pierluigi San Pietro. Springer Nature Switzerland, 2023, pp. 17–33. ISBN: 978-3-031-27534-0.
- [6] *C Cheat Sheet & Quick Reference*. Accessed: 2025-05-26. Quick Ref .ME. URL: <https://quickref.me/c.html>.
- [7] Yu. A. Cherenkova, Daniel Zingaro, and Andrew Petersen. “Identifying Challenging CS1 Concepts in a Large Problem Dataset”. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. 2014. URL: <https://api.semanticscholar.org/CorpusID:16850286>.
- [8] Edsger W. Dijkstra. *A Discipline of Programming*. Englewood Cliffs, NJ, USA: Prentice-Hall, Inc., 1976.
- [9] Benoit Donnet. *INFO0946-1 Introduction to Computer Programming*. Accessed: 2025-04-10. 2025. URL: <https://www.programmes.uliege.be/cocoon/20242025/en/cours/INFO0946-1.html>.
- [10] Robert W. Floyd. “Assigning Meanings to Programs”. In: *Mathematical Aspects of Computer Science*. Ed. by J. T. Schwartz. Vol. 19. Proceedings of Symposia in Applied Mathematics. Providence, RI: American Mathematical Society, 1967, pp. 19–32.

- [11] Carlo A. Furia, Bertrand Meyer, and Sergey Velder. “Loop Invariants: Analysis, Classification, and Examples”. In: *ACM Computing Surveys* 46.3 (Jan. 2014), 34:1–34:51. DOI: 10.1145/2506375. URL: <https://doi.org/10.1145/2506375>.
- [12] Luke Herbert, K. Rustan M. Leino, and Jose Quaresma. “Using Dafny, an Automatic Program Verifier”. In: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Ed. by Bertrand Meyer and Martin Nordio. Springer Berlin Heidelberg, 2012, pp. 156–181. ISBN: 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_6. URL: https://doi.org/10.1007/978-3-642-35746-6_6.
- [13] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Communications of the ACM* 12.10 (Oct. 1969), pp. 576–580. DOI: 10.1145/363235.363259. URL: <https://doi.org/10.1145/363235.363259>.
- [14] Jason Koenig and Rustan M. Leino. *Getting Started with Dafny: A Guide*. May 2012. URL: <https://www.microsoft.com/en-us/research/publication/getting-started-dafny-guide/>.
- [15] Lajavaness. *Lajavaness*. Accessed: 2025-06-01. URL: <https://www.lajavaness.com>.
- [16] Lajavaness. *Lajavaness/sentence-camembert-large*. Accessed: 2025-06-01. URL: <https://huggingface.co/Lajavaness/sentence-camembert-large>.
- [17] K. Rustan M. Leino. “Developing Verified Programs with Dafny”. In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 1488–1490. DOI: 10.1109/ICSE.2013.6606754. URL: <https://homepage.cs.uiowa.edu/~tinelli/classes/181/Fall15/Papers/Lein13.pdf>.
- [18] Simon Liénardy, Lev Malcev, and Benoit Donnet. “Graphical Loop Invariant Programming in CS1”. In: *Grascomp Doctoral Day 2019 (GDD’19)*. Namur, Belgium, 2019. URL: <https://orbi.uliege.be/bitstream/2268/241671/2/paper.pdf>.
- [19] Marqo AI. *Introduction to Sentence Transformers*. Accessed: 2025-06-01. URL: <https://www.marqo.ai/course/introduction-to-sentence-transformers>.
- [20] Mikael Mayer. *Making Verification Compelling: Visual Verification Feedback for Dafny*. Accessed: 2025-05-26. Apr. 2023. URL: <https://dafny.org/blog/2023/04/19/making-verification-compelling-visual-verification-feedback-for-dafny/#step-0-0>.
- [21] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

- [22] Nils Reimers and Iryna Gurevych. “Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Accessed: 2025-05-26. Association for Computational Linguistics, Nov. 2019. URL: <https://arxiv.org/abs/1908.10084>.
- [23] Sriya S et al. “Verification of C Programs Using Annotations”. In: *2019 IEEE Tenth International Conference on Technology for Education (T4E)*. 2019, pp. 106–109. DOI: 10.1109/T4E.2019.00-41.
- [24] The Dafny Team. *The Dafny Reference Manual*. Accessed: 2025-04-24. 2023. URL: <http://dafny.org/dafny/DafnyRef/DafnyRef>.
- [25] Wikipedia contributors. *Abstract Syntax Tree*. Accessed: 2025-05-26. Mar. 2025. URL: https://en.wikipedia.org/wiki/Abstract_syntax_tree.
- [26] Wikipedia contributors. *Loop Invariant*. Accessed: 2025-04-21. Apr. 2025. URL: https://en.wikipedia.org/wiki/Loop_invariant.
- [27] Sonnex Will and Sophia Drossopoulou. *Verified Programming in Dafny*. URL: https://www.doc.ic.ac.uk/~scd/Dafny_Material/Lectures.pdf.