Graphical Loop Invariant Based Programming

Géraldine Brieven[1 [0000−0003−1410−1470]], Simon Liénardy[1 [0000−0001−9081−4726]], Lev Malcev[1 [0000−0002−5259−4336]], and BenoitDonnet[1 [2222−−3333−4444−5555]]

Université de Liège, Montefiore Institute, Liège, Belgium

Abstract. This paper focuses on a programming methodology relying on an informal and graphical version of the Loop Invariant for building the code. This methodology is applied in the context of a CS1 course in which students are exposed to several C programming language concepts and algorithmic aspects. The key point in the course is thus to imagine a problem resolution strategy (the Graphical Loop Invariant) prior to writing the code (that becomes, then, reasonably easy once relying on the Graphical Loop Invariant). This paper exposes the rules for building a sound and accurate Graphical Loop Invariant as well as the programming methodology. As such, our programming methodology might be seen as a first step towards considering formal methods in programming courses without making any assumption on students mathematical background as it does not require to manipulate any mathematical notations. The paper also introduces an integrated learning tool we developed for supporting the Graphical Loop Invariant teaching and practice. Finally, the paper gives preliminary insight into how students seize the methodology and use the learning tools for supporting their learning phase.

Keywords: Loop Invariant · Graphical Loop Invariant · Graphical Loop Invariant Based Programming · Glide · Café

1 Introduction

This paper proposes and discusses a graphical methodology, based on the Loop Invariant [13,17], to help students in efficiently and strictly programming loops. This methodology is applied in the context of an Introduction to Programming (i.e., CS1) course alternating between specific C programming language concepts and algorithmic aspects. In particular, the course aims at introducing to first year students basic principles of programming. The concept of a correct and efficient algorithm is highlighted, in the context of a strict programming methodology. Typically, an algorithm requires to write a sequence of instructions that must be repeated a certain number of times. This is usually known as a *program loop*. The methodology we teach for programming a loop is based on an informal version of the *Loop Invariant* (a program loop property verified at each iteration – i.e., at each evaluation of the Loop Condition) introduced by Floyd and Hoare [13,17]. Our methodology consists in determining a strategy (based on the Loop Invariant) to solve a problem prior to any code writing and, next, rely on the strategy to build the code, as initially proposed by Dijkstra [11].

As such, the Loop Invariant can be seen as the cornerstone of code writing. However, the issue is that it relies on an abstract reflection that might confuse students who may not have the desired abstract background, specially if the Loop Invariant is expressed as a logical assertion. This is the reason why, according to Astrachan [1], Loop Invariants are usually avoided in introductory courses.

That statement is consolidated by much research [20,22] showing that teaching CS1 is known to be a difficult task since, often, students taking a CS1 class encounter difficulties in understanding how a program works [27], in designing an efficient and elegant program [10] (conditionals and loops have proven to be particularly problematic [8]), in problem solving and mathematical ability [22], and in checking whether a program works correctly [5]. Morever, in our context, due to the large variety of students entering the CS1 program in Belgium[1], we cannot make any assumptions about a first year student's background.

To ensure students follow a strict programming methodology despite their (potential) gaps, we propose a *Graphical Loop Invariant* (Gli). The Gli informally describes, at least, variables, constant(s), and data structures handled by the program; their properties; the relationships they may share, and that are preserved over all the iterations. The goal behind is to generalize what the program must have performed after each iteration. In addition to natural advantages of drawings [15,25,26], the Gli allows the programmer to visually deduce instructions before, during, and after the loop. That approach forges abstraction skills without relying on any mathematical background and lays the foundations for more formal methods where the Gli stands as an intermediate step towards a final formal Loop Invariant (being a logical assertion). Our programming methodology is supported by an integrated tool called Café.

The remainder of this paper is organized as follows: Sec. 2 presents the Gli and how to construct it; Sec. 3 discusses the programming methodology based on the Gli; Sec. 4 introduces the integrated tool for supporting the Gli teaching and practice; Sec. 5 presents preliminary results on how students seize the Gli; Sec. 6 positions this paper with respect to the state of the art; finally, Sec. 7 concludes this paper by summarizing its main achievements and by discussing potential directions for further researches.

2 Graphical Loop Invariant

2.1 Overview

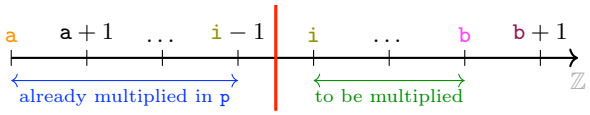
A *Loop Invariant* [13,17] is a property of a program loop that is verified (i.e., True) at each iteration (i.e., at each evaluation of the Loop Condition). The Loop Invariant purpose is to express, in a generic and formal way through a logical assertion, what has been calculated up to now by the loop. Historically, the Loop Invariant has been used for proving code correctness (see, e.g., Cormen et al. [9] and Bradley et al. [6] for automatic code verification). As such, the Loop Invariant is used "a posteriori" (i.e., after code writing).

[1] in which open access to Higher Education is the general rule, with some exceptions.

On the contrary, Dijkstra [11] proposed to first determine the Loop Invariant and then use it to deduce the code instructions. The Loop Invariant is therefore used "a priori". Our methodology differs from Dijkstra's as we propose to represent the Loop Invariant as a picture: the *Graphical Loop Invariant* (Gli). This picture must depict the variables, constants, and data structures that will appear in the code, as well as the constraints on them; the relationships they may share, and that are conserved all over the iterations. To illustrate the Gli, a very simple problem is taken as example throughout the paper :

Input : Two integers `a` and `b` such as $\texttt{a} < \texttt{b}$
Output : the product of all the integers in $[\texttt{a}, \texttt{b}]$

Fig. 1a shows how the problem should be solved through the corresponding Gli. We first represent the integers between the boundaries of the problem (`a` and `b`) thanks to a graduated line labelled with the integers symbol ($\mathbb{Z}$). It models the iteration over all the integers from `a` to `b`. Then, to reflect the situation after a certain number of iterations, a vertical red bar (called a *Dividing Line*) is drawn to divide the integer line into two areas. The left area, in blue, represents the integers that were already multiplied in a variable `p` (`p` is thus the accumulator storing intermediate results over the iterations). The right area, in green, covers the integers that still have to be multiplied. We decide to label the nearest integer at the right of the Dividing Line with the variable `i` that plays the role of the iterator variable in the range $[\texttt{a}, \texttt{b}]$. Of course, the variables `i` and `p` must be used in the code. In the following, based on seven rules (see Fig. 2) and predefined drawing patterns (see Sec. 2.3), we provide a methodology for easing the building of a correct Gli (see Sec. 2.2). Then, Sec. 3 details how to deduce code instructions from a Gli manipulation, based on this example.



$$a \leq i \leq b+1 \wedge p = \prod_{j=a}^{i-1} j.$$

(a) Graphical Loop Invariant. (b) Formal Loop Invariant.

Fig. 1. Loop Invariant for an integer product between two boundaries.

It is worth noticing here that playing with a graphical version of the Loop Invariant allows students to learn applying formal method programming without manipulating mathematical notations. Indeed, Fig. 1b provides the formal Loop Invariant corresponding to the Gli depicted in Fig. 1a (with the same color code). Producing such a predicate may appear harder to students as it requires to use Mathematical notations (such as $\prod$) with free (i, a, and b) and bound (j) variables. Therefore, the Gli and its usage in code construction might be seen as a first step towards learning and using formal methods in programming.

2.2 Constructing a Graphical Loop Invariant

Finding a Loop Invariant to solve a problem may appear as a difficult task. There are multiple ways to discover a Loop Invariant: e.g., by induction, by

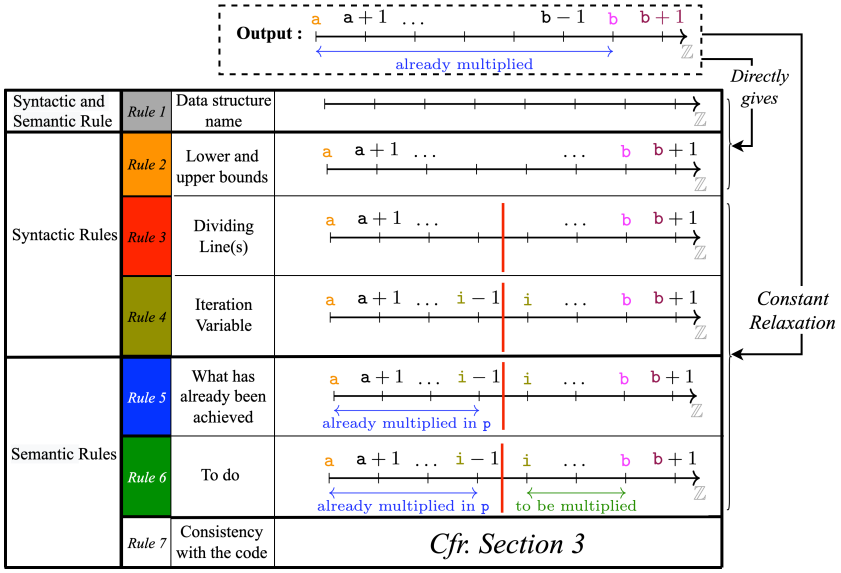


Fig. 2. Designing a GLI step-by-step, from the problem output.

working from the precondition, or by starting from the postcondition. For our course, we rather explain to students how to apply graphically the *constant relaxation* technique [14], i.e., 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. To help students across that abstract process, we provide seven rules they should apply when searching for a *sound* and *accurate* GLI, as illustrated through Fig. 2. Those rules are categorized into two main categories: (*i*) *syntax* (i.e., focusing on the drawing aspects only – Rule 1 → Rule 4), (*ii*) *semantic* (i.e., focusing on the explanations added to the drawings – Rule 1 and Rule 5 → Rule 7).

In particular, Fig. 2 shows that it first starts by drawing the program output thanks to a pre-defined pattern (Rule 1 – the different possible patterns are described in Sec. 2.3) and by explaining the program goal (blue arrow and text). This rule recommends to draw an accurate representation of the data or the data structures relevant for the given problem. Rule 1 also recommends to properly label each drawn data structure (e.g., with variable name). It is essential if several data structures are handled by the program, as they could be mistaken during the code writing. Then, boundaries must frame each structure (Rule 2). Applying Rule 2 prevents some common mistakes, when building the code (see Sec. 3) such as array out of bound errors or overflow. These errors would indeed be more unlikely if the the data structure length is properly mentioned in the GLI.

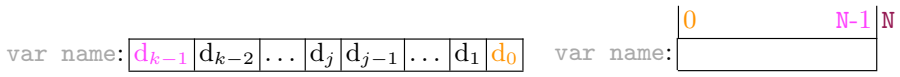
Next, Rules 3 → Rule 6 are sequentially applied in order to roll back the final perspective and visualize the solution under construction through each variable

state. Applying Rule 3 makes the Dividing Line appear, naturally reducing the blue zone length (Rule 5) and making room for the green one (Rule 6). The Dividing Lines are the core of our methodology. They symbolize the division between what was already computed by the program and what should still be done to reach the program objective. They enable to graphically manipulate the drawing in order to deduce the code instructions (see Sec. 3).

Applying Rule 4 requires to decide where to place the iteration variable around the Dividing Line, i.e., on the left (thus being part of what has already been achieved by previous iterations) or on the right (as part of the "to do" zone). Usually, we advice students to place it on the right, so that it references the current element to process in the Loop Body. Since a Dividing Line separates what has been done and what is going to be, that means that if we depicted such a line on the data representation as the program is executed, this line would move from one position to another: the first position would correspond to the initial state while the last position would correspond to the final one.

Currently, the application of Rule 5 is partial as it lacks a variable for accumulating intermediate results. It is thus enough to rephrase the sentence below the arrow by introducing the accumulator (i.e., variable `p`). Applying Rule 5 helps thinking about the behaviour of the program. In order to determine "what has been achieved so far", one should ask the question: "In order to reach the program goal, what should have been computed until now? Which variable properties must be ensured?" Most of the time, this reflection phase highlights either the need for additional variables that contain partial results or relationships between variables that must be conserved throughout the code execution. On the other hand, the information about what has been achieved so far is crucial during the code writing as it helps to decide what are the instructions to be performed during an iteration, i.e., to deduce the *Loop Body* (see Sec. 3).

Finally, it is enough to label the drawing with the "to do" zone right to the Dividing Line, following Rule 6. Naturally, the Gli obtained here is exactly the same as the one provided in Fig. 1a. Applying Rule 6 appears as the less important guideline as it does not bring additional information about the solution. In fact, if we expressed a Gli as a formal one (i.e., as a predicate), there would be no logical notation to describe "what should still be done". Nevertheless, drawing an area indicating what should still be done is a good way to ease the representation of the initial and final states of the program. In the initial state, this "to do area" should span over all the data that is concerned by the program. On the contrary, In the final state, this area should have disappeared while the only remaining area represents what has been achieved by the program. It is then easy to check if the purpose of the program is met in such a state. Moreover, when deducing the code instructions (see Sec. 3), this "to do area"helps to deduce the updates of the variables labelling the Dividing Lines, since the lines have to be moved in order to shrink the area. Finally, it helps finding a Loop Variant to show loop termination as the size of the "to do area" is often a good candidate for the Loop Variant ($b + 1 - a$ on the Gli illustrated in Fig. 1).



(a) Number pattern. (b) Array pattern.

Fig. 3. Pre-defined drawing patterns for Gli.

The last rule is a reminder to check if all the variables identified during that reflection phase are actually included in the code.

In addition, to help student identifying the various rules and their applications in a Gli, we adopt a color code (see Fig. 2). This color code is consistent throughout the course and the developed tools (see Sec. 4) and help students understanding exposed Gli during classes.

2.3 Graphical Loop Invariant Patterns

This section introduces some standard patterns for graphically representing common data structures in a CS1 course. Those patterns rely on the first two rules for a correct Gli with the associated color code (see Fig. 2).

Graduated Line. One of the most basic pattern is the graduated line, allowing to represent ordered sets like subsets of Natural or Integers. The line is labelled with the set name (e.g., $\mathbb{N}$ or $\mathbb{Z}$). Each tick on the line corresponds to a value and all those values are offset by the same step. The arrow at the far-right of the line indicates the increasing order of values. That pattern was supporting the Gli presented in the previous subsection and can be seen as resulting from from Rule 1 in Fig. 2. Moreover, that line should be framed by boundaries, as performed by applying Rule 2. That directly illustrates the relation $a \leq b$.

Number. For problems concerning a number representation (whether it is binary, decimal, hexadecimal, ...), one can represent this number as a sequence of digits named d_j. The most significant digit is at the right and the least significant one at the left. Often, the d_j are not variables explicitely used in the code but rather figures that, together, represent the actual variable. If a program must investigate the values of the digits in a certain order, it is possible to mention in the picture which is the first and last digits to be handled, as it is, for example, done in Fig. 3a where the least signifiant digit (in orange) will be used first and the most significant one (in magenta) will be used last.

Array. Fig. 3b shows the representation of an array containing `N` elements. The pattern follows a rectangular shape to depict the contiguous storage of the elements. Above this rectangle, we indicate indices of interest: at least the first (i.e., 0 – always on the left of the drawing, whatever the direction in which the array is processed) and the size `N`. It is important to see that `N` is written at the right of the array's border to mean that `N` is not a valid index as it is out of the array's bounds that are within $[0..N-1]$. The variable name for accessing the array is written at its left.

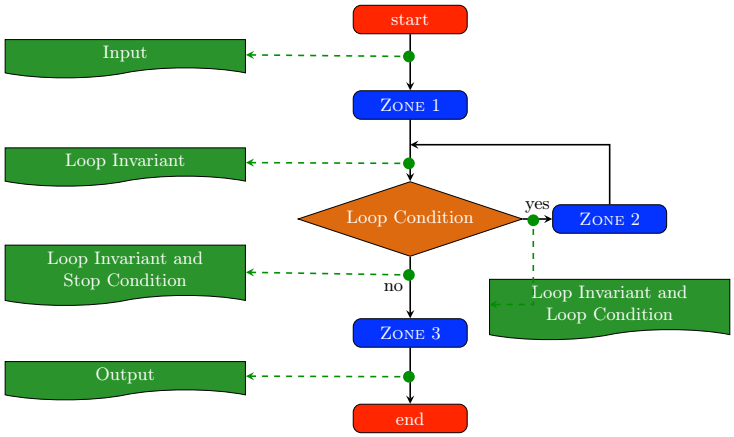


Fig. 4. Loop zones and logical assertions. Blue boxes are block of instructions, orange diamond is an expression evaluated as a Boolean, arrows give an indication of the program flow. Green boxes represent states.

There are other patterns, such as linked lists and files, but those are usually introduced in a CS2 course in our University.

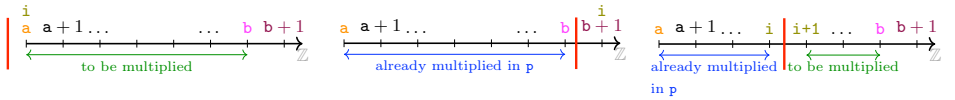
3 Programming Methodology

Once a Gli meets the rules previously introduced, it can be used to write the corresponding piece of code relying on an iterative process. The general pattern of such a piece of code is given in Fig. 4. Input and Output describe the piece of code input (e.g., a and b such that $a < b$ in the example provided in Sec. 2.1) and the result (e.g., the product of all the integers in $[\mathtt{a}, \mathtt{b}]$). By definition, a Loop Invariant must be True before evaluating the Loop Condition. The evaluation of the Loop Condition is not supposed to modify the truth value of the Loop Invariant[2], therefore the Loop Invariant is still True when the Loop Condition is evaluated at True (Loop Invariant and Loop Condition in Fig. 4) or False (Loop Invariant and Stop Condition in Fig. 4). Finally, it is up to the programmer to make sure that the Loop Invariant is True at the end of the iteration, just before the Loop Condition is evaluated, before the potential next iteration.

One can see appearing, in the pattern, four parts that must be filled to form the code : Zone 1, Zone 2 and Zone 3 (standing for instruction(s)) and Loop Condition (standing for a boolean expression). It is worth noting that deducing each part can be done independently, and this, with the help of the Gli.

To be precise, each part is surrounded, in Fig. 4, by commentaries (in green) that represent conditions that must be satisfied, i.e., be True (e.g., in Fig. 4, Zone 1 is surrounded by *Input* and *Loop Invariant*). While filling the code of a

[2] To make it simple, we do not consider here side effect expressions, e.g., pre- or post-increment.



(a) Zone 1. (b) Loop Condition and Zone 3. (c) Zone 2 after having multiplied p by i

Fig. 5. Manipulating the Gli for deducing Zone 1, Loop Condition, Zone 2, and Zone 3 for computing the product of integers between a and b. The corresponding Gli is provided in Fig. 1.

given part, we must take for granted the information contained in the condition that precedes it and find instructions that will ensure that the condition that follows it is True. The following details these four steps : (*i*) Deducing variables initialisation (Zone 1) from the drawing of the initial state; (*ii*) Deducing the Stop Condition (and thus the Loop Condition) from drawing the final state; (*iii*) Deducing the Loop Body (Zone 2) from the Gli; (*iv*) Deducing the instructions coming after the loop (Zone 3) from drawing the final state. These four steps can be achieved in any order, except the Loop Body determination that may require to know the Loop Condition. Both initial and final states are obtained from the Gli through graphical modifications. Those steps are detailed below, illustrated by the example introduced in Sec. 2.1.

It is worth noting that Fig. 4 and the various zones pave the way for a more formal approach in code construction that relies on Hoare's triplet [13,17], with respect to a strongest postcondition code construction approach [11]. For instance, the Loop Body (i.e., Zone 2) may be seen as {Inv ∧ B} Zone 2 {Inv}, where Inv stands for the formal Loop Invariant and B for the Loop Condition. In addition, graphical manipulation of the Gli corresponds to logical assertions describing states between instructions.

Zone 1. First, the Gli provides information about the required variables. In our example, we need four variables: `a`, `b`, `i`, and `p`. `a` and `b` are provided as input to the piece of code. It is worth noticing that the drawing provides a clue about the variables type: they are all on a graduated line labelled with ℤ, meaning they are of type `int`.

The initial values of the variables can be obtained from the Gli by shifting the Dividing Line (in red) to the left in order to make the blue zone (i.e, the zone describing what has been achieved so far by previous iterations) disappear. The variable labelling the Dividing Line (`i`) is also shifted to the left accordingly and stays at the right of the Dividing Line. By doing so, as seen in Fig. 5a, the initial value of `i` must be `a` (i.e., the particular value just below `i` in Fig. 5a). With respect to the variable `p`, we know from the Gli (see Fig. 1a) that it corresponds to the product of all integers between `a` and the left-side of the Dividing Line (i.e., `i-1`). As this zone is empty, we deduce the initial value of `p` as being the empty product, i.e., 1. The following piece of code sums up the deduced instructions for Zone 1:

```
1  int i = a;
```

```
int p = 1;
```

Stop Condition and Loop Condition. Determining the Loop Condition requires to draw the final state of the loop, i.e., a state in which the goal of the loop is reached. Since the purpose of our problem is to compute the product of the integers between `a` and `b`, we can obtain such a representation from the Gli (Fig. 1a) by shifting the Dividing Line (in red) to the right, until the green zone (i.e., "to do" zone) has totally disappeared. In the fashion of Zone 1, the labelling variable `i` is shifted at the same time as the Dividing Line. This graphical manipulation leads to Fig. 5b where we can see that the goal of the loop is reached when $\texttt{i} = \texttt{b} + 1$ and the iterations must thus be stopped. The loop Stop Condition is therefore $\texttt{i} = \texttt{b} + 1$. As the Loop Condition is the logical negation of the Stop Condition, it comes $\texttt{i} \neq \texttt{b} + 1$. However, We recommend, in order to properly illustrate the relationship between `i` and `b`, to use a stronger condition, i.e., $\texttt{i} < \texttt{b} + 1$ or $\texttt{i} \leq \texttt{b}$ that is, of course, equivalent. The following piece of code sums up the deduced instructions for the Loop Condition:

```
while(i <= b)
```

Zone 3. As we just depicted the final state (see Fig. 5b), we can see that the variable `p` holds the product of the integers between `a` and `b`, meeting the program goal. Nothing remains to be done after the loop in this case. However, Zone 3 is not necessarily empty. For example, in a program that computes the average of a certain numbers of values, the loop would sum and count the values and Zone 3 would be the division of the sum by the number of counted values.

Zone 2. Determining the Loop Body is often the most difficult step. We start from what we know: both the Loop Condition and the Gli are True (See the general loop pattern in Fig. 4). We must find instructions such that it will progress the situation towards the program goal. In other words, make the blue zone increase and the green zone decrease. As the blue zone represents the integers that are already multiplied in `p` (thus from `a` to `i`-1), we can make this zone grow by multiplying the next integer to `p`. This next integer is read in the Gli at the right of the Dividing Line: `i`.

After having multiplied `p` by `i`, the situation in Fig. 5c is obtained. It must be noted that is not the Gli anymore since the variable `i` is now at the left of the Dividing Line. In this particular situation, the Gli is False, whatever the particular values of `a`, `b`, `i`, or `p`. According to the loop pattern (See Fig 4), we must recover the Gli, i.e., make it True again, before the end of the Loop Body. By comparing the Fig. 1a and Fig. 5c, we can see that in the Gli, the value labelling the right side of the Dividing Line is `i` and in the current situation, this is `i` +1. Therefore, by assigning the value `i` +1 to `i` (i.e., increasing `i`), the Gli is restored. Finally, the following piece of code shows the Loop Body instructions:

```
p *= i;
i++;
```

4 Learning Tools

This section describes how students can practice the GLI. The main goal is to provide students with a structured and coherent framework so that they do not start their loop design from scratch. To meet that purpose, as early stage, the Blank GLI method is proposed through the *Programming Challenge Activity* (PCA) [18]. The Blank GLI provides a canvas to students. That canvas frames students' solutions so that the semantic of a given solution can be automatically corrected and commented. That GLI correction is handled through a tool called CAFÉ [19][3]. Besides this, CAFÉ also supports GLIDE, a sketching module dedicated to the GLI. Those different components are detailed below and their interaction is illustrated in Fig. 6.

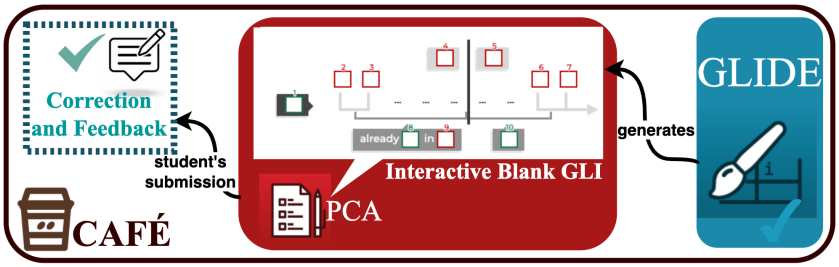


Fig. 6. Link between the Blank GLI, GLIDE, and CAFÉ.

GLIDE. The *Graphical Loop Invariant Drawing Editor* (GLIDE) guides students in drawing their GLI by using the predefined graphical patterns (see Sec. 2.3) and following the first six rules (Sec. 2.2). GLIDE is illustrated in Fig. 7. On the top-left, you can notice a drop-down list itemizing the different drawing patterns. Once a student has selected the appropriate one, they can start formally describing their loop mechanism according to the first six rules. The graphical components the student can use are available on the left. Each of them is mapped to one/several rule(s).

Once a student considers their GLI is completed, they can submit it and some basic checks are performed. In particular, syntactic mistakes are detected (such as the lowerbound being further than the upperbound or some description of what has been achieved so far that is missing). However, the GLI semantic is not verified, which means that the solution can be positively assessed by the GLIDE while the GLI does not make sense.

Interactive Blank GLI. The Blank GLI consists in providing a canvas the students have to fill out. The Blank GLI corresponding to the example introduced in Sec. 2.1 is illustrated as part of Fig. 6. Such a blank drawing depicts only the general shape a correct and rigorous GLI should follow (i.e., partially Rule 1) in

[3] The version of CAFÉ discusses in this paper corresponds to an upgrade with respect to Liénardy et al. [19]

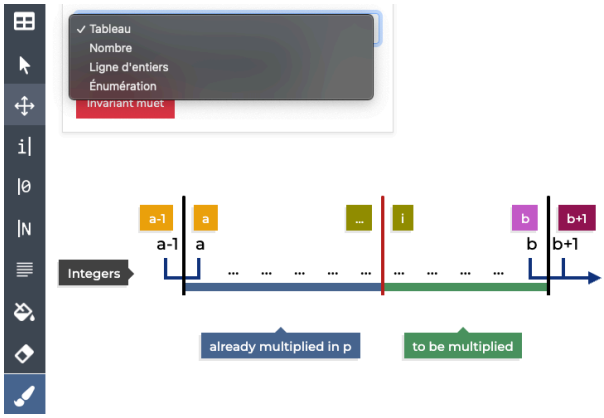


Fig. 7. Screenshot of GLIDE.

response to a given problem. Students must then annotate properly the canvas so that the drawing becomes the figure of their Loop Invariant for their solution to the particular problem to be solved.

Any Blank GLI always comes with two types of box: (*i*) red boxes standing to host expressions (i.e., constants, variables, operations, or left blank) and are to be completed by students without support; (*ii*) green boxes standing to host labels that students must drag and drop from a pre-defined list. That list contains multiple choices, some of them being the expected answers, others being purely random. Doing so, we pave the way for an automatic correction of the GLI (with strong feedback). This can be achieved thanks to the fact each box is numbered. In this way, when a student's solution gets corrected, each piece of solution is easy to be pointed out, allowing to bring a rich feedback while still keeping it clear and smooth to digest for the student. That process is supported by CAFÉ [19].

Programming Methodology with CAFÉ. CAFÉ [19] is a tool we initially developed in order to support a remote programming activity (PCA) [18]. CAFÉ's purpose is to correct students' work and provide instantaneous personalized feedback and feedforward, based on their mistakes. Their mistakes are mapped to error codes classified in a misconception library. That library has been fed based on previous experiences. Some error codes are defined for each step and each zone of the GLI. They also cover most of the inconsistencies that may occur between the GLI and the resulting code to make sure the student really utilizes the methodology. Also, it is worth noticing CAFÉ gives the opportunity to catch students' learning behavior by collecting data.

That correction and feedback scope got extended after having led an assessment of CAFÉ's impact on students' learning [7]. Now, CAFÉ embeds GLIDE and offers a friendly interface to students when they are solving a given problem. That interface (illustrated in Fig. 8) structures and sequences the construction of the solution, aligned with the programming methodology (Sec. 3). In particular,

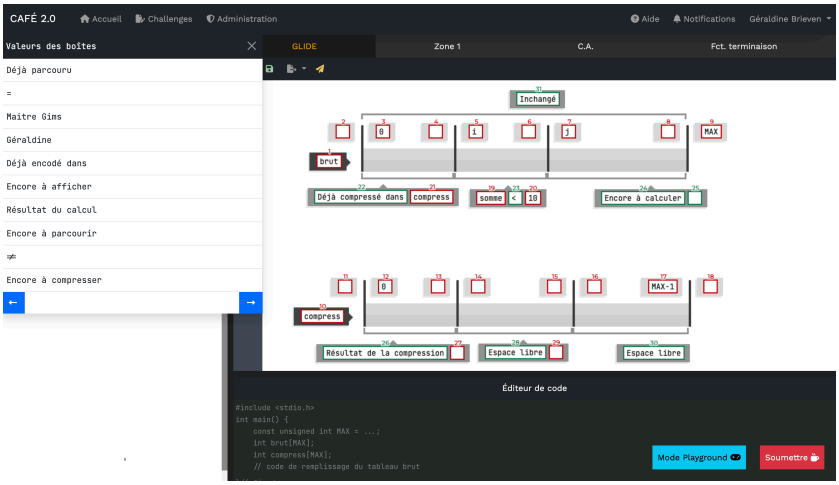


Fig. 8. Interface supporting the programming methodology.

Fig. 8 adresses the problem consisting in compressing a given array, based on consecutive elements whose sum is 10.

On Fig. 8, one can see that the GLI and the code are represented through successive frames. By taking a closer look, it can be noticed that the GLI (in the upper frame) is divided into four tabs, one for the Blank GLI, one for building and justifying (by moving the Dividing Line) ZONE 1, one for building and justifying (by moving the Dividing Line) the Stop Condition, and for the Loop Variant. It is important to notice that a student cannot access the next tab if the current one has not been filled. That locking path approach also applies at a higher level, between the GLI and the coding steps. That feature aims to impose students to sequentially follow the methodology and not directly jump to the code without any proper design to rely on.

5 Preliminary Evaluation

We surveyed students ($N = 70$), after the final exam, with the question "*What drives/discourages you in using the* GLI*?*". 36% of students highlighted the method difficulty, limiting so the advantages of the programming methodology. Then, 30% of students were convinced the methodology is useless and directly coding is manageable. This last opinion may suggest that the problems difficulty exposed to students should be increased, so that the importance of the program methodology would be better highlighted. However, a balance must be found between exercises difficulty and methodology mastering, which requires starting with easy problems. An alternative is to enforce the guidance over the GLI construction (like CAFÉ does in its most recent version), so that harder problems can be provided while remaining accessible.

With respect to GLI construction and tools usage, we can show how much the blank GLI (practiced through the PCA) and GLIDE can be relevant in students learning journey. First, there is a correlation between students' exam grades and students' participation to the PCA ($r = 0.57$, $p < 0.0001$). The GLI approach (supported by the PCA) seems thus to forge students' ability to construct a correct and sound GLI from scratch (which is what students are expected to perform in the exam). This inference gets corroborated by students' opinion collected through another survey ($N = 79$) addressed the year before. More precisely, from the statement "*The Blank* GLI *is useful to find out the* GLI.", 47.4% of students agreed or strongly agreed on, 24.4% disagreed and 25.3% standed in between.

In addition, we looked at the possible correlation between exam grades and GLIDE usage ($r = 0.42$, $p < 0.0001$). The lower impact of GLIDE compared to the Blank GLI may be due to the fact that some students still lack landmarks in using GLIDE while the Blank GLI frames more students' solution. Now that the guidance has been enforced in CAFÉ's last version, we expect to see students reaching an upper step and being able to better take advantage from their experience with GLIDE. That premise is subject to future work.

6 Related Work

While there is an abundant literature on Loop Invariants for code correctness and on automatic generation of Loop Invariants (see for instance [9] or Bradley et al. [6]), their use for building the code has attracted little attention from the research community. With respect to Loop Invariant based programming, the seminal work has been proposed by Dijkstra [11], followed by Meyer [23], Gries [16], and Morgan [24]. As such, the program construction becomes a form of problem-solving, and the various control structures are problem-solving techniques. Those works proposed Loop Invariants as logical assertions.

Tam [28] suggests to introduce students to Loop Invariant as early as possible in their courses and describes several examples of code construction based on informal Loop Invariants expressed in natural language. Astrachan [1] suggests the use of Graphical Loop Invariants in the context of CS1/CS2 courses. However, his approach is incomplete as the suggested drawing lack of completeness (e.g., objects manipulated, such as arrays, are not named in the drawing), might lead to confusion (e.g., variables positions around the dividing line are somewhat unclear), and the drawing is not explicitly manipulated to derive particular situations. Back [2,4,3] proposes nested diagrams (a kind of state charts) representing, at the same time, the Loop Invariant and the code. However, in such a situation, Loop Invariants are expressed as logical assertions. Since, Manilla [21] has evaluated the impact of errors in those nested diagrams. Finally, Erkisson et al. [12] propose a pictorial language for representing Loop Invariants. Their language only applies to arrays and is a mix between drawings (the data structure is drawn and partitions are colored to illustrate universally quantified predicate) and formal languages (the meaning of partitions is expressed as a predicate).

7 Conclusion

This paper introduced a Gli based programming methodology consisting in depicting a graphical representation of the Loop Invariant to solve a given problem prior to writing any piece of code. This methodology is currently taught in a CS1 course. Some preliminary results showed that many students cannot embrace it, mainly because they do not perceive its interest and they miss abstraction skills. Seing that, when we define a problem, a tradeoff must be found between its complexity (so that students feel the purpose of the methodology) and its accessibility (so that students are able to solve it). To reconcile those characteristics, Café was proposed as an integrated learning tool supporting the methodology and guiding students in solving more complex statements. In particular, a resolution framework is provided as well as personalized feedback so that students are able to refine their understanding. Besides this, it enables more transparency about individual students' learning behavior and resulting performance on the Gli thanks to collected data.

In future work, it is planned to harness that data to accurately assess the methodology by closely analysing students' learning path towards mastering the Gli. In particular, we will capture how much time students spend on the Gli and the code, respectively to see if they put their effort on the Gli. We will also track how students construct their solution to confirm they follow the steps suggested by Café. Finally, a focus will be dedicated to the way students read, integrate, and take advantage of the feedback to improve their skills in constructing a Gli. Besides that deeper analysis on the Gli, it is aimed at formalizing the translation from the Gli into a logical assertion in order to end the bridge towards a formal method (being the Loop Invariant here).

References

1. Astrachan, O.: Pictures as invariants. In: Proc. ACM Technical Symposium on Computer Science Education (SIGCSE) (March 1991)
2. Back, R.J.: Invariant based programming. In: Proc. International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (Petri Nets) (June 2006)
3. Back, R.J.: Invariant based programming: Basic approach and teaching experiences. Formal Aspects of Computing **21**(3), 227–244 (May 2009)
4. Back, R.J., Eriksson, J., Mannila, L.: Teaching the construction of correct programs using invariant based programming. In: Proc. 3rd South-East European Workshop on Formal Methods (SEEFM) (November/December 2007)
5. Ben-David Kolikant, Y., Mussai, M.: "so my program doesn't run!" definition, origins, and practical expressions of students' (mis)conceptions of correctness. Computer Science Education **18**(2), 135–151 (June 2008)
6. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer Science & Business Media (2007)
7. Brieven, G., Liénardy, S., Donnet, B.: Lessons learned from 6 years of a remote programming challenge activity with automatic supervision. In: Proc. European Distance and E-Learning Network (EDEN) (June 2022)

8. Cherenkova, Y., Zingaro, D., Petersen, A.: Identifying challenging CS1 concepts in a large problem dataset. In: Proc.ACM Technical Symposium on Computer Science Education (March 2014)
9. Cormen, T.H., Leiserson, C. E .and Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT press (2009)
10. Dale, N.B.: Most difficult topics in CS1: Results of an online survey of educators. ACM SIGCSE Bulletin **38**(2), 49–53 (June 2006)
11. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Inc. (1976)
12. Eriksson, J., Parsa, M., Back, R.J.: A precise pictorial language for array invariants. In: Proc. International Conference on Integrated Formal Methods (IFM) (September 2018)
13. Floyd, R.W.: Assigning meanings to programs. In: Proc. Symposium on Applied Mathematics (1967)
14. Furia, C.A., Meyer, B., Velder, S.: Loop invariants: Analysis, classification, and examples. ACM Computing Surveys **46**, 1–51 (January 2014)
15. Ginat, D.: On novice loop boundaries and range conceptions. Computer Science Education **14**(3), 165–181 (2004)
16. Gries, D.: The Science of Programming. Springer (1987)
17. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (October 1969)
18. Liénardy, S., Leduc, L., Verpoorten, D., Donnet, B.: Challenges, multiple attempts, and trump cards – a practice report of student's exposure to an automated correction system for a programming challenges activity. International Journal of Technologies in Higher Education (IJTHE) **18**(2), 45–60 (June 2021)
19. Liénardy, S., Leduc, L., Verpoorten, D., Donnet, D.: Café: Automatic correction and feedback of programming challenges for a CS1 course. In: Proc. ACM Australasian Computing Education Conference (ACE) (February 2020)
20. Luxton-Reilly, A., Albluwi, I., Becker, B.A., Giannakos, M., Kumar, A.N., Ott, L., Paterson, J., Scott, M.J., Sheard, J., Szabo, C.: Introductory programming: a systematic literature review. In: Proc. Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE) (July 2018)
21. Manilla, L.: Invariant based programming in education – an analysis of student difficulties. Informatics in Education (INFEDU) **9**(1), 115–132 (2010)
22. Medeiros, R.P., Ramalho, G.L., Falcão, T.P.: A systematic literature review on teaching and learning introductory programming in higher education. IEEE Transactions on Education **62**(2), 77–90 (May 2019)
23. Meyer, B.: A basis for the constructive approach to programming. In: IFIP Congress. pp. 293–298 (1980)
24. Morgan, C.: Programming from Specifications. Prentice-Hall (1990)
25. Nilson, L.B.: The Graphic Syllabus and the Outcomes Map: Communicating your Course. John Wiley & Sons (2009)
26. Pólya, G.: How to Solve It. Princeton University Press (1945)
27. Schröter, I., Krüger, J., Siegmund, J., Leich, T.: Comprehending studies on program comprehension. In: Proc. IEEE/ACM International Conference on Program Comprehension (ICPC) (May 2017)
28. Tam, W.C.: Teaching loop invariants to beginners by examples. In: Proc. ACM Technical Symposium on Computer Science Education (SIGCSE) (March 1992)