Graphical Loop Invariant programming in CS1

Simon Liénardy
Université de Liège, Montefiore Institute – Belgium
simon.lienardy@uliege.be

Lev Malcev
Université de Liège, Montefiore Institute – Belgium
l.malcev@student.uliege.be

Benoit Donnet
Université de Liège, Montefiore Institute – Belgium
benoit.donnet@uliege.be

ABSTRACT

This paper introduces the use of Graphical Loop Invariant as a programming methodology in a CS1 course, in which the Loop Invariant is determined prior to writing the code and is meant as a help to find the loop instructions. This paper also introduces two learning tools: GLI, an application helping students to draw Loop Invariant and Café, an on-line platform designed to assess and deliver automatic feedback and feedforward information to students, in particular on their Loop Invariants and the pieces of code based upon them. The paper reports preliminary evaluation on Café usage.

CCS CONCEPTS

• **Theory of computation → Algorithm design techniques**; • **Social and professional topic → CS1**.

KEYWORDS

Café, Graphical Loop Invariant, feedback, CS1, Assessment for Learning

1 INTRODUCTION

In Belgium, the access to the Higher Education curriculum in Computer Science is non-selective. Hence, we cannot make any kind of assumptions about first year students' background. Many of them enter the program with excitement (unfortunately due to a biased vision of the Computer Science field), but without being clearly aware of actual requirements. This contributes to a high failure rate, as well as a high withdrawal ratio throughout the year [5, 6, 28].

Typically, a CS1 course teaches students how 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 proposed in our CS1 course is based on an informal version of the Loop Invariant (a property of a program loop that is verified at each iteration – i.e., at each evaluation of the Loop Condition) introduced by Hoare [15, 16]. Our methodology consists in determining a strategy (i.e., 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 [12] and extended later by Back [2]. The methodology also implies to verify the loop ends. This is achieved, in our methodology, by measuring the progress made by each loop iteration through the definition of a Loop Variant (i.e., a function that measures the progress made by the loop towards the termination). As such, the Loop Invariant and the Loop Variant can be seen as the corner stone of code writing. The problem with such a methodology is that it is a quite abstract reflection phase that might confuse students who may not have the desired abstract background, specially if the Loop Invariant is expressed as a logical assertion. However, if practiced on a regular basis [17, 22] with increasing difficulties problems [24], students can, little by little, master this programming methodology.

In this paper, we propose two tools designed to help student to understand how to determine an Loop Invariant and how use it to derive code instructions. The first one, called Gli, is an on-line application allowing to draw Graphical Loop Invariant. One of the main key aspect of Gli is its ability to provide students with quick feedback and feedforward (i.e., how to correct it) on the Graphical Loop Invariant. The obtained Loop Invariant can then be used to write the corresponding piece of code. The coding processed is also eased by Gli.

The second tool is the result of a reflection that was carried out on a teaching activity that could fulfill the need of regular exercises and that could come in addition to classic exercises sessions (12 sessions of exercises and 5 labs in front of computers in our CS1 course). Limited by multiple constraints (a single Teaching Assistant, time, room availability, etc.), those exercises have to be done at home, with an automatic correction and feedback provided to students.

The main issue of such an automatic system lies in the fact that it is very difficult to, simultaneously, assess the program output and the cognitive process inherent to the program construction. In particular, as the course puts the emphasis on the Graphical Loop Invariant and the Loop Variant, it quickly appeared as desirable to include them in the automatic correction of the student's work. Furthermore, not doing so would have resulted in a dissonance between theoretical lessons and practical sessions.

This is exactly what we propose with our second tool: an online platform, called Café ("Correction Automatique et Feedback des Étudiants") for automatically assessing students' programming exercises. In addition, this platform is inspired by *assessment for learning*[1] (AfL) [24, 29].

In a nutshell, Café proposes to students several programming *Challenges* (roughly one Challenge every two weeks) spread over the CS1 semester. It encourages students to work on a regular basis on increasing difficulties problems. In addition, those Challenges could help students to better understand the course learning outcomes [27] as well as to increase their self-efficacy [3]. Further, for each Challenge, Café allows students to submit up to three times [18] their solution, thus closing the feedback loop [8]. For each submission, Café automatically provides a high quality feedback and feedforward information thought based on the literature [19], maximizing so student auto-regulation.

The remainder of this paper is organized as follows: Sec. 2 discusses and illustrates the programming approach we propose in our CS1 course; Sec. 3 presents the two tools we propose, Gli and Café; Sec. 4 discusses a preliminary evaluation of Café usage; Sec. 5

[1] Assessment for Learning is defined as informing learners of their progress to empower them to take the necessary action to improve their performance [17].

Excerpt 1: Binary search code with zones delimited by the Loop Invariant.

```
int bsearch(int *A, int N){                          <-- zone 1
  int l = 0, u = N-1;

  //Invariant
  while(l < u){
    //Invariant ∧ Loop Condition
    int m = (l + u) / 2;
    if(A[m] < X)                                     <-- zone 2
      l = m + 1;
    else if(A[m] > X)
      u = m - 1;
    else
      u = l = m;
  }
                                                     <-- zone 3
  //Invariant ∧ ¬Loop Condition
  if (A[u] == X)
    return u;
  else
    return -1;
}
```

positions this work with respect to the state of the art; finally, Sec. 6 concludes this paper by summarizing its main achievements.

2 PROGRAMMING METHODOLOGY

A *Loop Invariant* [15, 16] 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, Loop Invariant has been used for proving code correctness (see, e.g., Cormen et al. [10] and Bradley et al. [9] for automatic code verification). As such, the Loop Invariant is used "a posteriori" (i.e., after code writing).

In our CS1 course, we envision a different perspective in which the code is built upon the Loop Invariant that must be thus expressed before coding ("a priori" usage), as suggested by Dijkstra [12] and pushed further by Back [2]. This allows us to divide the code in three main zones, as illustrated in Listing 1, each zone being constructed thanks to the Loop Invariant. *Zone 1* refers to the code segment prior to the loop, typically used for initializing variables. At the end of Zone 1, the Loop Condition is evaluated, meaning that the Loop Invariant must be verified. Based on this statement, the Loop Invariant can be used to determine the required variables as well as their initial values. *Zone 2* refers to the Loop Body itself. As the Loop Condition has been verified, before executing any instruction of the Loop Body, the Loop Invariant is true and the Loop Condition is true. At the end of the Loop Body, the Loop Condition is evaluated again, meaning the Loop Invariant must be restored. Based on those two situations, one can derive the Loop Body instructions. *Zone 3* refers to the piece of code after the loop, when the Loop Condition has been invalidated. This zone contains instructions that should allow the program to finally solve the initial problem. Given that the Loop Condition has been evaluated, the Loop Invariant is true but the Loop Condition is false. Based on this situation, it is possible to derive the final instructions.

While Dijkstra expressed Loop Invariants as logical assertions, we believe this could be counter-productive in the context of a CS1 course, in which students may not have the required level of abstraction. Instead, we build our methodological approach on an informal version of Dijkstra's process by proposing an informal *Graphical Loop Invariant*.

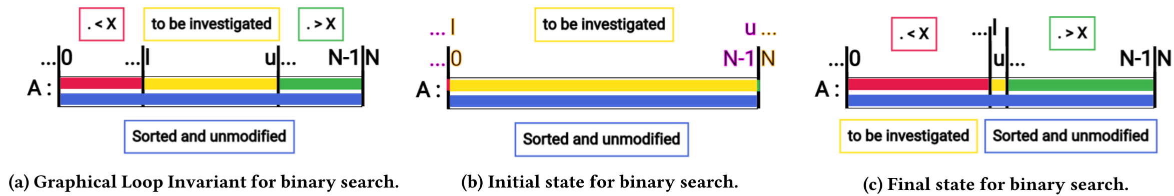
The Graphical Loop Invariant is supposed to contain key information that will eventually be used to actually write the code. As such, the Graphical Loop Invariant represents a strategy to solve the problem and is used to support thoughts on the code. Although being informal, this drawing must at least detail variables, constant(s), and data structures manipulated by the program; the constrains on them; the relationships they may share, and that are conserved all over the iterations. It should also express, in a general way, what has been already computed by the program after a certain number of loop iterations. With this method, we clearly shift the difficulty not anymore in writing the code itself but in the reflection phase that is prior to the code. This step requires thus training and experience. But, once mastered, it becomes possible to efficiently solve complex problem.

In the remainder of this section, we illustrate this process (from drawing the Graphical Loop Invariant to writing the code) with a well-known problem: the binary search in a sorted (in the increasing order) array `A` of length `N`, `A` being indexed from `0` to `N-1`. The function implementing the binary search will return the index of the researched value `X`, or a special value `-1` if `X` does not belong to `A`.

The basic idea of the binary search is to divide, at each step, the search zone based on the ordered property of `A` and the value of `X`. Generally speaking, one can thus divide `A` in three zones: (*i*) array elements $< X$, (*ii*) array elements $> X$, and (*iii*) array portion in which the search must be performed.

Those considerations are depicted in Fig. 1a where we delimit the $\cdot < X$ zone with a variable `l` (for **l**ower indices) and the $\cdot > X$ zone with variable `u` (for **u**pper indices). As can be seen in Fig. 1a, we carefully depict the array as a rectangle labeled with the array name and the indices written above it. Labeling a data structure with its name is, at first, useful for the understanding and becomes mandatory when multiple structures come into the game. It is also the first required step for making the Loop Invariant coherent with the code when it will be written. It should be noted that, in our Figure, the $\cdot < X$ zone is formally comprised between indices 0 and `l` − 1, which is represented by the letter `l` written at the right of the vertical bar determining this zone. We call such a vertical bar *dividing line* (it appears thicker in Fig. 1a). This accuracy in the drawing is of the highest importance to easily determine `l` initial value . The same remark applies for `u`. This Graphical Loop Invariant also makes easier the discovery of a Loop Variant (i.e., a function that measures the progress made by the loop towards the termination). It is, simply, the length of the "to investigate" zone, i.e., $u - l + 1$ in Fig. 1a.

As explained earlier in this section, the Graphical Loop Invariant can be used to divide the code in three zones and to write the code of those zones. In particular, Fig. 1b illustrates, graphically, the situation in the first zone. This is obtained by stretching the two dividing lines to their minimum (for `l`) and maximum (for `u`) values. At start, we cannot ensure that any values in `A` is less than X, neither we can ensure that any values in `A` is greater than X. Hence, the corresponding zones in the array `A` at the initialization are empty, as shown in Fig. 1b. From this drawing, we see that in this situation, `l` (resp. `u`) corresponds to index 0 (resp. $N-1$) and we



(a) Graphical Loop Invariant for binary search. (b) Initial state for binary search. (c) Final state for binary search.

Figure 1: Graphical Loop Invariant and particular cases for binary search. The figures were obtained thanks to the tool described in Sec. 3.1

conclude that we must initialize l and u to these respective values, as done in Line 2 of Listing 1.

Similarly, we can modify Fig. 1a to depict the loop final situation (see Fig. 1c). The zone to be checked in A (i.e., the zone A[l...u]) has now the minimal size, leading to a case where l == u. In such a case, the loop must be stopped and one can derive the Loop Condition (i.e., $l \neq u$). Pedagogically speaking, we can use a lighter condition, $l < u$ that has the advantage to better represent the relationship between those indices.

Zone 2 (i.e., Loop Body) consists in making the loop progressing toward the final objective: finding X and returning its position in A. To do so, we must investigate the unknown zone (labeled "to investigate" in Fig. 1a) and make growing one of the two other zones, thus moving the associated dividing line. The binary search consists in looking the middle of that zone, at index $m = \frac{l+u}{2}$ (line 7 in Listing 1). If it appears that $A[m] < X$, one concludes that X does not belong in the left part of A and that the left dividing line must be translated to the right (line 9 of Listing 1) by taking the value $m + 1$ (+1, since we know that $A[m] < X$). A similar reasoning can be made for the $A[m] > X$ case. Finally, otherwise, the loop must end (i.e., A[m] == X). This is achieved by forcing the final situation, as illustrated in Fig. 1c (see line 13 in Listing 1).

Finally, zone 3 is reached once one leaves the loop, i.e., when l == u, as explained by Fig. 1c. In such a case, the zone "to investigate" contains a single element. We thus only have to check whether this element is X or not (lines 17 → 20 in Listing 1).

To sum up, here are the guidelines a student should follow when searching for a *good* Graphical Loop Invariant, according of the methodology:

(1) The drawing corresponds to the problem;
(2) The boundaries of the problem are provided;
(3) One (or more) dividing line(s) is (are) provided;
(4) Each dividing line is properly labeled (e.g., with a variable);
(5) The drawing is labeled for explaining what has been achieved so far (this often introduces new variables);
(6) The drawing is labeled to indicate what should still be done;
(7) All the named structures and variables are present in the code.

These focus points can also be used to evaluate students' performance. The typical mistakes can be sorted into three main categories: (*i*) syntax (i.e., the Graphical Loop Invariant is poorly drawn or the items are misplaced or absent – guileine 1 → 4), (*ii*) semantic (i.e., what was drawn does not represent an Loop Invariant or the labels explaining what has been achieved so far does not make sens – guideline 1 and 5 → 7), and (*iii*) code matching (i.e., the program does not correspond to what has been drawn).

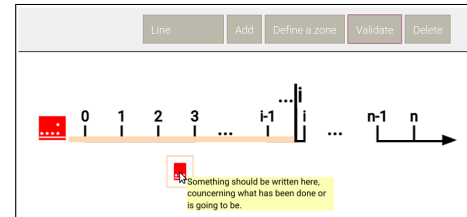


Figure 2: Gli screenshot. Missing items or errors are mentioned in red. More details about the problem and how it can be solved is provided by a tooltip.

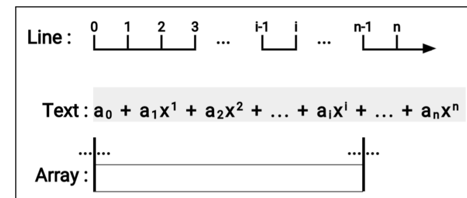


Figure 3: Pattern available in Gli to draw Graphical Loop Invariants. The text zone is here used to depict a polynomial thanks to subscripts and superscripts

3 LEARNING TOOLS

This section introduces two tools that were developed to help students grasping the methodology presented in Sec. 2: the first one enables to draw easily Graphical Loop Invariant (Sec. 3.1), the second one is an on-line platform for automatically assessing students' programming exercises (Sec. 3.2).

3.1 Gli Tool

The Graphical Loop Invariant (Gli) tool is an application written in javascript using the FabricJS HTML5 canvas library [30]. It enables students to easily draw Graphical Loop Invariants that respect the guidelines mentioned in the Sec. 2.

The GUI of this application is quite simple, as can be seen in Fig.2. There are five buttons:

Piece of Invariant enables to select the piece of Invariant to add in the drawing. Three kinds of pattern are implemented: a graduated line (e.g., to represent a range of integers), a 1-dimension

array, and a text zone that allows to represent a large number of problem. Fig. 3 illustrates these patterns.

Add actually draws the selected pattern in the canvas.

Define a zone enables to draw a colored zone in order to define what was already computed and what should still be done. The legend of the zone must be provided in the box of the same color.

Validate launches several tests checking whether the Loop Invariant respects the guidelines (See Sec. 2). More details are provided in Sec. 3.1.1.

Delete enables to delete an item from the canvas.

The user can also click on a drawn piece of invariant to add a dividing bar and label it. Two bars can be linked with a colored bar, defining so a zone. An example of fully labeled Graphical Loop Invariant is provided in the Fig. 1a.

3.1.1 Graphical Loop Invariant Validation. The application can perform several checks ensuring the methodological guidelines are respected: all the drawings must be named (guidelines 5 and 6), each dividing line and the boundaries of the problem must be labeled with a variable or a constant (guideline 2 and 4), there should be at least one zone that indicates what was previously computed and one zone that designates what should be done (guidelines 3 and 5, 6). If one of the tests fails, an error is reported in red in the canvas and a tooltip appears when the mouse hovers the red zone, as cdepicted in Fig. 2. It is worth noting that these tests can only challenge the syntactical part of the guidelines (e.g., the presence of a particular item) and not their semantics (e.g., the soundness of a particular drawing with respect to the problem actually being tackled). Nevertheless, the tool was designed to give students quick feedback on the form of their Graphical Loop Invariant and serve as a reminder in case of missing items.

3.1.2 Writing the Code with Gli. Once the validation step is passed without error, the Gli tool can be used to write the associatd piece of code, as explained in the Sec. 2. The dividing lines can be displaced to transform the Loop Invariant into the initial step (Zone 1) and the final step (Zone 3). From the initial steps, the initial values of the variables can be deduced, Gli can even highlights the matching of two overlapping bars, as illustrated in Fig. 1b. The final state illustrates the Loop Condition, as shown in Fig. 1c.

3.2 Café

This section introduces Café ("Correction Automatique et Feedback des Étudiants"), our on-line platform for automatically assessing students' programming exercises. We use Café through six assignments called *Challenges* distributed throughout the first semester approximately every two weeks. Challenges account for 10% of the final grade, each Challenge having the same weight. Each Challenge consists in a small –yet not too easy– programming/algorithmic task in C.[2] The first Challenge (called "Challenge 0") helps students in grasping how Café works and does not account in the final mark. It is worth to notice that we allow students to not submit one of the five Challenges (concept of *Joker*). In that case, the Challenge does not account in the final mark.

[2] Café has been designed to be programming language independent, meaning that it can be used with other programming languages (e.g., Python).

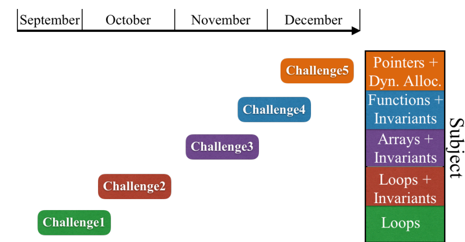


Figure 4: Challenges timeline over the semester.

The Challenges are of increasing complexity (from a simple loop to write – Challenge 1 – to a modular program solving a reasonably complex problem – Challenge 4), referring thus to assessment for learning (AfL) [22, 24]. The last Challenge is dedicated to pointers and dynamic allocation, as we noticed those particular topics appear to be difficult for students. The Challenges timeline is illustrated in Fig. 4 (Challenge 0 not shown).

3.2.1 Students Interactions with Café. A Challenge lasts three days. The first day, the subject is made available for download on the course blackboard. In addition to the subject, students must download a template to fill in with their answers. The correct way to format the answer in the template is provided in the Challenge subject as well as in the template itself.

Once ready, the student answer can be uploaded to Café via a web platform. Café immediately corrects it and produces a feedback and a feedforward that are directly made available to the student. She can then consider these feedback and feedforward to improve her answer and submit it again [14]. Students benefit up to two retries (for a total of three submissions [18] – this way, we avoid the traditional trial and error approach) over the Challenge duration. This process enables the students to learn from their errors by actually taking into account the feedback and feedforward and by submitting an improved solution, closing so the feedback loop [8]. Doing so prevents also the student from being bogged down. At the end, only the last submission accounts for the mark.

Loop Invariant in Challenges. As most of the Challenges consist in writing loops and as the course requires to write loops based on Graphical Loop Invariant (see Sec. 2), the Challenges must embed Loop Invariants so that students can train themselves. At first, it may appear difficult to combine automatic correction and graphical representation. We solve this by asking students to fill in a blank Graphical Loop Invariant. Such a blank drawing depicts only the general shape that should follow a correct and rigorous Loop Invariant (i.e., guideline 1). Students must then annotate properly the figure so that the drawing becomes their Loop Invariant for their solution to the particular problem to be solved. An example of blank Graphical Loop Invariant– for a Challenge whose goal is to compute the intersection of two sorted arrays, `A` and `B`, and to place the result in a third one, `C` – is provided in Fig. 5. The instructions state that the green boxes 1. to 15. should be replaced by variables or constants names or left blank. The box 16. has to be replaced by a number corresponding to the multiple choice: 1: different from; 2: common to; etc. (some inconsistent possible answers are added). Finally, the boxes 17. to 19. must be replaced by a number corresponding to one of the part written in red in Fig. 5. A mock example of such a replacement is always added in the instructions. The way

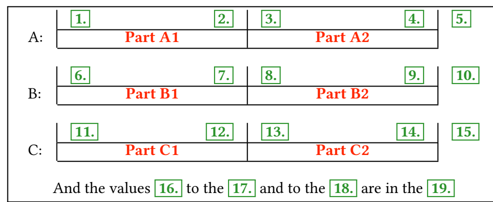


Figure 5: Example of a blank Loop Invariant given in a Challenge consisting in computing in C the intersection of two sorted arrays A and B.

to encode the Graphical Loop Invariant in the Challenge template is also clarified in the instructions, with an example, to be the as clear as possible. It is expected students make use of Gli for building the Challenge Graphical Loop Invariant.

3.2.2 Automatic Correction and Loop Invariant. Café is written in Python 2.7 and easily extensible for new features. The Python script is run in a dedicated sandbox (for avoiding any security issue), on a submission platform, each time a student submits a Challenge. Currently, correcting and grading a Challenge is done in three main steps: (*i*) the preprocessing (i.e., splitting the submitted Challenge into several answers, (*ii*) the correction per se (i.e., each answer is corrected, graded, and commented, and (*iii*) feedback generation (i.e., the various grades are combined and comments are concatenated to form the feedback to be provided to students). It is worth noticing that the steps are independent from each other: one can easily modify the correction step as soon as it handles the answers from the preprocessing and generates data that can be transformed into feedback at the next step.

As long as the code is concerned, the correction step is in charge of comparing the output of the preprocessing step with the expected result(s).

The correction step is able to check whether syntactic constraints are met by the code (e.g., using a `while` loop instead of a `for` one). In addition, by modifying student's code before the compilation, Café can also count the number of loop iterations to verify whether the code is compliant with complexity constraints and ensure that all the array accesses are within the array bounds.

Regarding the Loop Invariant, the correction step verifies that what has been proposed as replacement of the boxes (see Sec. **??** and Fig. 5) is relevant. Most of the time, several answers are possible and are considered by Café.

Finally, there is always the risk a student will submit her Challenge with the Loop Invariant produced after the code (which clearly violates the methodology we propose). To limit this risk, Café checks if variables used in the Loop Invariant are consistent with the one in the code and if they are initialized accordingly (Guideline 7). The matching between the Loop Condition and the Loop Invariant is checked by verifying that the Loop Condition makes used of the proper variables and leads to the correct number of iterations. The Loop Body is not checked against the Loop Invariant as it would be too time consuming to design a system that would cover all the code alternatives. If both the Loop Invariant seems correct and the code produces the expected results, we "a priori" believe the student has followed the methodology. Anyway, a student that would write the code first and later the Loop Invariant would, first, work twice and, second, just lie to herself.

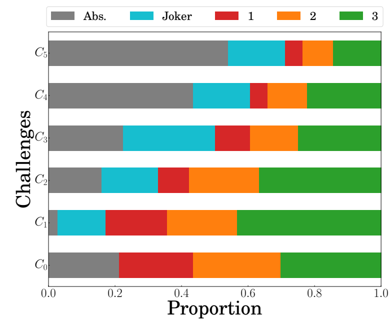


Figure 6: Distribution of students' involvement in Challenges over the semester. 1, 2 and 3 refer to the number of submissions, "Joker" to Students not submitting for the first time, and "Abs." to Students not submitting at least for the second time.

4 PRELIMINARY EVALUATION

The platform on which we deploy Café allows us to collect various data on students involvement and performance in Challenges. In addition, we conducted a survey, during Academic Year 2018–2019, between February 2019 and March 2019, after the course and the Final Exam. The survey was anonymous, to let the students express freely their opinions. We received 22 answers over the 30 students who still continued their curriculum during the second semester. This section investigates the data collected in 2018–2019 and discusses lessons learned from Café usage.

One key point with Café is to ensure students involvement in the course and to ensure a minimum amount of regular practice over the course duration. Fig. 6 shows the distribution of students' involvement in Challenges over the semester. It is worth reminding that students can play a Joker during the semester (see Sec. 3.2). After this Joker has been played, any non-submission is accounted as an "Absence".

Students' involvement in Café is pretty good in the beginning of the Semester (above 60% until Challenge 2). A drop is observed starting Challenge 3 (that follows the Mid-Term Exam organized during All Saints week). It seems that the Mid-Term plays an important role regarding students' involvement. In 2018–2019, the participation rate decreases over the semester until reaching a low 28% of the students participating to Challenge 5. This is aligned with the attrition rate observed during the final Exam (61% of Participation).

The survey conducted shows that the majority of respondents (18/22, 81.8%) acknowledges Café and Challenges for forcing them to work on a regular basis over the semester.

Café allows students to submit a solution to a given Challenge up to three times, with feedback and feedforward sent back to students after each submission. Fig. 6 also shows how students manage multiple submissions. Multiple submissions (i.e., 2 or 3) are common, suggesting so that feedback and feedforward provided by Café are useful for students for improving their solution. It is confirmed by the survey results, as it appears that, for 59.1% of the students, the feedback provided by 3 Challenges or more enabled them to better understand the course (31.8% of them if we reduce to 1 or 2 Challenges). Also, 45.5% of the students admit that the feedback helped them to realize they had a learning gap regarding the subject tackled by 3 Challenges or more (36.4% of them if we reduce to 1

or 2 Challenges). Finally, after receiving the feedback, 59.1% of the students admit they went back, for 3 Challenges or more, to the theoretical course to reread the corresponding theoretical notions (13.7% of them if we reduce to 1 or 2 Challenges).

From the surveys, some students mentioned that the Challenges were too easy compared to the Mid-Term and the Final Exam, highlighting so a constructive alignment issue [7]. This perceived difference is mainly due to the fact that the overall Loop Invariant structure is provided with the Challenge but not for the formal evaluations. However, a large part of students (67.9% in 2017–2018, 77.3% in 2018–2019) agreed that the Challenges enabled them to understand the Loop Invariant determination and few of them (2/22) acknowledged, in open comments, that the concept of Loop Invariant was understood "thanks to the given blank Loop Invariant". This tends to confirm the bootstrap effect of the blank Loop Invariant and suggests to focus the classroom sessions on graphical methods to find and master the Loop Invariant since it is not fully tackled by the Challenges

5 RELATED WORK

While there is an abundant literature on Loop Invariants for code correctness and on automatic generation of Loop Invariants (e.g., [10]), their usage for building the code has attracted little attention from the research community. Tam [26] proposed incomplete and informal Loop Invariants written in natural language. Astrachan [1] is probably the closest to our approach as he proposed Graphical Loop Invariants. Finally, Back [2] proposed nested diagrams (a kind of state charts) representing, at the same time, the Loop Invariant and the code. None of these approaches come with an automatic system that is able to assess the code construction and the Loop Invariant.

Many automated system for providing feedback to programming exercices were already proposed (e.g., [4, 11, 13, 20, 21, 23]). Most of them apply test-based feedback, i.e., student's code is corrected through unit testing testing (except UNLOCK [4] that tackles the problem solving skills in general, not just coding skills). Web-CAT [13] even makes students write their own tests too. Kumar's Problets [20] enables step by step code execution as part of feedback. More advanced automatic feedback has been proposed by Singh et al. [25] by providing, to students, a numerical value (the number of required changes) and the suggestion(s) on how to correct the mistake(s).

6 CONCLUSION

This paper introduces a methodology for programming based upon Graphical Loop Invariant we use in a CS1 course. In order to ease students' learning, two tools were proposed: Gli, an application for drawing Graphical Loop Invariants and Café, an online platform to assess small programming Challenges, including the Loop Invariants used to write the code.

Café has been used, in the context of a CS1 course using the C programming language but could be very easily adapted to Java or Python. Café enables to make the students work and improve their solution according to feedback and feedforward information, three times within three days and at least five times during the semester. This would be unfeasible without an automatic assessment.

The preliminary evaluation provided in this paper does not allow to conclude on Café's effect on students' performance. However, a longer use of Café will enable to collect information about how it impacts students' understanding of the course and the programming methodology and how it improves their programming capabilities. Gli's impact will be studied in the same way. The evaluation of both the methodology and the two tools is an ongoing work.

REFERENCES

[1] O. Astrachan. 1991. Pictures as Invariants. In *Proc. Technical Symposium on Computer Science Education (SGICSE).*
[2] R-J. Back, J. Eriksson, and L. Mannila. 2007. Teaching the Construction of Correct Programs using Invariant Based Programming. In *Proc. 3rd South-East European Workshop on Formal Methods.*
[3] A. Bandura. 1993. Perceived self-efficacy in cognitive development and functioning. *Educational psychologist* 28, 2 (1993), 117–148.
[4] T. Beaubouef, R. Lucas, and J. Howatt. 2001. The UNLOCK System: Enhancing Problem Solving Skills in CS-1 Students. *ACM SIGCSE Bulletin* 33, 2 (June 2001), 43–46.
[5] T. Beaubouef and J. Mason. 2005. Why the High Attrition Rate for Computer Science Students: Some Thoughts and Observations. *ACM SIGCSE Bulletin* 37, 2 (June 2005), 103–106.
[6] J. Bennedse and M. E. Caspersen. 2007. Failure Rates in Introductory Programming. *ACM SIGCSE Bulletin* 39, 2 (2007 2007), 32–36.
[7] J. Biggs and C. Tang. 2011. *Teaching for Quality Learning at University* (4th ed.). Open University Press.
[8] D. Boud. 2000. Sustainable Assessment: Rethinking Assessment for the Learning Society. *Studies in Continuing Education* 22, 2 (August 2000), 151–167.
[9] A. R. Bradley and Z. Manna. 2007. *The Calculus of Computation: Decision Procedures with Applications to Verification.* Springer.
[10] T. H Cormen, C. E Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms.* MIT press.
[11] G. Derval, A. Gego, P. Reinbold, B. Frantzen, and P. Van Roy. 2015. Automatic Grading of Programming Exercises in a MOOC Using the INGInious Platform. In *Proc. European MIIC Stakeholder Summit (EMOOC).*
[12] E. W. Dijkstra. 1976. *A Discipline of Programming.* Prentice-Hall, Inc.
[13] S. H. Edwards and M. A. Perez-Quinones. 2008. Web-CAT: Automatically Grading Programming Assignments. In *Proc. Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE).*
[14] N. Falkner, R. Vivian, D. Piper, and K. Falkner. 2014. Increasing the Effectivement of Automated Assessment by Increasing Marking Granularity and Feedback Units. In *Proc. ACM Technical Symposium on Computer Science Education (SIGCSE).*
[15] R. W. Floyd. 1967. Assigning Meanings to Programs. In *Proc. Symposium on Applied Mathematics.*
[16] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580.
[17] C.A. Jones. 2005. *Assessment for learning.* Learning and Skills Development Agency.
[18] V. Karavirta, A. Korhonen, and L. Malmi. 2006. On the Use of Resubmissions in Automatic Assessment Systems. *Computer Science Education* 16, 3 (September 2006), 229–240.
[19] H. Keuning, J. Jeuring, and B. Heeren. 2019. A Systematic Literature Review of Automated Feedback Generation for Programming Exercices. *ACM Transactions on Computing Education (TOCE)* 19, 1 (January 2019).
[20] A. N. Kumar. 2013. Using Problets for Problem-Solving Exercises in Introductory C++/Java/C# Courses. In *Proc. IEEE Frontiers in Educatoin Conference (FIE).*
[21] R. Lobb and J. Harlow. 2016. Coderunner: a Tool for Assessing Computer Programming Skills. *ACM Inroads* 7, 1 (March 2016), 47–51.
[22] D. Nicol. 2009. *Quality Enhancement Themes: The First Year Experience: Transforming Assessment and Feedback: Enhancing Integration and Empowerment in the First Year.* The Quality Assurance Agency for Higher Education.
[23] N. Parlante. 2011. CodingBat: Code Practice. https://codingbat.com [Online; accessed: 30 March 2019].
[24] K. Sambell, L. McDowell, and C. Montgomery. 2013. *Assessment for Learning in Higher Education.* Routledge.
[25] R. Singh, S. Gulwani, and A. Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*
[26] W. C. Tam. 1992. Teaching Loop Invariants to Beginners by Examples. In *Proc. Technical Symposium on Computer Science Education (SIGCSE).*
[27] V. Tinto. 1999. Taking Retention Seriously: Rethinking the First Year of College. *NACADA Journal* 19, 2 (Fall 1999), 5–9.
[28] C. Watson and F. W. Li. 2014. Failure Rates in Introductory Programming Revisited. In *Proc. Conference on Innovation & Technology in Computer Science Education (ITiCSE).*

[29] D. Wiliam. 2011. What Is Assessment for Learning? *Studies in Educational Evaluation* 37, 1 (March 2011), 3–14.

[30] J. Zaytsev, S. Kienzle, and A. Bogazzi. 2008. Fabric.js, a powerful and simple Javascript HTML5 canvas library. http://fabricjs.com [Online; accessed: 24 October 2019].