

Another Brick in the Loop: From Graphical Loop Invariant to Formal Proofs

Géraldine Brieven[0000−0003−1410−1470]✉, Antoine Grosjean, Pascal Fontaine[0000−0003−4700−6031], and Benoit Donnet[0000−0002−0651−3398]

Université de Liège, Montefiore Institute, Liège, Belgium
gbrieven@uliege.be, Pascal.Fontaine@uliege.be, benoit.donnet@uliege.be

Abstract. The steep learning curve of formal methods deters their adoption in introductory programming courses (CS1), even though such methods can be instrumental in teaching students rigorous programming practices. As part of our CS1 course, we use Graphical Loop Invariants (Glis). These semi-formal diagrams describe relationships between data structures and variables. They substitute Formal Loop Invariants, which are inaccessible to students without a strong mathematical background. However, this gain in accessibility comes at a cost: the semi-formal nature of Glis complicates automated verification. To address this challenge, we present a proof-of-concept framework that translates Gli-annotated C code into Dafny, a verification-aware language. Glis are translated into Dafny predicates by extracting three core components: (*i*) the traversed data structures, (*ii*) the zone delimiters, and (*iii*) the textual zone descriptions, including accumulator variables. The textual descriptions are matched to predicates based on semantic similarity, computed using a Sentence-BERT model. This matching achieves 92.5% accuracy on our test set. We validate the framework on ten representative CS1 problems, whose solutions are correctly verified by Dafny. We describe the pipeline for one of these examples: finding a maximum value in an array.

Keywords: Graphical Loop Invariant · Correctness-by-construction · Formal Methods in Education · Dafny · Similarity sentence measure

1 Introduction

Dongol et al. [17] argue that Formal Methods (FM) thinking is a mindset that computer scientists should develop from their first day of study. We implement this vision in our CS1 course by fostering the habit of reasoning about program properties rather than solely testing. When students solve loop-based problems, they should use Graphical Loop Invariants (abbreviated as Glis) [7]. Since most students lack foundations in logic and mathematics [1], this semi-formal visual and textual approach avoids the notational overhead of mathematical formalism.

Usually, the Loop Invariant is a tool to establish code correctness *a posteriori* (see for instance the work of Cormen, Leiserson, Rivest and Stein [13] or Bradley and Manna [4]). In contrast, in our Gli-based programming methodology, the

Gli is designed and the code is derived afterward, as proposed by Dijkstra [16] and extended by Meyer [23], Gries [20], and Morgan [24]. Back [2] later introduced nested diagrams resembling state charts that represent both the invariant and the code structure, still relying on logical assertions. More recently, Mannila [21] developed an approach targeting novice programmers through visual program construction designed to minimize notational overhead. Like our approach, Loop Invariant states are represented visually. However, predicates are defined within the diagrams while we use textual notations. Mannila also conducted an empirical study on common student errors in Loop Invariant construction, including incorrect variable updates and loop guard mistakes. More recently, Fowler, Kraemer, Sitaraman and Hollingsworth [19] documented students' difficulties in using Loop Invariants to understand and debug algorithms. This further motivates the need for accessible tooling support.

In practice, our students train on designing a Gli (from which they should derive the code) using the Café 2.0 platform [8]. This platform runs unit tests on students' code. It also checks whether the code is consistent with the Gli, primarily by verifying that the variables introduced in the Gli appear in the code with their initial values matching those suggested by the Gli, and that the loop guard aligns with the Gli's final state. However, it does not provide rigorous consistency checking between the Gli and the loop body. This gap stems directly from the textual nature of the Gli's zone descriptions, which resists straightforward automated analysis. To address this gap, we present a framework that translates Glis, coupled with a C implementation, into Dafny-annotated code. To ensure reliability, we adopt an encoder-only approach (Sentence-BERT [29]). And to avoid introducing variability and iterative prompting to generate predicates using LLMs, we predefine a set of predicates and map student sentences to them through semantic similarity.

To evaluate our framework, we focus on ten problems and use a subset of C suitable for our CS1 course, testing both consistent and inconsistent solutions. Our problem statements involve at most two nested loops and the object(s) being iterated over can be arrays, integer intervals, number digits, and terminal positions for drawing figures. This paper describes the worked example of finding a maximum value in an array. Our results demonstrate the feasibility of this framework as a basis for integrating formal verification feedback into the Café 2.0 platform, providing students with rigorous yet accessible consistency guarantees.

The paper is organized as follows. Section 2 introduces the key concepts of the (Graphical) Loop Invariant. Readers familiar with Loop Invariants may skip most of this section, though we note that we advocate a correct-by-construction approach in which invariants are written *a priori*. We also use a visual, semi-formal representation of Loop Invariants. Section 3 describes the context of the CS1 course where Gli-based programming is taught. Section 4 describes the different steps of our framework. Section 5 summarizes the results, and Section 6 concludes with a discussion of future work. An artefact is available at `https://gitlab.uliege.be/cse/gli2fli`. This source code is still subject to change as we extend its capabilities and address remaining limitations.

2 Prerequisites

The use of FM in industry has increased, and so has their relevance to education [33]. Programmers should adopt a correctness-oriented mindset, for instance using informal assertions in natural language to document and reason about *what's true there* during program construction [17,10].

At the same time, in CS1 lectures, teaching loops may definitely benefit from a deeper structural explanation, seeing how often it is poorly mastered by novices [12]. The Loop Invariant concept addresses this issue by providing a methodology for designing correct loops. However, insufficient mathematical and logical background [22] often prevents students from correctly understanding Loop Invariants in their traditional format (see Subsection 2.1). To improve accessibility, we adopt a formulation that combines graphical representations (i.e., drawings) with textual descriptions (see Subsection 2.2).

2.1 The Invariant Expressed Formally (Formal Loop Invariant)

```
function findMax(int * a, unsigned N)
    input     : a, a non-empty array of N integers
    output    : index, the index of a maximum in a
    requires  : N > 0 ∧ |a| = N
    ensures   : a == old(a)
    ensures   : 0 ≤ index < N
    ensures   : ∀k . 0 ≤ k < N ⟹ a[index] ≥ a[k]
1   unsigned i = 0, index = 0
2   while i < N do
        invariant: a == old(a)
        invariant: 0 ≤ i ≤ N
        invariant: 0 ≤ index < N
        invariant: ∀k . 0 ≤ k < i ⟹ a[index] ≥ a[k]
        variant  : N − i
3       if a[i] > a[index] then index = i
4       i = i + 1
5   return index
```

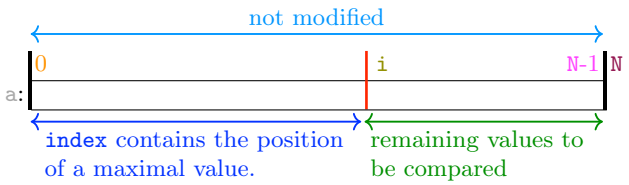
Fig. 1. Fully annotated pseudocode for finding a maximal value in an array

Fig. 1 provides the pseudocode for our worked example. It also includes the pre- and postcondition enforcing correctness, the invariant, and the loop variant. Applying an *a priori* approach, the programmer starts with the precondition. In the example, the existence of a maximum element requires the array to be non-empty, as expressed in the **requires** line. Second, the programmer focuses on

the expected program behavior. They identify that (*i*) the array should remain unchanged, (*ii*) the returned value (`index`) must be a valid array index, and (*iii*) this index must refer to a maximum element in the array. This postcondition corresponds to the **ensures** lines. Since the postcondition requires a property to hold for all elements, a loop iterating over the array is needed. The iteration variable `i` ranges from 0 to `N`. The postcondition is generalized to the intermediate solution state, forming the invariant (lines **invariant**). Once established, the loop body follows naturally: to maintain the invariant, `index` is updated to `i` whenever the current element exceeds the element at position `index`. The variable `i` is then incremented until it reaches `N`. Finally, the loop variant, representing the number of elements remaining to process, also emerges naturally (line **variant**).

2.2 The Invariant Expressed Graphically (Graphical Loop Invariant)

FM thinking can be informal in the way it is applied, according to Morgan [25]. In a CS1 context, logical assertions can be replaced by a graphical and textual representation, as is done with GLIS. To best reflect the Formal Loop Invariant, the GLI must convey the same properties of the variables and data structures.



(a) Graphical Loop Invariant (GLI)

$$\begin{aligned} \texttt{a} &== \mathit{old}(\texttt{a}) \land \\ 0 \leq \texttt{i} &\leq \texttt{N} \land \\ 0 \leq \texttt{index} &< \texttt{N} \land \\ \forall k \cdot (0 \leq k &< \texttt{i} \\ \Rightarrow \texttt{a}[\texttt{index}] &\geq \texttt{a}[k]). \end{aligned}$$

(b) Formal Loop Invariant

Fig. 2. Loop Invariant for finding a maximal value in an array

Fig. 2a provides the GLI corresponding to the algorithm from Fig. 1. The array `a`, traversed by the loop, is depicted as a rectangle, whose upper portion hosts the indices. A vertical red line (the *dividing line*) splits the array into two zones to illustrate the program state after several iterations. The blue area (referred to as the *done*-zone) includes the elements that have already been compared with a current maximum. The position of the current maximum in this zone is stored in `index`. The green area contains the remaining elements to compare. Its first element, attached to the *dividing line*, is located using the iteration variable `i`. The black-bold lines act as boundaries for the *dividing line*, showing that `i` goes from `0` to `N`. In contrast to the Formal Loop Invariant (Fig. 2b), the graphical format avoids logical and mathematical notations, and is therefore much more accessible to students, particularly with colors highlighting its different components. GLIS are amenable to represent loops over a variety of data structures. For more details, we refer to our previous work [7].

2.3 Dafny

To verify the consistency between a Gli and its C implementation, we rely on Dafny[1]. In Dafny, specifications are defined alongside code. These are automatically converted into proof obligations verified by an SMT solver. Dafny is now recognized as an effective way for students to learn the correctness-by-construction approach. A previous study [26] showed that, when Dafny is used as the teaching language, many students can argue informally about code correctness, but struggle to formalize the properties and the proof. This motivates the use of another, semi-formal front end, as the Gli. More recent works [11,27] explored teaching software correctness using Dafny with LLMs. Namely, to improve Dafny's accessibility, Loop Invariants can be generated by LLMs [27]. Compared to our approach, this shifts the burden from formulating invariants to understanding LLM-generated predicates and crafting effective prompts. Our framework takes a different stance: rather than generating predicates from code, it derives them from the Gli, keeping the student at the center of the specification process.

3 Educational Context

Our CS1 course. It is the first programming course taken by CS students, who have no prior programming experience. The course is given in French and covers fundamental concepts (variables and types, conditionals, loops, arrays, functions, and pointers and memory management). C was chosen as the programming language for its broad applicability, unlike Dafny which is tailored to formal verification. Loop construction represents a third of the course schedule. It is where the Gli is introduced. To validate the use of Glis in CS1, we conducted a preliminary study in our classroom and demonstrated a positive impact of Gli-based programming on students' ability to solve algorithmic problems [5].

Our learning platform Café *2.0.* To practice Gli-based programming, we developed Café 2.0. It provides three homework assignments on Gli-based programming, each one taking about four hours to complete and involving up to four loops. For each loop, students form a Gli, derive the initial and final states, and implement the corresponding C code. Once submitted, Café 2.0 provides feedback on each of these four solution parts, like illustrated in green in Fig. 3. The consistency between the Gli and the code is also *partially* verified by comparing the starting position of the *dividing lines* with the initial variable values, and their ending position with the loop guard. Screenshots of the Gli in Café 2.0 and feedback students receive are illustrated in previous work [6]. In previous papers, we also studied students' problem-solving behavior in the platform [8] as well as their ability to auto-regulate from the automated feedback [9]. That study revealed a gap between homework and exam grades, suggesting that students may game the platform. Upon investigation, we found some code snippets passing Café 2.0's tests despite being semantically inconsistent with their Gli. Completing the consistency checks aims to address this. In the to-be version, if all Café 2.0's

[1] `https://github.com/dafny-lang/dafny`

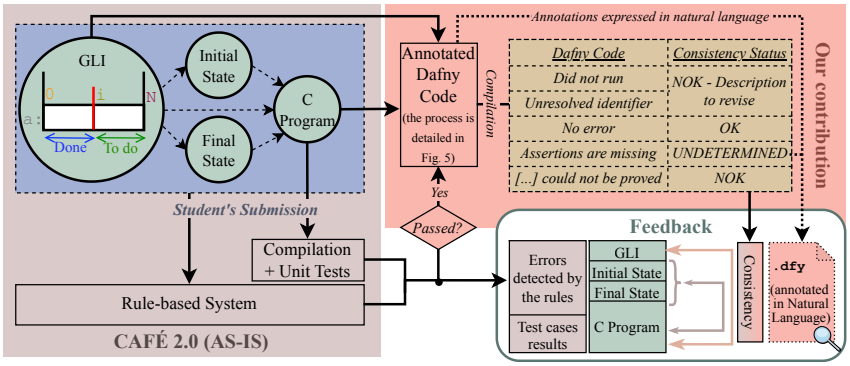
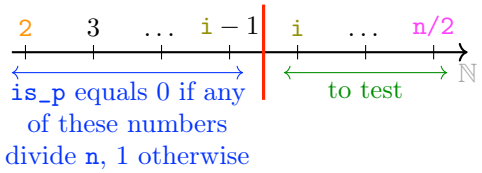


Fig. 3. Integration of our proof-of-concept prototype into our learning platform

checks have passed, our proof-of-concept prototype will complete the verification by inspecting if the GLI is restored after running the loop body, as shown in the red rectangle in Fig. 3. Without it, a student may receive a maximal score despite a loop body inconsistent with their GLI, masking some misunderstanding of the methodology. An example of such a solution, answering Problem 3 from our ten representative problems, is given in Fig. 4. While the GLI introduces `is_p` as the variable holding the intermediate result of the divisor search, the code only updates it after the loop. This is an example of a semantic gap between the GLI and the code that our prototype can detect, whereas CAFÉ 2.0 cannot. An example of consistent code snippet is given by `Examples_input/c_prog/3_prime.c`.




```
unsigned prime(unsigned n){
  unsigned i = 2;
  unsigned is_p=0;
  while(i<=n/2 && n%i!=0){
    i++;
  }
  is_p = (n!=1 && i>n/2);
  return is_p;
}
```

Fig. 4. Example of a solution to determine if `n` is a prime number (Problem 3). This solution scores maximally in CAFÉ 2.0's AS-IS version despite an inconsistency between the code and the GLI (due to `is_p` that is not initialized to 1).

To verify the consistency between a student's GLI and their C code, Dafny is used as a back-end verifier. As shown in yellow in Fig. 3, the Dafny output is translated into plain-language feedback so that no knowledge of formal methods is required from the students. The text may express that: (i) their GLI and C code are

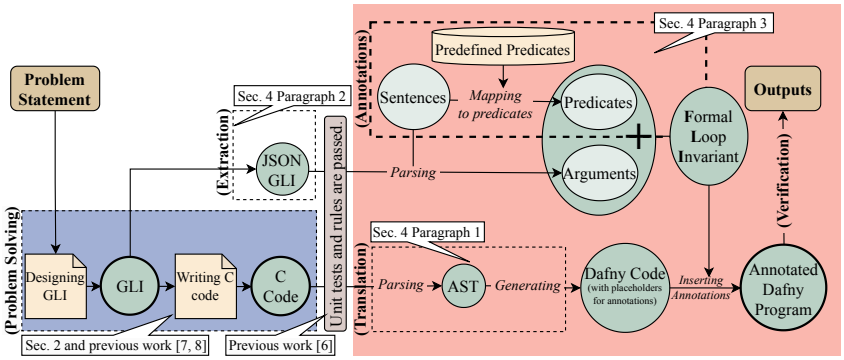


Fig. 5. The steps to verify the consistency between a GLI and a C code snippet

consistent, (*ii*) an inconsistency was detected and their text description (or whole solution) should be revised, or (*iii*) the consistency could not be determined[2]. To complement this feedback, students may inspect the resulting Dafny code themselves, annotated with natural language sentences instead of assertions to make it accessible (see examples in `Examples_output/dafnyNaturalLanguageAnnotation/`). In this way, they can evaluate the GLI-to-FLI translation and get used to code annotated with assertions, in preparation for the CS2 course.

4 From GLI and C Code to Dafny Framework

Fig. 5 provides an overview of the framework to verify the consistency between GLI(S) and a C program. Three main steps are followed.

Translating C Code to Dafny. Inspired by Sriya et al. [31], we implemented a prototype Python translator based on the `pycparser` library [3]. This library provides a C parser (ISO C99) that generates an Abstract Syntax Tree (AST) for analyzing and transforming code. It can also generate C code from the AST, which we adapted to produce Dafny code instead, handling their main syntactic differences. That is documented in `step1_translation/c_to_dafny_translator.py`.

Extracting Relevant Information from GLI. For any exercise, students build their GLIS using a graphical editor integrated into CAFÉ 2.0. Any GLI is saved as a `json` file storing its key components. They are colored in Fig. 2a. Examples of such files are given in the artefact (see `Examples_input/GLI_json/`) with their corresponding GLI (see `Examples_input/GLI_drawing/`). From such a file, all structural information needed for predicate generation — data structure names, bounds, iteration variables, and zone descriptions — is directly accessible.

[2] That may happen due to missing assertions in the generated Dafny code. Such a case is discussed in Section 5.4. It may also happen if the text description could not be matched to any predicate from the Database (see Section 5.1).

From GLI *Information to Dafny Annotations.* To automatically translate a GLI and its C implementation into Dafny, we defined a Dafny code template whose annotation placeholders are filled in with predicates derived from the GLI. Two nested loops are modeled with Pseudocode 1. This hierarchical structure has

```
   requires : Precondition
   ensures  : All Preserved zone properties
   ensures  : GLI_ex Done zone
   ensures  : ¬guard loop_ex
1  initialization_ex
2  while loop guard_ex do
       invariant : GLI_ex Domain of the iteration variable
       invariant : GLI_ex Preserved properties
       invariant : GLI_ex Done zone
3      ...
4      initialization_in
5      while loop guard_in do
           invariant : GLI_in Domain of the iteration variable
           invariant : GLI_in Preserved properties
           invariant : GLI_in Done zone
6          ...
7      end_in
8      ...
9  end_ex
```

Pseudocode 1: Skeleton of two nested loops to insert annotations

two implications: (*i*) the postcondition of `GLI_in` contributes to the verification of `GLI_ex`, and (*ii*) the outer loop invariant must imply the precondition of `GLI_in`.

Once each GLI sentence is matched to a predicate (see the predicates related to our CS1 course context in `Predefined_predicates/predicates.dfy`), annotations are inserted automatically. The arguments of each predicate are drawn from the GLI variables (iteration variable(s), accumulator variable(s) etc.). Predicate bodies are also predefined and verify the property indicated in the name. The suffix "`UpTo`" denotes a property within a *done*-zone (representing a portion of the whole object being iterated over). For example:

```
predicate IsMaxPosUpTo(a: seq<int>, size: nat, lb: nat, ub: nat,
                       i: nat, variable: nat)
  requires size =|a|
  requires lb ≤ub < size
  requires lb ≤i ≤size
{
  lb ≤variable < size ∧∀ k • lb ≤k < i ⟹a[variable] ≥a[k]
}
```

Filling in the template thus requires *autoformalizing*[3] the Gli text descriptions. Each sentence is matched to one of the predefined predicates. For this selection, each predicate is characterized by several text descriptions (see `Predefined_predicates/__init__.py`). To perform sentence matching, the semantics of the sentences must be extracted. Among the available techniques (syntax analysis, pattern matching, and statistical methods), we adopt the latter, relying on Sentence-BERT [29], a transformer-based model producing sentence embeddings particularly effective for semantic similarity.[4] We use the Python module SentenceTransformers [29], which provides over 10,000 pre-trained models via Hugging Face. Pavlyshenko and Stasiuk [28] compare these models in terms of performance and efficiency. We selected ours based on their findings. These models are free, lightweight enough for local deployment,[5] and accurate enough to support our framework, as demonstrated in Section 5.1.

Fig. 6 illustrates how these models bridge natural language descriptions and formal predicates. The horizontal pyramid frames the different representations of two properties, from a more flexible syntax (natural language) to a more constrained one (predicates). As an intermediate representation, the Sentence-BERT model produces a vector of values carrying the semantics of the input sentences (σ_A and σ_B). Then, the `PredicateMatcher` compares these two embeddings with all embeddings of the predicates' descriptions (represented in the cache in the figure). In this example, 50 predicates are defined with 10 corresponding text descriptions. Therefore, for each input sentence, up to 500 comparisons are performed. After computing the cosine similarity of each pair of embeddings (σ_Y, σ_{IJ}), their scores are compared. For each input sentence, the predicate P_I, corresponding to the description S_{IJ} with the highest score, is returned. To mitigate the risk of false positives (where all scores are low, including the highest one), we introduced a threshold below which no match is returned (see the red cross in Fig. 6). Section 5.1 discusses the value of this threshold.

5 Evaluation

5.1 Model and Parameter Selection for Predicate Mapping

Before applying our framework, we calibrated four parameters: the sentence embedding model, the number of descriptions per predicate, the predicate selection function (`max()` or `median()`), and the similarity score threshold.

Dataset. First, we identified all the properties involved in the problem statements in the context of our CS1 course from 2013 to 2024, leading to 50 predicates

[3] *Autoformalization* is defined as the process of automatically translating natural language (based on informal representations) into formal specifications or proofs [32].

[4] `https://www.marqo.ai/course/introduction-to-sentence-transformers`

[5] To map two sentences using `sentence-transformers/all-MiniLM-L12-v2`, the running time to embed the $2 + 500$ sentences and perform 2×500 comparisons is less than 30 seconds on a regular laptop.

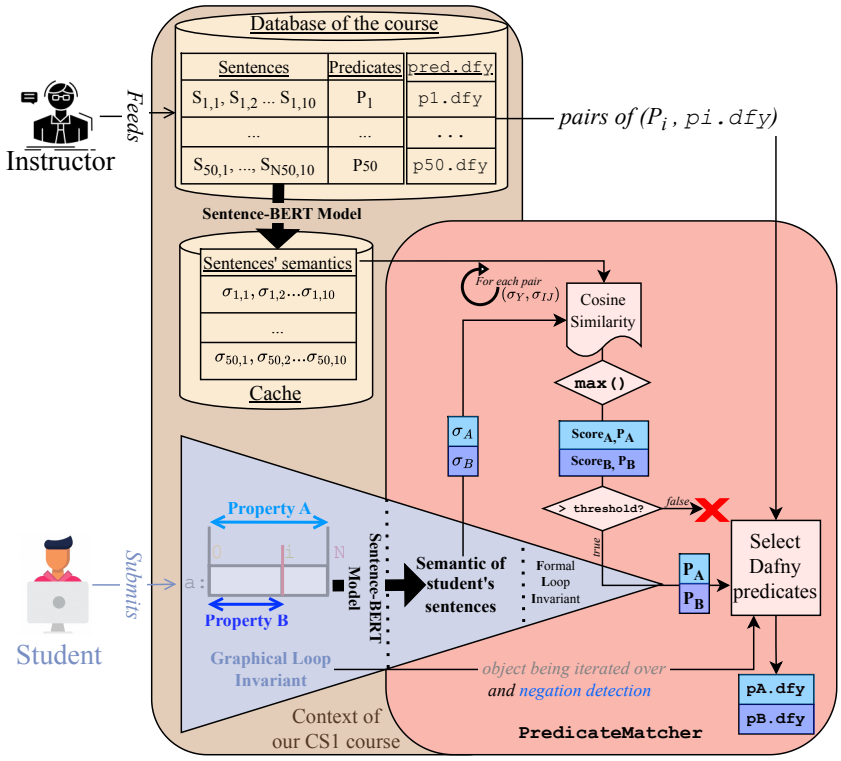


Fig. 6. Mapping between sentences from the GLI and predicates

in total. Next, an instructor produced 10 English descriptions per predicate, covering various formulations of each property. To tune our mapping, we built a validation set from student midterm and exam sentences produced over the past five years. Since our students are not expected to be fluent in English, they write in French. Then, these sentences were translated using Google-translate, prior to evaluation. From 600 analyzed sentences, we removed duplicates and near-duplicates and got 400 sentences. Then, we randomly retained 40 sentences expressing different properties. This set includes 19 sentences that should map to a predefined predicate, 6 inaccurate ones (e.g., missing accumulator references), and 15 that are meaningless or describe properties absent from our database.

Parameter selection. Through multiple simulations, we selected the `sentence-transformers/all-MiniLM-L12-v2` model with 6 descriptions per predicate and a maximum-based predicate selection (median-based selection performed slightly worse). We set the similarity threshold to 0.77, maximizing the number of true positives and true negatives, as shown in Fig. 7.

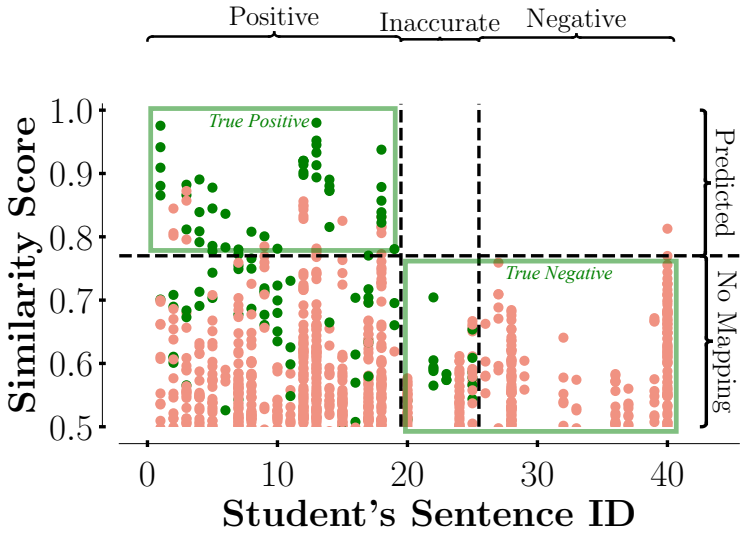


Fig. 7. Performance of the model mapping sentences to predicates. Scores range from 0 to 1, but the figure only shows scores exceeding 0.5, as lower scores are not relevant for predicate prediction. Positive sentences are expected to be mapped to a predicate while inaccurate and negative ones should not.

Validation set results. Fig. 7 shows the similarity scores between each students' sentence and each predicate description. The 40 vertical strips correspond to the 40 students' sentences. Within each strip, the green dots highlight scores with the expected predicate's descriptions. A correct prediction requires the highest green dot to exceed the threshold. Of the 19 positive sentences, 16 are correctly classified. The misclassified ones fall into two patterns:

- **Semantically close predicates.** When two predicates describe the same property applied to different element types, or a property and its negation, misclassification can occur. This motivates maintaining one predicate instance per property, and dispatch to the correct Dafny predicate based on structural information (e.g., detecting words like "digits" or "array elements") and negation detection from the sentence. This is illustrated in Fig. 6 by the arrow connecting the Gli and the final selection of Dafny predicates.
- **Insufficient coverage of student phrasing.** Some sentences score low against all descriptions, suggesting the database should be enriched with actual sentences students wrote to better align with natural phrasing. Additionally, translating sentences into English likely increased that gap.

No one of the 6 inaccurate sentences led to a score above the threshold, suggesting that students will have to be accurate enough to get their sentence mapped. Finally, among the 15 negative sentences, one exceeds the threshold (standing as a false positive), confirming that the system correctly handles the large majority of cases on both positive and negative sentences.

Test set performance. After consolidating predicates as recommended, we evaluated the mapper on the remaining 360 students' sentences not used during validation. The 67 positive sentences among them served as the test set. 62 were correctly mapped, yielding an accuracy of 92.5%. Negative sentences were not included in the test set, as the focus here is on predicate recall.

Limitations. The predicate-mapping evaluation relies on a limited amount of sentences (400), due to the constraint of collecting hand-written students' sentences. Additionally, the tuning described above was performed on translated sentences, which may have increased the gap between the descriptions from the database and actual student's sentences, and decrease the evaluated performance of the mapper. More broadly, both the predicate database and the parameter calibration are context-dependent anyway. The instructors adopting this framework in a different course or language setting should rebuild the database descriptions and re-run the tuning process on a sample representative of their own context.

5.2 Validating our Proof-of-concept Prototype

Scope of problems. To validate our framework presented in Section 4, we tested it on ten problems representative of our CS1 curriculum:[6]

1. Calculate the sum of the first `(n+1)` natural numbers.
2. Calculate the product of naturals in the interval `[a,b]`.
3. Check whether a number `n` is prime.
4. Count the number of occurrences of a digit `x` for each number in `[a,b]`.
5. **Determine the index of a maximum value of an array `a` of size `N`.**
6. Sort an array `tab` of size `N` by implementing the selection sort algorithm.
7. Check whether all the elements from two given arrays `a` (of size `N_a`) and `b` (of size `N_b`) are present in the array `c` (of size `N_c`).
8. Reverse the digits of a number `n`.
9. Count the number of odd digits in a natural number `n`.
10. Draw an isosceles triangle of height `h` in the terminal.

These problems cover the loop patterns taught in our course: accumulator across a number sequence, digit manipulation, array processing, and nested loops. They also align with the canonical iteration examples identified by Scapin and Mirolo [30].

Possible Outputs and tested solutions. Since Café 2.0 has already validated the Gli and the C code prior to invoking our framework (see Fig. 3), the Dafny verifier can produce four possible outcomes:

1. **Verified (no error).** It confirms that the Gli and the C code are consistent with each other. In addition to the unit tests already passed in Café 2.0, it strengthens both correctness and consistency.

[6] They are classified and described in `Examples_input/statements.pdf`

2. **Missing assertions (error).** Assertions are missing in the loop body for verification. This occurred for Problem 7, as discussed in Section 5.4.
3. **Unresolved identifier (error).** It occurs if the system attempts to map the GLI variables with arguments from an incorrect predicate.
4. **Invariant could not be proved (error).** It indicates the loop body does not always make true the GLI at each evaluation of the loop guard.

Additionally, if a text description cannot be mapped to any predicate in the database, the system reports this failure and no Dafny code is executed.

For each problem, the instructor composed one correct solution (a GLI and C code that are correct and mutually consistent). In addition, for each problem, correct (based on CAFÉ 2.0's primary checks) but inconsistent solutions were produced to verify that the system triggers errors as expected. These solutions are referenced with the suffix "inconsist" in the names of the files in `Examples_input/c_prog/` and `Examples_input/GLI_json/`. From Problem 1 to 6, an inconsistency was introduced in the code while for the four last problems, the inconsistency has been incorporated in the text descriptions in the GLI.

All ten consistent solutions were successfully verified by Dafny, while all inconsistent ones triggered the expected errors. Specific limitations encountered for individual problems (such as C instructions that are not explicit enough to generate all required loop body assertions) are discussed in Section 5.4.

5.3 Application to the Problem of Finding a Maximum in an Array

In this section, we describe the framework for Problem 5, with a consistent solution (see the GLI in Fig. 2a and C code in the artefact (`Examples_input/c_prog/5_max.c`)).

Once our framework receives the solution, it translates the C code into Dafny. This step is labeled "Translation" in Fig. 5. The `include` commands import the formal predicates that were defined beforehand. This intermediate result is `Examples_output/dafnyNotAnnotated/5_max.dfy`.

Next, the framework extracts relevant information from the GLI and derives annotations. These steps are labeled "Extraction" and "Annotations" in Fig. 5. This information corresponds to the object being iterated over, the lower and upper bounds, the iteration variable(s), and the sentences describing the zones. In this example, the sentences are *"index contains the position of a maximal value"* and *"not modified"*. After comparing them to all the descriptions of the predefined predicates, we obtained their highest similarity scores (0.8994 and 0.7913, respectively), associated with the expected predicates ("`IsMaxPosUpTo`" and "`IsUnchanged`"). The predicate `IsMaxPosUpTo` takes as arguments the name of the array `a`, its size `N`, the iteration variable `i` and its (intermediate) result `index`. The first three arguments are extracted from specific locations in the GLI (respectively colored in grey, purple and green in Fig. 2a), while `index` is retrieved from the sentence describing the *done*-zone. Next, `IsUnchanged` simply takes the last and the initial version of `a`. The resulting formal loop invariant, made up of these two predicates, is stored as `Examples_output/GLIwithIdentifiedPredicates/5_max.json`.

Once the formal loop invariant(s) have been formed, they are inserted into the code template (Pseudocode 1) at the appropriate locations. The resulting Dafny code is as follows:

```
include "predicates.dfy"
include "helpers.dfy"

method findMax(a: array<int>, N: nat) returns (index: nat)
requires IsArraySize(a [..], N)
requires IsNoEmptyArray(a[..], N)
ensures IsUnchanged(a[..], old(a [..]) )
ensures IsMaxPosUpTo(a[..], N, 0, N−1, N, index)
{
var i: int := 0;
index := 0;
while (i < N)
 invariant IsWithinBounds(N, 0, N−1, i)
 invariant IsUnchanged(a[..], old(a [..]) )
 invariant IsMaxPosUpTo(a[..], N, 0, N−1, i, index)
{
 if a[i] > a[index] {
   index := i;
 }
 i := i + 1;
}
}
```

Finally, as indicated in Fig. 5 as the "Verification" step, the framework runs and verifies the Dafny code. In this case, it runs without any error.

5.4 Discussion and Limitations

Discussion. Among the ten consistent solutions to each of the representative problems, two notable cases required specific attention.

First, for Problem 4 and its consistent solution, which relies on two nested loops, we had to assist Dafny in connecting the postcondition of the inner loop with the invariant of the outer loop. To do so, we introduced a lemma (called `IsLocalCount`).

Second, regarding Problem 7, we simplified its original formulation to allow Dafny to verify the code without requiring additional assertions. The original version of the problem consisted in computing the intersection of arrays `a` and `b`. In our simplified version, the goal is instead to check whether all elements of `c` are present in both `a` and `b`, without verifying that *all* common elements of `a` and `b` appear in `c`. For more complex programs such as this one, it is worth noting that Dafny presents a trade-off between introducing additional variables and adding more annotations to clearly relate variable states. In the context of automatic annotation generation, lemmas can be defined in advance with relative ease to bridge annotations. However, automatically generating the `assert()` statements

sometimes needed to complement the invariants remains a harder problem. This leads us to introduce the notion of *verification-friendliness*: for our framework to verify a program automatically, each loop guard must involve a single iteration variable, and intermediate results must be made explicit through dedicated variables rather than inferred from the final state of multiple indices. Code must therefore be written so as to satisfy this *verification-friendliness* property. For instance, we consider a first version that uses a single outer loop with condition `i<N_a && j<N_b && k<N_c`, inside which two inner loops respectively search for `c[k]` in `a` and `b`. The iteration variable `k` is incremented only when both searches succeed. This version cannot be automatically verified by Dafny based solely on the generated annotations. The difficulty stems from the fact that the loop guard conflates three independent indices, and the search result is expressed implicitly through the final values of `i` and `j`, leaving Dafny unable to establish the necessary connections between loop states. The second version, available at `Examples_input/c_prog/7_intersect.c`, restructures the code by introducing two explicit auxiliary variables: `is_intersection`, initialized to `1`, and `is_present`, reset to `0` before each inner search. The outer loop iterates over `k` only while `is_intersection` remains `1`. Each inner loop sets `is_present` to `1` upon finding a match, and `is_intersection` is set to `0` immediately after each search if no match was found. This version can be automatically verified by Dafny, as the state of each search is made explicit through dedicated boolean variables, giving the verifier clear handles on which to attach invariants.

Limitations. A primary limitation of this study is the lack of student data. As a result, the instructor drew on their experience with student solutions to produce representative examples that are currently gaming Café 2.0 and evaluate the end-to-end flow on them. Since these examples now trigger an error, we can be reasonably confident that the TO-BE version of Café 2.0 no longer allows students to obtain a maximal grade with incomplete or inaccurate descriptions. However, since the framework is ultimately intended to process student submissions, its robustness to the variability of real student code needs to be tested in future work. More generally, our evaluation was restricted to a limited subset of C solutions to ten representative exercises drawn from our CS1 course. To advance beyond the proof-of-concept stage, the framework should be evaluated on a broader range of problems and student solutions. In particular, the *missing assertion* Dafny error, that was occurring for the first version of correct solution for Problem 7, may occur in other cases. To manage these edge cases, we should identify them by mapping them to a pattern and automatically produce additional assertions. That update may also come from the Dafny's maintainer team.

Next, our framework relies on a sentence-to-predicate mapper, which implies that predicates must be defined beforehand. LLMs could overcome this limitation by generating predicates from scratch based on natural language sentences [14,18]. Beyond the scope of formalizing properties and translating them into a formal representation, LLMs can also generate loop invariants directly from program code [27]. However, the lack of rigorous logical reasoning in LLMs combined with the absence of frame (i.e., nothing is predefined) makes them a delicate tool to

rely on in the context of FM [32]. Dantas, Cordeiro, Nowroozi and Norbert [15] recently led a survey on LLM's reliability and pointed out the natural-language-to-formal translation as its primary limitation. Additionally, LLMs carry constraints such as integration, data privacy etc. Finally, our framework currently requires variable names to be explicitly marked with curly brackets within sentences for extraction. This constraint could be relaxed by adopting Named Entity Recognition. Furthermore, when a sentence contains multiple variables, the system may fail to map each one to the correct argument of the corresponding predicate. Both limitations could ultimately be addressed by employing a more capable LLM, able to identify variable names automatically and infer their roles without explicit markup.

6 Conclusion

In this paper, we propose a proof-of-concept framework to bridge the gap between informal and formal loop invariants. Given C code coupled with Graphical Loop Invariants (Glis), our framework automatically translates them into annotated Dafny code. We evaluate the suitability of this framework using ten representative problems from our CS1 course. All ten consistent solutions were verified by Dafny without errors, while all inconsistent ones triggered at least one verification error.

To derive annotations from a Gli, we leverage its structured format to extract the variables and sentences it contains. This semi-formal diagram follows established patterns whose structure encodes semantic information, including array names, iteration variable(s), and the sentences describing what has been achieved by previous iterations. Our framework extracts them. It then matches the sentences to predefined predicates using similarity scores between the sentences and predicate descriptions in the database. Finally, the predicate arguments are mapped to the extracted variables. The sentence-to-predicate matcher achieved 92.5% accuracy on our test set, demonstrating the feasibility of the approach.

In the future, we plan to fully integrate this framework into Café 2.0, deploying it on a broader range of problems and, crucially, on students' actual solutions to provide automated feedback on loop body consistency. More broadly, our work provides a blueprint for bridging informal, diagram-based program reasoning and formal verification, applicable beyond our specific pedagogical context. LLMs could further extend this approach by generating predicates directly from natural language, though human validation would remain necessary to ensure their formal correctness.

Acknowledgments. This work is supported by the CyberExcellence project funded by the Walloon Region, under number 2110186.

Disclosure of Interests. The authors declare no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

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: Basic approach and teaching experiences. Formal Aspects of Computing **21**(3), 227–244 (May 2009). https://doi.org/10.1007/s00165-008-0070-y
3. Bendersky, E.: pycparser: Complete c99 parser in pure python. GitHub (March 2024), `https://github.com/eliben/pycparser`, [Last Accessed: 2026-09-10]
4. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74113-8
5. Brieven, G., Donnet, B.: Assessing graphical loop invariant based programming performance in a CS1 course. Informatics in Education pp. 1–38 (2026). https://doi.org/10.15388/infedu.2507.010
6. Brieven, G., Labrahimi Kasdaoui, A., Donnet, B.: Café 2.0, an automated feedback tool to approach formal methods. cs.SE 2607.03001, arXiv (July 2026). https://doi.org/10.48550/arXiv.2607.03001
7. Brieven, G., Liénardy, S., Malcev, L., Donnet, B.: Graphical loop invariant based programming. In: Proc. Formal Methods Teaching (FMTea) (March 2023). https://doi.org/10.1007/978-3-031-27534-0
8. Brieven, G., Malcev, L., Donnet, B.: Practicing abstraction skills through diagrammatic reasoning over Café 2.0. In: Proc. IEEE Global Engineering Education Conference (EDUCON) (May 2024). https://doi.org/10.1109/EDUCON60312.2024.10578665
9. Brieven, G., Malcev, L., Donnet, B.: How to automate feedback on diagrammatic reasoning with a relevant degree of freedom? In: Proc. IEEE Global Engineering Education Conference (EDUCON) (April 2025). https://doi.org/10.1109/EDUCON62633.2025.11016495
10. Broy, M., Brucker, A.D., Fantechi, A., Gleirscher, M., Havelund, K., Kuppe, M.A., Mendes, A., Platzer, A., Ringert, J.O., Sullivan, A.: Does every computer scientist need to know formal methods? Format Aspect of Computing **37**(1), 1–17 (December 2024). https://doi.org/10.1145/3670795
11. Carreira, C., Silva, A., Abreu, A., Mendes, A.: Can large language models help students prove software correctness? an experimental study with dafny. In: Proc. Software Engineering and Formal Methods (SEFM) (November 2025). https://doi.org/10.1007/978-3-032-10444-1_13
12. Cherenkova, Y.A., Zingaro, D., Petersen, A.: Identifying challenging CS1 concepts in a large problem dataset. In: Proc. ACM Technical Symposium on Computer Science Education (SIGCSE) (March 2014). https://doi.org/10.1145/2538862.2538966
13. Cormen, T.H., Leiserson, C. E .and Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT press (2009)
14. Cosler, M., Hahn, C., Mendoza, D., Schmitt, F., Trippel, C.: nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In: Enea, C., Lal, A. (eds.) Proc. Computer Aided Verification (July 2023). https://doi.org/10.1007/978-3-031-37703-7_18
15. Dantas, P., Cordeiro, L., Nowroozi, E., Norbert, T.: Toward safe LLM agents: A survey of specification, verification, and enforcement. cs.AI 2608.14590, arXiv (July 2026). https://doi.org/10.48550/arXiv.2608.14590
16. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Inc. (1976)

17. Dongol, B., Dubois, C., Hallerstede, S., Hehner, E., Morgan, C., Muller, P., Ribeiro, L., Silva, A., Smith, G., de Vink, E.: On formal methods thinking in computer science education. Formal Aspects of Computing **37**(1), 1–23 (December 2024). https://doi.org/10.1145/3670419
18. Felix, V., Till, M., Björn, G.: Advancing natural language formalization to first order logic with fine-tuned LLMs. cs.CL 2509.22338, arXiv (September 2025). https://doi.org/10.48550/arXiv.2509.22338
19. Fowler, M., Kraemer, E., Sitaraman, M., Hollingsworth, J.E.: Tool-aided loop invariant development: Insights into student conceptions and difficulties. In: Proc. ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE) (June/July 2021). https://doi.org/10.1145/3430665.3456351
20. Gries, D.: The Science of Programming. Springer (1987). https://doi.org/10.1007/978-1-4612-5983-1
21. Mannila, L.: Invariant based programming in education – an analysis of student difficulties. Informatics in Education **9**(1), 115–132 (April 2010). https://doi.org/10.15388/infedu.2010.07
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). https://doi.org/10.1109/TE.2018.2864133
23. Meyer, B.: A basis for the constructive approach to programming. In: IFIP World Computer Congress. pp. 293–298 (1980)
24. Morgan, C.: Programming from Specifications. Prentice-Hall (1990)
25. Morgan, C.: (in-)formal methods: The lost art. In: Proc. Engineering Trustworthy Software Systems (September 2014). https://doi.org/10.1007/978-3-319-29628-9_1
26. Noble, J., Streader, D., Gariano, I.O., Samarakoon, M.: More programming than programming: Teaching formal methods in a software engineering programme. In: Proc. NASA Formal Methods (May 2022). https://doi.org/10.1007/978-3-031-06773-0_23
27. Pascoal Faria, J., Trigo, E., Abreu, R.: Automatic generation of loop invariants in dafny with large language models. In: Proc. Fundamentals of Software Engineering (April 2025). https://doi.org/10.1007/978-3-031-87054-5_10
28. Pavlyshenko, B., Stasiuk, M.: Semantic similarity analysis using transformer-based sentence embeddings. Electronics and Information Technologies **30**, 43–58 (August 2025). https://doi.org/10.30970/eli.30.4
29. Reimers, N., Gurevych, I.: Sentence-BERT: Sentence embeddings using siamese BERT-networks. In: Proc. Conference on Empirical Methods in Natural Language Processing (EMNLP) (November 2019). https://doi.org/10.18653/v1/D19-1410
30. Scapin, E., Mirolo, C.: An exploration of teachers' perspective about the learning of iteration-control constructs. In: Proc. Informatics in Schools. New Ideas in School Informatics (2019). https://doi.org/10.1007/978-3-030-33759-9_2
31. Sriya, S., Lavanya, L., Manohar, A.M., Kumar, N.S.: Verification of C programs using annotations. In: Proc. IEEE International Conference on Technology for Education (T4E) (December 2019)
32. Yedi, Z., et al.: The fusion of large language models and formal methods for trustworthy AI agents: A roadmap. cs.AI 2412.06512, arXiv (December 2024). https://doi.org/10.48550/arXiv.2412.06512
33. Zhumagambetov, R.: Teaching formal methods in academia: A systematic literature review. In: Proc. Formal Methods – Fun for Everybody (December 2021). https://doi.org/10.1007/978-3-030-71374-4_12