ESPECIFICA++: A DSL for Developing the Competency of Formal Specification in Computer Programming Education

José-Luis Sierra-Rodríguez
Dpto. Ingeniería del Sw. e Inteligencia Artificial
Universidad Complutense de Madrid
Madrid, Spain
jlsierra@ucm.es

Mercedes Gómez-Albarrán
Dpto. Ingeniería del Sw. e Inteligencia Artificial
Universidad Complutense de Madrid
Madrid, Spain
mgomeza@ucm.es

Ana-María González-de-Miguel
Dpto. Ingeniería del Sw. e Inteligencia Artificial
Universidad Complutense de Madrid
Madrid, Spain
agonzale@ucm.es

Marta López-Fernández
Dpto. Ingeniería del Sw. e Inteligencia Artificial
Universidad Complutense de Madrid
Madrid, Spain
mlopezfe@ucm.es

Antonio Sarasa-Cabezuelo
Dpto. Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Madrid, Spain
asarasa@ucm.es

***Abstract*—Competence in Formal specification is a key skill with which Computing Engineering students often face significant challenges. In this paper, we describe ESPECIFICA++, a domain-specific language (DSL) embedded in the programming language C++ that allows for the description of executable specifications, resulting in a useful tool for fostering the development of formal specification skills among students. Using ESPECIFICA++, students can describe aspects such as preconditions, postconditions, and algorithm invariants. These descriptions then become executable objects that can be validated against sets of test cases, as if they were programs. The paper outlines the main features of ESPECIFICA++, illustrates its use with a representative case study, and conducts a preliminary evaluation of its applicability and usefulness based on a SWOT analysis.**

Keywords—Computer Programming Education, Formal Methods Education, Algorithmics, Program Specification, DSL

I. INTRODUCTION

Programming is a core subject in many university studies in Computing, Software, and Computer Engineering [1], as well as in other engineering disciplines, where programming is a valuable problem-solving tool. Among all the specific competencies that students in this subject must acquire, the most important is the ability to develop error-free programs that correctly solve the problems for which they were designed. Therefore, in our opinion, and that of others [2], computer engineering students must have at least a basic competence in formal specification. Indeed, the first step in developing a correct program is to specify the problem the program must solve. Specification is, in fact, a central aspect covered in many Computing, Software, and Computer Engineering curricula, where it is typically introduced in introductory algorithm courses. In these courses, students must acquire, among other skills, proficiency in the formal specification of problems, usually using a language based on predicate logic [3] [4]. They must also be able to justify that their solutions meet these specifications.

This work has been supported by the UCM's INNOVA 2024-2025 program (project nº 152), as well as by the Spanish AEI (research project PID2021-123048NB-I00).

However, our experience teaching introductory algorithm courses has shown that students face significant difficulties in acquiring basic competencies related to these aspects (this concern has also been observed by others, e.g., [5]). Students frequently misuse formal language, often creating specifications that are nonsensical and disconnected from the problem to be solved. Since, as we've mentioned, creating a good specification is the foundation in the algorithm development process, the rest of the *building* that students are supposed to construct collapses. As a result, students move directly to coding in a *trial-and-error* process, which ultimately leads to programs that, after being subjected to an exhaustive battery of tests, reveal serious flaws. Worse still, these programs are undocumented, unjustified, and very difficult to understand and maintain.

To mitigate this situation, we are currently working on an educational strategy focused on *executable specifications*. Our proposal is for students to be able to construct executable specifications that can be validated using the same sets of test cases they use to validate their code. As a result, students will stop perceiving specifications as *passive documentation* and will begin to see them as *active objects* on the same level as programs.

To implement this strategy, we have started by defining a domain-specific language (DSL) [6] for algorithm specification, based on a subset of the predicate logic formalism that we currently use in our introductory algorithm courses. This DSL, which we have named ESPECIFICA++ (an acronym for E*xecutable* SPECIFICA*tions in C*++), has been implemented as an embedded language in C++ (the primary language used in our courses). Utterances in ESPECIFICA++ are *executable*. It means that, using ESPECIFICA++, students can *build* predicates and other specification-related expressions as executable artifacts. With this, our goal is for students to change their perception of specifications, viewing them as fully equivalent to programs, as objects that can be executed, experimented with, and validated using meaningful test cases.

This paper describes the current version of ESPECIFICA++, illustrates its use with a case study, and provides an initial qualitative evaluation. The structure of the

rest of the paper is as follows. In Section II, we describe the main features of ESPECIFICA++ and provide some details of its implementation as an embedded DSL in C++. Section III illustrates the use of ESPECIFICA++ in a case study based on a prototypical example that we use in our classes on iterative algorithm design. Section IV presents our initial evaluation of the language. Section V presents some related work. Finally, Section VI presents some conclusions and future work directions.

II. THE ESPECIFICA++ DSL

As we mentioned earlier, ESPECIFICA++ is an *embedded language* [7] in C++. This means that statements in ESPECIFICA++ are described as valid C++ expressions, which can be executed like any other type of expressions. Below, we describe the different types of concepts and constructs supported by ESPECIFICA++, along with their purpose and informal meaning (subsections II.A, II.B, II.C, II.D, II.E, and II.F). Subsection II.G provides some details about the language implementation.

A. *Predicates and Expressions. Primitives* `pred` *and* `exp`.

The ultimate purpose of ESPECIFICA++ is to define predicates and other types of expressions. A predicate corresponds to a *boolean* expression that typically involves the values of the variables of a program. In contrast, the idea of an expression is analogous, although its type is not restricted to being *boolean* (it can be any type supported by C++).

The most basic way to create predicates and expressions in ESPECIFICA++ is through the primitives `pred` and `exp`. The argument for these primitives is, in the first case, a *boolean* C++ expression, and an arbitrary expression in the second case. The predicates and expressions constructed this way can be *executed*, just like the rest of the predicates and expressions in ESPECIFICA++. To do this, simply use them as if invoking a function without arguments (i.e., append the invocation operator `()`).

The result of executing the predicates and expressions constructed by `pred` and `exp` is the result of evaluating the underlying C++ expression in the context in which the primitive was applied. Therefore, if `pred` and `exp` are applied to expressions with variables, their execution will consider the value of those variables *at the time they were applied*. If the variables change later, these changes will not be reflected in the constructed predicates and expressions. Thus, a suitable way to think of `pred` and `exp` is as *photo cameras*, taking *snapshots* of the state of the underlying program.

Fig. 1 illustrates the use of `pred` and `exp`. Note that, although the program variables change, the values of the captured predicates and expressions do not change. Fig. 1 also illustrates the fact that, since ESPECIFICA++ is an embedded language in C++, ESPECIFICA++ constructs are also valid C++ constructs. Therefore, they can be interspersed with C++ code, as long as the underlying type system is observed.

Code	Output

`int b = 100;`
`int h = 30;`
`auto e = exp(b * h);`
`auto p = pred(b > h);`
`cout << boolalpha;`
`cout << e() << " "`
`  << p() << endl;`
`b = 5;`
`h = 6;`
`cout << e() << " "`
`  << p() << endl;`
	`3000 true`
	`3000 true`

Fig. 1. Examples of the use of the primitives `pred` and `exp` (the ESPECIFICA++ constructs are highlighted).

B. *Combining Predicates and Expressions*

ESPECIFICA++ allows the construction of more complex predicates from simpler ones using logical connectives. The logical AND is represented by the binary infix operator `&&`, the logical OR by the binary infix operator `||`, negation by the unary prefix operator `!`, implication by the binary infix operator `>>`, and equivalence by the construct `equiv` (it is used in functional mode, i.e., `equiv(`p_1`,` p_2`)`).

Similarly, ESPECIFICA++ introduces the comparison operators `==` (equality), `!=` (inequality), `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to) to construct predicates involving the comparison of expressions (the types returned by the compared expressions must be comparable with each other).

Finally, ESPECIFICA++ introduces basic arithmetic operators for combining expressions: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `%` (modulus). The types of the involved expressions must be compatible with each other.

Fig. 2 illustrates the use of some of these operators for combining expressions and predicates. Note that, although in this case the same effect could have been achieved at the level of C++ expressions, the utility of the introduced operators will become apparent when quantifier-based mechanisms for creating predicates and expressions are introduced.

C. *Quantifications*

ESPECIFICA++ allows the creation of new predicates and expressions using *quantification* (i.e., constructs that evaluate simpler predicates and expressions over different values and appropriately combine the obtained results). The first step is to introduce *quantified variables* that will range over the values referred to in the quantifications.

ESPECIFICA++ requires the explicit declaration of quantified variables as a prerequisite to formulating quantifications. This is done using the `var` primitive: `var(`T`,`v`)` declares a quantified variable v of type T. These variables can then be used in predicates and expressions. To access their value, the prefix operator * is used. Unlike regular C++ variables, ESPECIFICA++ predicates and expressions are sensitive to changes in quantified variable values (i.e., if a predicate or expression uses a quantified variable, and that variable value changes, the result of the predicate or expression may vary accordingly).

Code	Output

`int b = 100;`
`int h = 30;`
`auto e = exp(b * h) +`
`  exp(2 * b);`
`auto p = pred(b > h)&&`
`  (pred(b == 0)||`
`  pred(h == 0));`
`cout << boolalpha;`
`cout << e() << " " <<`
`  p() << endl;`
	`3200 false`

Fig. 2. Examples of predicate and expression combinations in ESPECIFICA++.

All the quantifications Q follow a common format:

$$Q\text{<}typeSpecs\text{>}(qvars).\text{range}(inf,sup)(.\text{filter}(F))?.ev(V) \quad (1)$$

where (…)? represents *optionality*. Constructs of this type build a predicate or an expression (depending on the type of quantification Q) that:

- Generates all possible value assignments within the quantification range to the quantified variables. The quantification range is indicated in the `range` clause and is determined by a lower bound *inf* and an upper bound *sup*. These bounds can be: (i) C++ expressions; (ii) quantified variables; or (iii) ESPECIFICA++ expressions.
- If a predicate F is specified as the value for the optional `filter` clause, it discards all assignments for which F evaluates to *false*.
- For each non-discarded assignment, it evaluates the predicate or expression V indicated as the value of the *ev* clause (the specific form of this clause depends on the quantification).
- Finally, it combines the results of the evaluations to produce the result of the quantified predicate or expression.

In the application of quantification, *typeSpecs* explicitly defines the type of the values in the quantification range and, if necessary, the type of the values returned by V. Additionally, it should be noted that all the expressions and predicates involved are constructed when the quantification is applied. Nevertheless, the proper use of quantified variables in F or V will allow them to evaluate to different values for different assignments to quantified variables. Likewise, the quantification range can be modulated for different evaluations of the produced predicate or expression simply by using quantified variables taking values between *inf* and *sup*.

D. Quantified Predicates

ESPECIFICA++ introduces two types of quantifications for constructing predicates. In both, `yield` is used as the *ev* clause, and only the type of the values in the quantification range is specified in *typeSpecs*. The quantifications themselves are:

- *Universal Quantification*. The `forall` quantifier is used as Q in (1). The result of the constructed predicate is *true* if all evaluations of V for the different value assignments to quantified variables are *true*, and *false* otherwise. Thus, if there are no valid assignments, the result will always be *true*.
- *Existential Quantification*. The `exists` quantifier is used as Q in (1). The result of the constructed predicate is *true* if any of the evaluations of V for the different value assignments to quantified variables is *true*, and *false* otherwise. Note that if there are no valid assignments, the result is *false*.

Fig. 3 illustrates the use of these quantifications. Predicate `p1` indicates that all elements of the vector contained in the first `n` positions of the array `a` are positive, while predicate `p2` indicates that the vector contains at least one even number (both predicates are also commented with the usual predicate calculus notation used in program specification [3]). As Fig. 3 illustrates, predicates constructed using quantifications are ESPECIFICA++ predicates, analogous to those discussed in previous subsections, and are therefore executable objects.

E. Quantified Expressions

Following the usual notation used in the formal program specification, ESPECIFICA++ introduces five types of quantified expressions. In four of them, V is itself an expression. Thus, in *typeSpecs* in (1), it is necessary to specify first the type of the values in the range and then the type returned by V. Additionally, `val` is used as the *ev* clause in (1), introducing the V expression to be evaluated on the different assignments to the quantified variables. These four quantifications are as follows:

- *Sum*. The `sum` quantifier is used. The constructed expression returns the sum of all the values returned by V. If there are no valid assignments, it returns `0`.
- *Product*. This quantification uses the `prod` quantifier. The constructed expression returns the product of all the values returned by V. If there are no valid assignments, it returns `1`.
- *Minimization*. The `mini` quantifier is used. The constructed expression returns the minimum of all the values returned by V. If there are no valid assignments, it signals an error.
- *Maximization*. It uses the `maxi` quantifier. The resulting expression returns the maximum of all the values returned by V. Like `mini`, it signals an error if there are no valid assignments.

Finally, a fifth quantification, *Count*, is introduced where V is a predicate. In this case, the quantifier is `number` and `yield` is used as the *ev* clause. The expression returns the number of assignments for which V evaluates to true.

Fig. 4 illustrates all these quantifications. Expression `e1` sums all the even elements in the vector stored in the first `n` positions of `a`. Expression `e2` multiplies all elements greater than `5` in this vector. Expression `e3` finds the minimum of the even numbers, while expression `e4` finds the maximum value in the vector. Finally, expression `e5` counts the number of elements greater than the first one. As in Fig. 3, formalizations using the usual notation in program specification are also included as comments.

Code

```
int a[]={1,2,5,15,21};
int n = 5;
var(int, i);
/* ∀i:0≤i<n:a[i]>0 */
auto p1= forall<int>(i).range(0, n-1).
                        yield(pred(a[*i] > 0));
/* ∃i:0≤i<n:a[i]%2=0 */
auto p2= exits<int>(i).range(0, n-1).
                       yield(pred(a[*i]%2 == 0));
cout << boolalpha;
cout << p1() << " " << p2() << endl;
```

Output

```
true true
```

Fig. 3. Examples of universal and existential quantifications in ESPECIFICA++.

Code

```
int a[] = { 7,2,5,10,21 };
int n = 5;
var(int, i);
/* Σi:0≤i<n ∧ a[i]%2=0: a[i]*/
auto e1 = sum<int, int>(i).range(0, n - 1).
              filter(pred(a[*i] % 2 == 0)).
              val(exp(a[*i]));
/* Πi:0≤i<n ∧ a[i]>5: a[i]*/
auto e2 = prod<int, int>(i).range(0, n - 1).
              filter(pred(a[*i]> 5)).
              val(exp(a[*i]));
/* min i:0≤i<n ∧ a[i]%2=0: a[i]*/
auto e3 = mini<int, int>(i).range(0, n - 1).
              filter(pred(a[*i] % 2 == 0)).
              val(exp(a[*i]));
/* max i:0≤i<n: a[i] */
auto e4 = maxi<int, int>(i).range(0, n - 1).
              val(exp(a[*i]));
/* # i:0≤i<n: a[i]>a[0] */
auto e5 = number<int>(i).range(0, n - 1).
              yield(exp(a[*i]>a[0]));

cout << e1() << " " << e2() << " " << e3()
     << " " << e4() << " " << " " << e5() << endl;
```

Output

```
12 1470 2 21 2
```

Fig. 4. Examples of quantified expressions in ESPECIFICA++

F. Abstraction Mechanisms

Although ESPECIFICA++ does not explicitly support abstraction mechanisms, being embedded in C++, it can take advantage of C++'s own abstraction mechanisms (for example, C++ functions to introduce symbols for predicates and expressions). These mechanisms can facilitate the specification process by promoting a top-down approach and improving the readability of specifications. However, it should be noted that if parts of quantifications are abstracted, it will be necessary to pass quantified variables as arguments. To describe the types of these parameters, ESPECIFICA++ introduces the `Var<T>` primitive, which constructs the type of quantified variables that store values of type *T*. Section III will illustrate the use of these abstraction mechanisms.

G. Implementation

In the backstage, ESPECIFICA++ heavily relies on the *lambda expression* mechanism (anonymous functions) supported since C++11 [8]. In fact, the predicates and functions constructed with ESPECIFICA++ are represented through lambda expressions that capture the states of their occurrence contexts (i.e., by *lexical closures* [9]). The primitives `pred`, `exp`, and `var` are supported via macros, the combination mechanisms through function templates that overload the different operators involved, and the quantification mechanisms through class templates. These templates use the *fluent interface* pattern [6] to appropriately support clause chaining in quantifications. The implementation is provided as a file `especificapp.h`, which can be used like any other C++ header file.

III. CASE STUDY

We will illustrate the use of ESPECIFICA++ with a case study focused on a typical problem in iterative algorithm design: the longest constant segment problem, which is an instance of the *longest segment problem* [10]. This problem consists of finding, given a non-empty vector of integers, the longest *constant segment* (i.e., sequence of consecutive elements where the same element is repeated).

The development of an algorithm to solve this problem begins by specifying its *precondition* (the condition that the input data must meet) and its *postcondition* (the condition that the results must satisfy). If we assume that the vector is stored in the first `n` positions of an array `a` of size `N`, the precondition is simple (`n` must be between `1` and `N`). On the other hand, assuming that the result of the problem will be stored in a variable `resul`, the postcondition must indicate that this variable will contain the maximum length of all constant segments in the vector. Fig. 5 shows useful definitions, expressed in ESPECIFICA++, to represent this postcondition. In particular, the `max_lcs` function constructs the required predicate. To achieve this, it uses a `cons_seg` function that defines a predicate that is *true* when the segment between two given indices is constant. Note that `cons_seg` assumes that the indices are quantified variables, as required in the definition of `max_lcs` .

Once the problem to be solved is specified, the next step is to devise an efficient solution (with a time complexity in $\Theta(n)$). To achieve this, the vector is explored from left to right, maintaining the length of the longest constant segment immediately to the left of the current position, as well as the length of the longest constant sequence in the portion of the vector already explored. The current position will vary between `1` and `n`. All these conditions, in fact, are part of the *invariant* that characterizes the behavior of the iterative process (i.e, the conditions that must be met by the states at the beginning of the process, as well as after each iteration) [11][12]. In describing these conditions, `max_lcs` can be reused. Additionally, it will be useful to employ a `last_cons` function, which, given a position and a length, constructs a predicate that is *true* when this length is the length of the longest constant segment immediately to the left of the given position (Fig. 6).

Once the algorithm's concept is clarified and the invariant is established, its implementation becomes straightforward. Assuming that the current position is stored in a variable `i` and the length of the last explored constant segment is stored in a variable `len`:

- The invariant can be established initially by setting `i`, `len` and `resul` (the variable that will store the result) to `1`.

```
auto cons_seg (int a[], Var<int> i, Var<int> j){
 var(int, k);
 /* ∀k:i≤k≤j:a[i]=a[k] */
 return forall<int>(k).range(i,j).
              yield(pred(a[*i]==a[*k]));
}
auto max_lcs(int a[], int n, int resul){
  var(int, i);
  var(int, j);
 /* max i,j:0≤i≤j≤n ∧ cons_seg(a,i,j):(j-i)+1 */
  return
    exp(resul) ==
    maxi<int, int>(i, j).
      range(0, n - 1).
      filter(pred(*i <= *j)&&cons_seg(a,i,j)).
      val(exp((*j - *i) + 1));
}
```

Fig. 5. Abstractions for the postcondition

```
auto last_cons(int a[], int i, int len) {
  var(int, k);
/* (∀k:i≤k≤j:a[i-len]=a[k])∧
   (i-len =0 ∨ a[i-len-1] ≠ a[i-len] )    */
  return forall<int>(k).range(i-len, i-1).
           yield(pred(a[i-len] == a[*k])) &&
           pred(i-len == 0||a[i-len-1]!=a[i-len]);
}
```

Fig. 6. Abstractions for the invariant.

- The iterative process continues until `i` reaches `n`.
- In each iteration, if `a[i]` contains the same value that repeats in the segment being explored, the segment is extended (incrementing `i` and `len`), updating `resul` if necessary. If not, a new segment is started (incrementing `i`, and setting `len` to `1`).

Finally, since `i` is incremented in each iteration, to ensure that the algorithm terminates, we can observe that the algorithm maintains the *bound condition* `n - i ≥ 0` invariant.

Fig. 7 shows the resulting implementation, annotated with the conditions previously described. To include the annotations, the `assert` macro from C++'s `cassert` library is used. This macro allows checking the fulfillment of required conditions during program execution, and aborts execution with an error report if a violation occurs. An `assert` with the precondition is included at the start of the algorithm, and another one with the postcondition at the end. Additionally, `assert`s are included for the invariant conditions and the bound condition, placed right before the loop (to check their validity just before the beginning of the loop execution) and at the end of the loop body (to check their validity after each iteration). Note that some of the simpler conditions are expressed directly as C++ expressions (as they can be described with simple expressions, there's no need to reify them as ESPECIFICA++ expressions). The rest use the ESPECIFICA++ definitions introduced earlier.

```
int max_len_cons_seq(int a[], int n) {
  /*PRE:*/assert(0 < n && n <= N);

   int i = 1;
   int len = 1;
   int resul = 1;

   /*INV:*/ assert(max_lcs(a, i, resul)());
   /*INV:*/ assert(last_cons(a, i, len)());
   /*INV:*/ assert(0<i<=n);
   /*BOUND:*/  assert((n - i) >= 0);
    while (i < n) {
     if (a[i] == a[i - len]) {
         i++;
         len++;
         if (len > resul) resul = len;
     }
     else { i++; len = 0;}

     /*INV:*/ assert(max_lcs(a, i, resul)());
     /*INV:*/ assert(last_cons(a, i, len)());
     /*INV:*/ assert(0<i<=n);
     /*BOUND:*/  assert((n - i) >= 0);
    }
    /*POST:*/assert(max_lcs(a, n, resul)());
    return resul;
}
```

Fig. 7. Annotated implementation.

IV. PRELIMINARY EVALUATION

We have conducted a preliminary evaluation of the suitability of ESPECIFICA++ for our educational strategy based on a SWOT (*Strengths*, *Weaknesses*, *Opportunities*, and *Threats*) analysis [13]. Below, we summarize the main findings obtained:

- *Strengths*: (i) *Goal fulfillment*: All participants in the project agreed that ESPECIFICA++ meets the goal of providing a DSL that enables the construction of executable specifications. (ii) *Alignment with formal notation*: It reasonably reflects the formal notation used in introductory algorithm courses, facilitating the transition from formal to practical approaches. (iii) *Integration with* C++: Its close relationship with C++ allows students to integrate specifications directly into implementations using tools like `assert` and leverage C++'s abstraction mechanisms. This also has the advantage that students do not have to switch to a more specific programming language, which supports formal assertions, but can continue to use a mainstream programming language (C++, in our case). (iv) *Conciseness*: ESPECIFICA++ has a limited number of constructs, making it easier to learn and apply in introductory courses. (v) *Incremental validation*: Predicates and expressions can be developed and incrementally validated by students through test cases, promoting gradual learning.
- *Weaknesses*: (i) *Limited error diagnostics*: Being embedded in C++, error messages generated by incorrect specifications resemble those of C++, which can be confusing for students as they are not tailored to ESPECIFICA++. (ii) *Difficulty diagnosing execution errors*: Incorrect specifications can lead to execution errors that are hard to diagnose for less experienced students. (iii) *Confusing variable behavior*: The behavior of variables in predicates and expressions, where changes in program variables are not reflected automatically, can be perplexing—particularly in distinguishing between program variables and quantified variables. (iv) *Potential inefficiency in the execution of quantifications*: Quantifications explore all possible assignments of values in the range to the quantified variables. If the range has n values, and there are k quantified variables, n^k assignments will be explored (i.e., for a fixed number k of quantified variables, the number of assignments grows as a polynomial of degree k, while for a fixed number n of values, it grows exponentially with the number of quantified variables). Fortunately, in practice it is rare to find quantifications with more than two variables. Nevertheless, executable specifications will continue to be inherently more inefficient than final implementations, an aspect that will have to be considered in the selection of test batteries for ESPECIFICA++'s executable specifications.
- *Threats*: (i) *Resistance from formalist approaches*: The idea of treating specifications as executable programs may not be well-received by those educators who favor more formalist approaches, potentially leading to resistance against adopting ESPECIFICA++. (ii) *Dependency on C++*: ESPECIFICA++'s dependence on C++ may limit its

usefulness for educators and students who prefer other programming languages, hindering its adoption outside of C++ environments.

- Opportunities: (i) *Development of formal specification skills*: The difficulty faced by students in mastering formal specification is a key opportunity for ESPECIFICA++, as it provides a practical tool to improve this competency, which has traditionally been challenging in introductory algorithm courses.

This analysis highlights the potential of ESPECIFICA++ to improve learning in programming and algorithm courses, though it faces challenges that need to be addressed to maximize its adoption and effectiveness.

V. RELATED WORK

There are multiple works that propose strategies to facilitate the development of formal specification competencies among students. For example, in [14] *Dafny* is proposed as a vehicular language in an introductory algorithms course (*Dafny* is a programming language that integrates formal specifications into its type system, ensuring that the compiler rejects programs that cannot be proven to meet their specification [15]). In [16], an educational strategy is also proposed, based on *Dafny*, for learning formal methods. This strategy starts with programs and advocates for the progressive incorporation of assertions (preconditions, postconditions, invariants). In [17], also in the context of *Dafny*, mechanisms are proposed for correcting specifications that are not satisfied by the programs, which is useful in scenarios where the program is considered correct rather than the specification. In [18], the lack of support from programming environments for languages where programs incorporate their specifications is identified as a key factor for success in education, and the integration of the PEST language (a language similar to *Dafny*) with the Eclipse platform is proposed.

While the purpose of these strategies is similar to ours, the difference lies in that we promote executable specifications that can be developed and validated independently of the programs and later easily integrated with them. In this way, our approach is somewhat complementary to the use of languages that support automatic verification in compile-time: with our approach, the student can crystallize the specifications with the help of a mainstream programming language and later verify their programs using more specialized languages (e.g., the aforementioned *Dafny* and PEST).

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we have described ESPECIFICA++, a DSL embedded in C++ that allows students to create executable specifications. This enables students to view specifications at the same level as programs, as active objects that must *work*, rather than mere documentation. This is expected to facilitate the development of formal specification skills, an area where the average student typically encounters appreciable difficulties.

In our preliminary evaluation, we have verified that the DSL allows for the effective creation of executable specifications at the level required in an introductory algorithmics course. We have also identified certain weaknesses and threats, which we are working to resolve and plan to address in the future. Specifically, we are currently exploring the creation of similar DSLs in other frequently used object-oriented programming languages (particularly Java and Python). We are also working on approaches to improve error diagnostics and potential inefficiency issues. In the immediate future, we plan to conduct an educational efficacy evaluation of the approach with students in introductory algorithmics courses at Complutense University of Madrid (UCM). Additionally, we aim to develop a repository of ESPECIFICA++ specification activities and deploy these activities into an online automated judge system (e.g., DomJudge [19]) for student use.

REFERENCES

[1] CC2020 Task Force, Computing Curricula 2020: Paradigms for Global Computing Education. New York, USA: ACM, 2020.

[2] M. Broy et al., "Does Every Computer Scientist Need to Know Formal Methods?," Form. Asp. Comput., in press.

[3] D. Gries, The Science of Programming. Springer, 1981.

[4] A. Kaldewaij, Programming: The Derivation of Algorithms. Prentice Hall, 1990.

[5] T. Magdalina and D. Orozova, "Training Difficulties in Deductive Methods of Verification and Synthesis of Program," Int. J. Advanced Computer Science and Applications, vol. 9(7), pp. 18-22, 2018.

[6] M. Fowler, Domain-Specific Languages. Addison-Wesley, 2010.

[7] M. Mernik, J. Heering and A.M. Sloane, "When and How to Develop Domain-Specific Languages," ACM Comp. Surv., vol. 37(4), pp. 316-344, 2005.

[8] B. Stroustrup, The C++ Programming Language, 4th ed. Addison Wesley, 2013.

[9] D.P. Friedman and M. Wand, Essentials of Programming Languages, 3rd ed. Cambridge, MA: MIT Press, 2008.

[10] H. Zantema, "Longest segment problems," Science of Comp. Programming, vol. 18(1), pp. 39-66, 1992.

[11] C.A. Furia, B. Meyer, B. and S. Velder, "Loop invariants: Analysis, classification, and examples," ACM Comput. Surv., vol. 46 (3), pp. 1-51, 2014.

[12] D. Gries, "A note on a standard strategy for developing loop invariants and loops," Science of Comp. Programming, vol. 2(3), pp. 207-214, 1982.

[13] M.A. Benzaghta, A. Elwalda, M.M. Mousa, I. Erkan and M. Rahman, "SWOT analysis applications: An integrative literature review," Journal of Global Business Insights, vol. 6(1), pp. 55-73, 2021.

[14] E. Braude, "Using the Dafny Verification System in An Introduction To Algorithms Class," Computer Science and Education in Computer Science, vol. 10(1), pp. 23-29, 2014.

[15] K.R.M. Leino, "Accessible software verification with Dafny," IEEE Software, vol. 34(6), pp. 94-97, 2017.

[16] J. Noble, D. Streader, I.O. Gariano and M. Samarakoon, "More programming than programming: Teaching formal methods in a software engineering programme," in NASA Formal Methods, Lecture Notes in Computer Science, vol. 13260, J.V. Deshmukh, K. Havelund and I. Perez, Eds. Springer, 2022, pp. 431-450.

[17] A. Abreu, N. Macedo and A. Mendes, "Exploring Automatic Specification Repair in Dafny Programs," 2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), Luxembourg, Luxembourg, 2023, pp. 105-112.

[18] G. de Caso, D. Garbervetsky and D. Gorín, "Integrated program verification tools in education," Software: Practice and Experience, vol. 43(4), pp. 403-418, 2013.

[19] M.T. Pham and T.B. Nguyen, "The DOMJudge Based Online Judge System with Plagiarism Detection," in Procs. 2019 IEEE-RIVF International Conference on Computing and Communication Technologies, pp. 1-6. IEEE, 2019.