Revista Brasileira de Informática na Educação – RBIE
Brazilian Journal of Computers in Education
(ISSN online: 2317-6121; print: 1414-5685)
http://br-ie.org/pub/index.php/rbie

Submission: 09/Fev/2021; 1st round notif.: 19/Apr/2021; New version: 11/May/2021; 2nd round notif.: 19/Jun/2021;
Camera ready: 28/Jun/2021; Edition review: 30/Jun/2021; Available online: 04/Jul/2021; Published: 04/Jul/2021;

A-Learn EvId: A Method for Identifying Evidence of Computer Programming Skills Through Automatic Source Code Assessment

Andres Jessé Porfirio
Universidade Tecnológica Federal do Paraná (UTFPR)
andresporfirio@utfpr.edu.br

Roberto Pereira
Universidade Federal do Paraná (UFPR)
rpereira@inf.ufpr.br

Eleandro Maschio
Universidade Tecnológica Federal do Paraná (UTFPR)
eleandrom@utfpr.edu.br

Abstract

Contextualized in the teaching of computer programming in Computing courses, this research investigates aspects and strategies for automatic source code assessment. Continuous on-time assessment of source codes produced by students is a challenging task for teachers. The literature presents different methods for automatic assessment of source code, mostly focusing on technical aspects, such as functional correctness assessment and error detection. This paper presents the A-Learn EvId method, having as the main characteristic its focus on the assessment of high-level skills instead of technical aspects. Automatically assessing high-level skills gives insights into the thought process students used to elaborate their responses, contributing to quality and timely feedback generation. The method is characterized by three fundamental steps: (1) inserting students' source code as input data; (2) identifying evidence of skills through automatic strategies; and (3) representing identified skills through a student model. The following contributions are highlighted: updating the state of the art on the topic; a set of 37 skills identifiable through 9 automatic source code assessment strategies; construction of datasets totaling 8651 source codes.

Keywords: *Computer programming, Automatic assessment, Skills identification*

Cite as: Porfirio, A. J., Pereira, R., Maschio, E. (2021). A-Learn EvId: A Method for Identifying Evidence of Computer Programming Skills Through Automatic Source Code Assessment. Revista Brasileira de Informática na Educação, 29, 692-717. DOI: 10.5753/RBIE.2021.29.0.692.

1 Introduction

Computer programming is one of the very first topics in Computer Science courses and, sometimes, one of the most complex from students' point of view (Ullah et al., 2018). Learning how to program computers requires students to understand a new set of concepts and to develop new thinking strategies very different from what they are used to.

Research and Development of computer programming teaching support tools is a widespread topic in the literature, typically aiming to provide resources to support teachers' activities. Among these activities, two of the major difficulties faced are assessing students' programming exercises (Souza et al., 2016) and providing individualized timely feedback (Ihantola et al., 2010; Ullah et al., 2018).

Assessing large amounts of source code developed by a large number of students is a complex and exhausting task for teachers (Ullah et al., 2018; Rahman & Nordin, 2007). This situation is even more critical in the current pandemic times where much of our educational systems have migrated to online environments: while teachers still have to deal with classes, exercises, and student assessments, several new tasks and responsibilities have been added to their workload, including mastering different online systems and preparing interactive material for students (Pimentel & Carvalho, 2020). Therefore, investigating methods and developing supporting tools is a way to help teachers in their work, especially by reducing their workload with repetitive tasks and offering resources for them to provide personalized tutoring.

Although the automatic source code assessment has come under investigation for decades (Liang et al., 2009; Rahman & Nordin, 2007; Souza et al., 2016; Ullah et al., 2018), identifying evidence of computer programming skills is still a challenge. Several source code aspects can be assessed via different strategies (Souza et al., 2016), not always automatically possible, and such a diversity of aspects leads to a dispersed literature in which numerous methodologies are applied to problem-specific scenarios.

A systematic mapping of the literature (Porfirio, 2020) revealed many works dealing with automatic assessment, usually focusing on technical aspects only, such as functional correctness (Jackson & Usher, 1997; Morris, 2003) and error detection (Wilcox et al., 1976; Ahmed et al., 2018). Initial programming courses, however, usually have their syllabi focused on concepts and desired skills, not on technical aspects, which are often conveyed through classes and assessed in specific situations in which students are supposed to succeed only if they have mastered certain programming skill.

For this research, the definition of skill is grounded in DeKeyser's skill acquisition theory (VanPatten & Williams, 2015, p. 95), which accounts for how people progress in learning skills. The theory holds that knowledge is initially acquired by the apprentice, who subsequently starts to manifest it through behavioral changes. In the computer programming context, we consider that students acquire knowledge through learning concepts, and later manifest it by applying different programming resources in source codes. Therefore, we consider that behavioral changes are marked by using previously unreported programming resources, thus suggesting evidence of new skills development.

When it comes to automatic source code assessment, Hettiarachchi et al. (2013) presents two main types: knowledge-based and skill-based. Knowledge-based assessment is described by the

authors as a simplified form of assessment, usually easy to apply, but with a limited scope that may lead to just a quiz of facts about the area of study. Skills-based assessment, in turn, is described as more authentic and capable to assess higher-order cognitive skills, however, hard to apply. Also, while knowledge-based assessment is related to simple aspects and rarely gives any insight into the thought process students used to elaborate their responses (analogous to technical aspects assessment previously mentioned), skills-based assessment can be applied to evaluate high-level cognitive skills (Hettiarachchi et al., 2015).

The **central problem** addressed in this research is focused on the teacher's point of view: the challenge of assessing students' source codes and providing feedback in a continuous and timely manner and, with this, identifying the manifestation of programming skills. Therefore, the **main objective** of this research is to investigate a method for the automatic and continuous assessment of programming skills via source code analysis. To achieve the main objective, the following activities were established:

- Identify the state of the art and elaborate a literature mapping;
- Identify programming skills candidate to automatic assessment;
- Identify a programming skill-set able to be automatically assessed;
- Investigate strategies to automatically assess the identified skills;
- Implement strategies as algorithms that receive student source codes as input and returns the identified skills as output;
- Implement a learner model to represent student knowledge based on a predefined skill-set;
- Apply strategies results as input data to feed the learner model;
- Provide resources to track student progress through the learner model; and
- Evaluate the proposed method regarding its automatic assessment capacity.

This paper summarizes the main results from the first author's doctoral thesis (Porfirio, 2020), defended at the Federal University of Paraná (UFPR). The paper is an extended and revised version of the paper (Porfirio et al., 2020) published and awarded in the "*Alexandre Direne Contest of Theses in Computers in Education (CTD-IE) 2020*", promoted by the Special Committee of Computers in Education of the Brazilian Computer Society. It extends the original version by further developing the research foundation, method and discussion, and by presenting details of the A-Learn EvId method, from the skills-set definition process to the student model that represents the method's output, covering the automatic strategies responsible for identifying evidence of skills manifestation.

2 State of Art

A systematic literature mapping (Porfirio, 2020) was conducted to identify what aspects of source code have been assessed automatically, and what strategies have been adopted. Based on 126

papers selected from databases ACM, IEEE, Scopus, Scielo, and CEIE, until 2019, our analysis revealed 43 different aspects of source code identified automatically via 25 different strategies.

The literature mapping revealed different initiatives to use automatic strategies for source code assessment. Data revealed that the most popular aspects addressed are the ones dealing with automatic assessment of source code as a whole, pointing to generic results such as *Functional Correctness* assessment and detection of *Semantic and Compilation Errors*. Table 1 presents the main aspects identified and the number of papers (total) and percentage related to them.

Table 1: Aspect, total of papers and percentage.

Aspect	**Total**	**Percentage**
Functional Correctness	66	42.31
Semantic Errors	21	13.46
Compilation Errors	17	10.90
Syntactic Errors	12	7.69
Complexity, Efficiency, Style	10	6.41
Execution Errors	9	5.77
Other, Simulation	6	3.85
Antipatterns, Concurrency, Methods, Computational Thinking	5	3.21
Conditionals, Problem Solving Strategy, Originality, Tests, Variables	4	2.56
Classes, Lexical Errors, Loops, Types	3	1.92
Abstraction, Constructors, Input and Output, Scope, Inheritance, User Interface, Recursion	2	1.28
Algorithm, Constants, Enumerators, Heterogeneous Structures, Homogeneous Structures, Events, Exceptions, Functions, Interfaces, Polymorphism, Procedures, Code Reuse, Strings	1	0.64

Regarding the strategies employed for the automatic assessment process, we identified *Test Cases* and *Unit Tests* as popular approaches. The majority of studies (96.03%) focused on applying three or fewer strategies (59.52% applied only one), providing specialized solutions to assess specific source code aspects. While approaches focused on specific aspects can be powerful for specific purposes, they become limited for more complex contexts such as programming teaching where all the possible aspects may coexist and even influence each other. Therefore, using hybrid approaches to mix different strategies and assess multiple code aspects simultaneously was identified as a gap in the literature.

The systematic mapping suggested that the more aspects are assessed in a source code the richer the feedback possibilities could be and, consequently, more clues about students' programming skills could be provided. Results suggested that, although specialized solutions are adequate for specific contexts, methods combining multiple strategies to assess multiple aspects and to provide detailed and holistic feedback is a research gap. The A-Learn EvId method conceived in this research contributes to addressing this gap, improving our capacity to identify evidence of students' programming skills by automatically analyzing the source code they produce. The A-Learn EvId method is capable to identify evidence for 37 different skills by applying 9 automatic source code assessment strategies. In comparison to the analyzed literature, it was found that 92.06% of the papers deal with four or fewer aspects, usually applying less than four strategies. Also, the analyzed literature points out a maximum of 10 aspects evaluated in a single study

(Rajala et al., 2016).

3 A-Learn EvId: Automatic Learning Evidence Identification Method

The Automatic Learning Evidence Identification (A-Learn EvId) method adopts a hybrid approach that employs static and dynamic source code analysis strategies to identify learning evidence, valuate programming skills, and feed a learner model. While the static approach consists of assessing source codes without performing their execution, the dynamic approach requires the source code to be executed. To conceive the method we take advantage of literature experiences regarding aspects automatically identifiable, strategies employed, as well the results and learned lessons about limitations and possibilities.

Figure 1 presents a scheme for the method. Source code serves as input to assessment, where different strategies are applied to identify evidence used for feeding a skill-based learner model (output). Evidence can be identified from single or multiple strategies, as well as from inference through combinations of previously assessed skills.

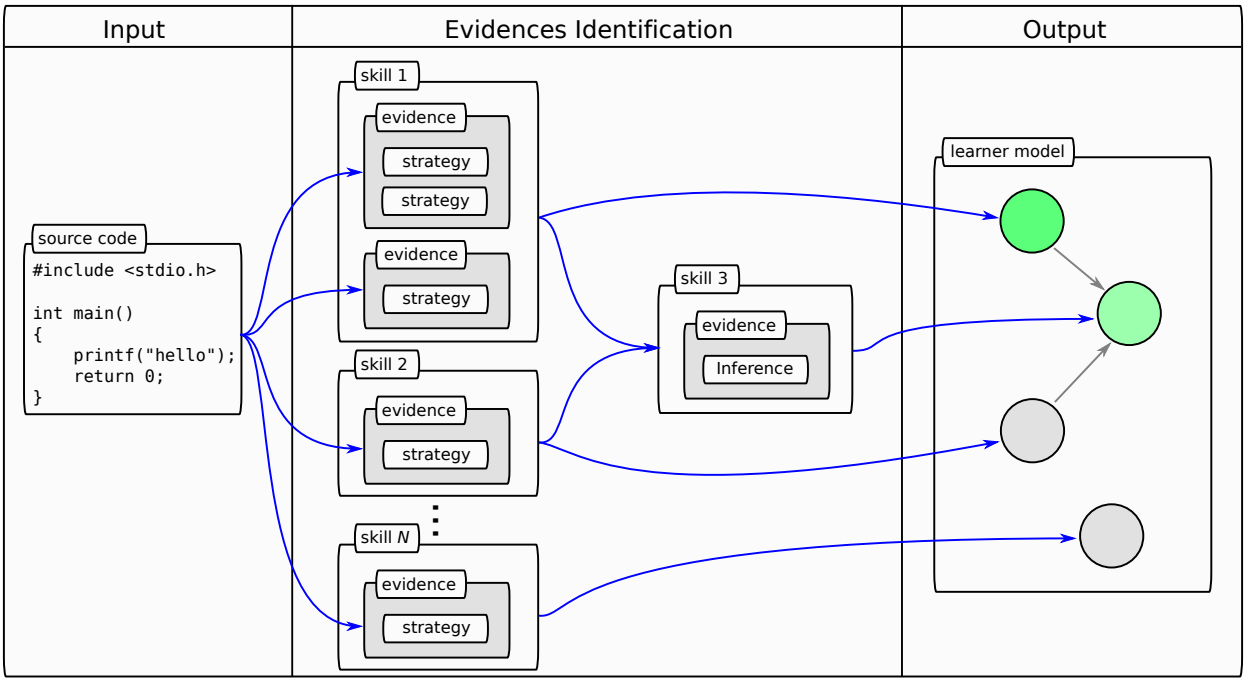

Figure 1: A-Learn EvId Method overview.

Figure 2 presents examples of skills and their respective valuation sources (when automatically identifiable). Skills are valuated by one or more evidence; evidence can be implemented with one or more automatic strategies; and, finally, each strategy analyses a source code and returns a valuation regarding certain programming aspects (e.g., a percentage of success regarding the use of a specific programming resource).

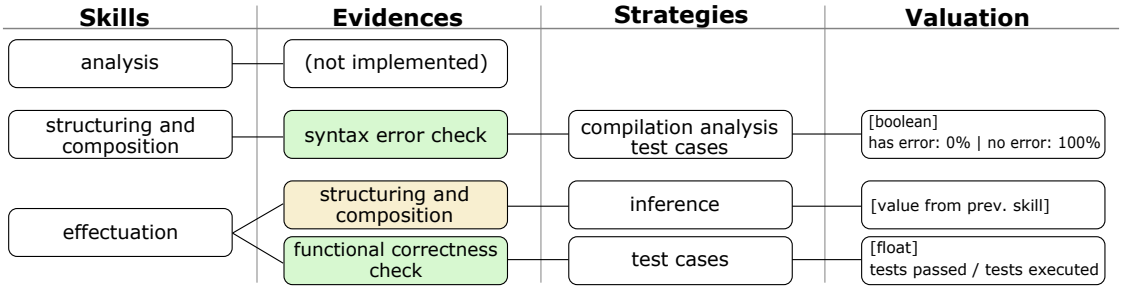

Figure 2: Skills valuation sample.

3.1 Skill Set Definition

Ideally, the A-Learn EvId method should use as much information as possible to realistically assess students' skills. The main challenge at this point concerns which aspects should be considered in our method implementation. Previous literature mentions the existence of desired skill sets related to training students in programming. Research from Maschio (2013) and analysis of real-world programming courses syllabus were used to define a *Full Skill Set* containing aspects candidates to be implemented via strategies for automatic skills identification.

Situated on the imperative paradigm, Maschio (2013) presents a subset of 41 programming skill categories organized in the form of an overlay graph (genetic graph). The graph models knowledge from initial concepts, such as variables and constants, to conditionals and repetition structures. In addition, interconnections between skills, such as prerequisites, dependencies, analogies, and generalizations, are also represented.

Considering the many skills presented by Maschio (2013), defining which are candidates for automatic assessment can be a challenging task. There may be different views and approaches to introduce students to programming, however, there are basic concepts in programming that must be mastered regardless of the adopted approach. To identify basic concepts, the syllabus of ten introductory programming chairs from different Brazilian universities were analyzed. The analysis was organized as follows:

1. Two federal universities from each of the 5 Brazilian regions were selected: North, Northeast, Central-West, Southeast, and South;
2. For each university, select an undergraduate course in Computing area, prioritizing bachelors in Computer Science;
3. For each course, extract the syllabus of the first course to teach Algorithms/Programming offered to students; and
4. Summarize the programming topics, counting their occurrences.

For the analysis, universities were selected based on the RUF 2018 Ranking[1] for Computing courses. Results are shown in Table 2.

[1]RUF - Ranking Universitário Folha, website: http://ruf.folha.uol.com.br/2018/ranking-de-cursos/computacao/ Last access: 30 January 2021.

Table 2: Selected universities.

Region	University
Central-West	UFG - Universidade Federal de Goiás (Federal University of Goias) UnB - Universidade de Brasília (University of Brasilia)
Northeast	UFCG - Universidade Federal de Campina Grande (Federal University of Campina Grande) UFPE - Universidade Federal de Pernambuco (Federal University of Pernambuco)
North	UFAM - Universidade Federal do Amazonas (Federal University of Amazonas) UFPA - Universidade Federal do Pará (Federal University of Pará)
Southeast	UFMG - Universidade Federal de Minas Gerais (University of Minas Gerais) UFRJ - Universidade Federal do Rio de Janeiro (Federal University of Rio de Janeiro)
South	UFRGS - Universidade Federal do Rio Grande do Sul (Federal University of Rio Grande do Sul) UFSC - Universidade Federal de Santa Catarina (Federal University of Santa Catarina)

By analyzing the syllabus of each computing course from the selected universities, a ranking of the most cited topics was elaborated. Because of variations in the spelling of concepts (e.g., "conditional structures" can be called "decision structures", or can even be generalized as "control structures"), equivalences of different concepts and synonyms were manually analyzed and grouped.

Table 3 shows the most common topics found in the introductory programming chairs, only topics cited in more than five syllabi were considered[2]. Also, related skills (from Maschio's 41 skills) are highlighted. In this case, a skill refers to the ability to understand a concept and apply it appropriately in the code – or manifest it, as stated by VanPatten & Williams (2015, p. 95). From the 10 syllabus common topics, 7 have related skills, however, 3 of them were out of Maschio's research scope and, consequently, not represented on his skill-set: *arrays*[3], *functions*, and *matrices*. Considering the popularity of these unrelated topics in syllabus analysis, they were included for further analysis.

"Introduction to Programming" was cited as a syllabus topic, however we considered this topic as the main objective that is achieved by mastering the other ones. Therefore, we consider that students must develop abilities related to different topics and that the more developed their skills are in such topics, the more skilled students tend to be in basic programming activities. In the end, developing skills related to all the topics means the student masters the introduction to programming. The resulting 44 skills were then classified regarding their potential and priority for identification through automatic source code analysis, results are shown in Figure 3.

From the 44 skills present in our full skill set, 37 skills were selected for automatic assessment/strategy application. Skills classified as challenging/not feasible were not selected due to the following reasons: (1) *algorithm* and *analysis* were considered abstract skills, strongly dependent on the interpretation of student thinking; (2) *value changes*, *loss of value*, *pipelining (conditionals)* and *pipelining (loops)* were considered problem-dependent skills, hard to assess without knowing the program execution context and objectives[4]; (3) *counters and accumulators* were also considered problem-dependent skills, however, Gerdt & Sajaniemi (2006) suggests

[2]Full table data and supplementary research files are available online at http://bit.ly/doc_syllabus.

[3]Arrays refer to one-dimensional homogeneous structures, also called vectors.

[4]For Maschio (2013), the *loss of value* skill concerns to overlap and consequent loss of stored values. No strategy has been identified that can accurately assess this type of skill because it is difficult to know whether the loss of value was a student mistake or an intentional act, for example by reusing the same variable for another purpose

it is possible to identify Pascal programming language role variables, including the location of "counters" through flow analysis strategy. *Counters and accumulators* were not explicitly listed as common topics in the analyzed syllabi and left as future work for our method.

Table 3: Programming topics ranking.

Syllabus Topic	Occurrences	Related Skills
Conditional structures	10	Control structures Conditional structures Multiple selection conditional Simple and compound conditionals
Repetition structures	10	Repetition structures Infinite loops Counted loops Conditional loops Pre-evaluated Post evaluated
Data types	9	Types compatibility Types of literals
Variables	8	Variables
Input and output	8	Output Input
Operators and expressions	8	Arithmetic expressions Relational expressions Boolean expressions Compound expressions
Arrays	8	*
Functions	8	*
Introduction to programming	8	Algorithm
Matrices	7	*

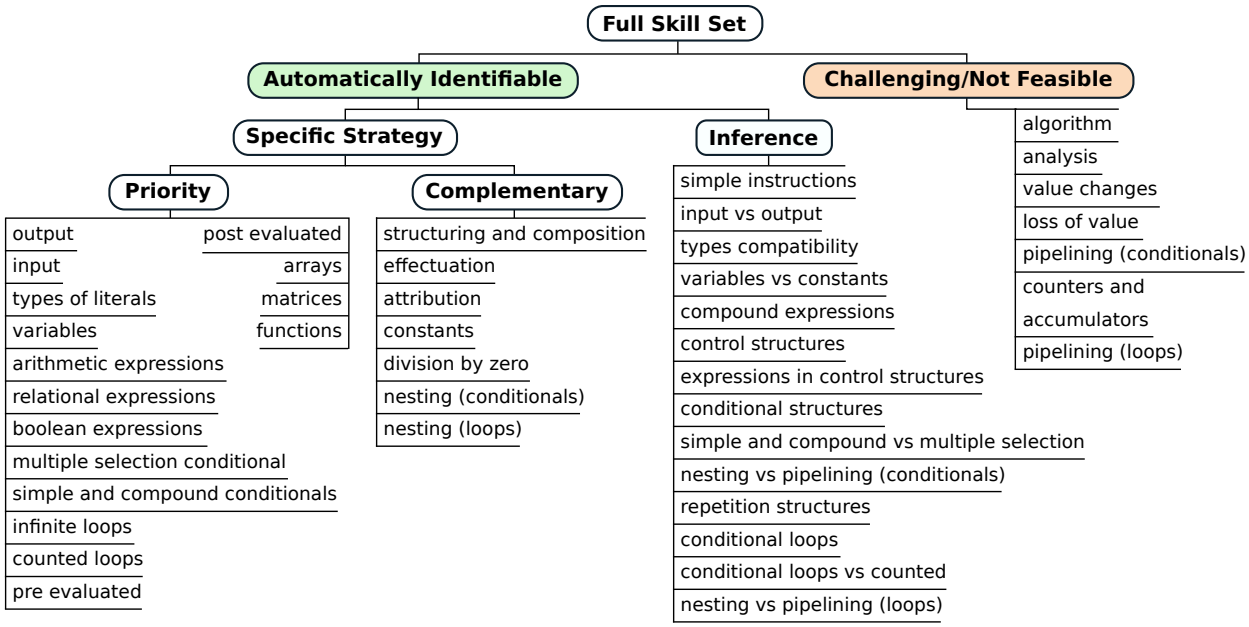

Figure 3: Skills categorized by their potential and priority for automatic identification.

3.2 Strategies Implementation

For each aspect representing a skill, automatic strategies were defined and implemented. Detecting evidence for different skills requires different strategies, some with easy automatic detection and others not viable for practicing. Considering the *automatically identifiable* skills previously defined, 9 of the 25 strategies from our systematic mapping were selected to compose the method: *AST* (Abstract Syntax Tree), *Code Mutation*, *Compilation Analysis*, *Debug Analysis*, *Execution Traces Analysis*, *Parser*, *Regular Expressions*, *Software Metrics*, and *Test Cases*. Hybrid systems employing two or more of the selected strategies were also applied. Strategies were selected according to: (1) source code aspect it was capable to analyze; (2) type of analysis (static or dynamic); (3) availability of implementation documentation; and (4) adaptation possibility.

AST and Parser: static analysis can be applied for inspecting and identifying internal source code details and small knowledge units (Kautzmann & Jaques, 2020). A common approach to performing static analysis consists of applying a Parser to decompose a source code and output details in an organized structure, such as an AST. According to Cui et al. (2010), these trees represent source code instruction hierarchy and are constructed from a process involving: (1) source preprocessing; (2) lexical analysis; (3) parsing; and (4) generation of the tree.

AST and Parser strategies are demonstrated in Figure 4. The source code is written in C language, where it is possible to notice two functions: *test* and *main*. Each function has internal elements, which can be: **(1)** variable declarations or **(2)** function calls. Executing parser strategy in the listed source code generates the AST briefly represented in the referred figure. Each function of the source code is stored in a tree node, whose children are instructions and blocks that compose it. Furthermore, the parser's ability to detect attributes of each element, such as names, arguments, and literal types of each variable/function is highlighted.

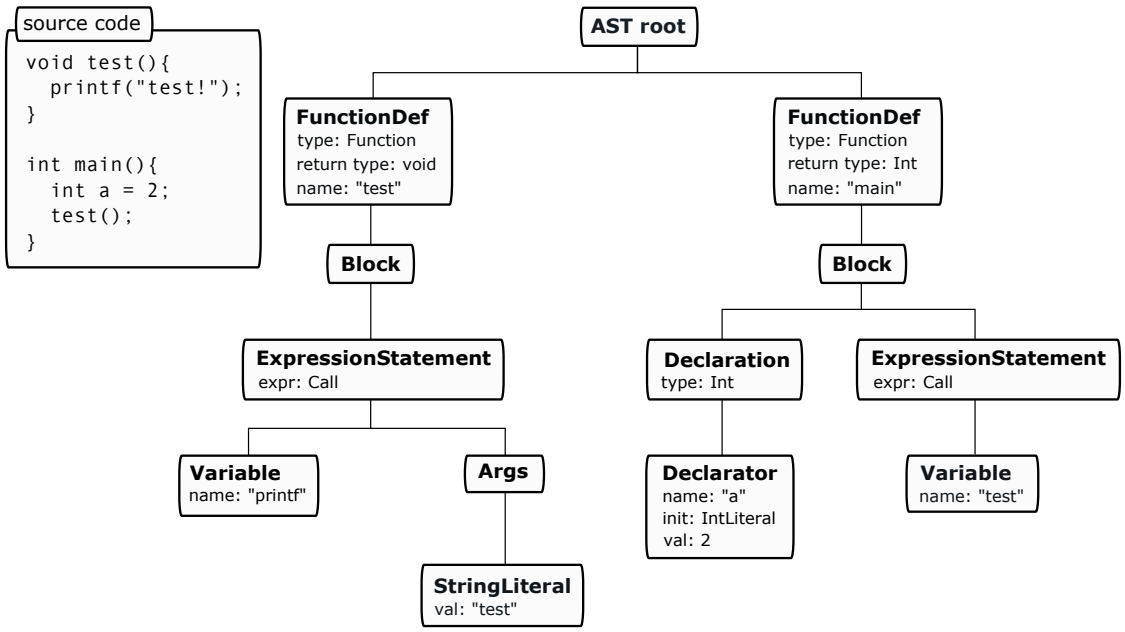

Figure 4: Source code and AST resulting from parser execution.

Thus, it is possible to perceive the parser strategy as a powerful tool able to identify internal

code construction details. Considering the parser's resulting tree, it is possible to implement heuristics to search for evidence of programming skills, such as to verify if students are capable of creating and initializing variables, declaring simple and compound conditional structures, loops among other programming instructions.

Test Cases: AST and Parser strategies are limited to analyzing source code structural elements and cannot assess programs' execution. Dynamic analysis, in turn, can provide runtime-based assessments. Test cases, although being a fairly simple strategy, are widely used and provide good results when assessing programs' execution output. Assessing a program with test cases requires the prior specification of input and expected output value sets. Strategy application is performed by executing programs with the predefined input values, followed by a comparison of the produced outputs.

Considering a previously compiled executable program, operating systems allow redirecting[5] input data to the program's process through the standard input stream. Input data can be sent as raw text files. Values specified in the input text file are automatically entered in place of keyboard input commands. Subsequently, outputs generated are captured by the standard output stream, allowing plain text comparison with the outputs expected in test cases. The program is functionally correct when outputs match, and wrong otherwise. Using multiple test cases ensures accurate assessments.

Figure 5 shows a C-Language program, which requires the user to enter two numbers and print the double of the first and triple of the second, and exemplifies the execution of a test case-based assessment on the referred program. Test cases strategy was applied to all skills assessed in our method. Preliminary assessment of functional correctness ensures any evidence found by other strategies (such as parser) refers to a functionally correct source code. Thus, gathering evidence from different strategies contributes to the accuracy of our method.

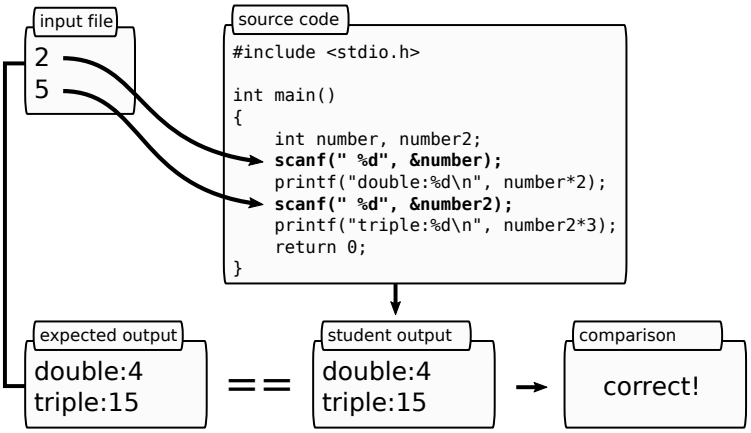

Figure 5: Test case application example.

Functional correctness assessment is an important aspect to be evaluated in students' source codes, however, its capacity does not go beyond comparing program's output. Checking programs' internal execution behavior, such as the valuation of variables during the execution process, is unfeasible with Test Cases. Such analysis requires a different strategy capable of performing

[5]Also known as I/O Redirection and Pipes.

deep runtime inspections.

Compilation and Debug analysis: Compilation and debugging tools such as GCC (GNU Compiler Collection) and GDB (GNU Project Debugger) are commonly used in teaching computer programming, specifically as support tools to teach C-Language. These applications are part of a tool-set[6] focused on transforming source codes into executable objects and can be configured so that messages can be displayed in case of failures. Compilation messages can point to where a fault was detected, usually signaling the type of error occurred and the corresponding line in the source code. Similarly, a debugger running a faulty program will interrupt when unexpected or erroneous behaviors are encountered.

GCC compiler logs can be exemplified with the C-Language source code shown in Figure 6, where a failure while printing the value of the variable *a* is present. The stored value is of floating-point type, however, the print command has been set to integer values, which will result in output's precision error. The referred figure gives a snippet of the output log generated while trying to compile the source code. GCC has detected the failure and issued a warning message indicating the given data type is floating-point but the printout is waiting for an integer argument.

source code

```
#include <stdio.h>

int main() {
  float a = 10.5;
  printf("float value:%d\n", a);
  return 0;
}
```

GCC output log

```
source.c: In function 'main':
source.c:5:10: warning: format '%d' expects argument of type 'int',
  but argument 2 has type 'double' [-Wformat=]
   printf("float value:%d\n", a);
          ^
```

Figure 6: Source code with output precision error and GCC output log.

There are also cases where a source code can be compiled correctly but give incorrect results during execution. Such situations are not detectable by the compiler, which can generate an executable object without accusing any error. Some of these situations can be detected through a debugging process, usually done manually, where the programmer executes the program step-by-step to find the error source.

Automating GDB debugging process provides resources to identify learning evidence through programs' runtime analysis. The C-Language code shown in the Figure 7 exemplifies the use of debugging process as an evidence location strategy. There is an arithmetic failure in the listed code where a division by zero occurs during the assignment operation of variable *c*. When submitting the program for execution through GDB, an output log is generated indicating the type of error occurred and its exact location. The figure shows a snippet of the debugging log, where an "Arithmetic exception" error is pointed out in the operation performed in line 7.

Analyzing compilation and debug logs can be a challenging task as, most of the time, going through all the returned output lines is necessary to identify the desired information. *Regular expressions* strategy was employed to support data localization both in static mode (i.e., looking

[6]The Gnu Project, website https://www.gnu.org/gnu/thegnuproject.en.html.

source code

```
1  #include <stdio.h>
2
3  int main() {
4
5    int a = 10;
6    int b = 0;
7    int c = a/b;
8
9    return 0;
10 }
```

GDB output log

```
Program received signal SIGFPE, Arithmetic exception.
0x00000000004004ec in main () at source.c:7
7   int c = a/b;
```

Figure 7: Sample source code that generates execution error due to division by zero and GDB output log.

for data in the source code itself) and dynamic mode (i.e., acting on execution logs).

Although the cases presented suggest a potential for using isolated strategies, techniques such as debug analysis and test cases may suffer from some limitations, for example, the challenge to differentiate two situations: *has a student become expert on a particular topic or has simply stopped using the feature that caused the error?* This kind of situation requires the strategy mechanism to be able to distinguish correct codes using certain programming features from codes that simply compiles and executes without errors but use improper solution subterfuges.

Considering the arithmetic exception identification shown in Figure 7 as an example: the absence of division by zero errors cannot indicate success if students do not perform any division operation in their source codes. *Debug analysis* strategy must consider a pre-evaluating step where the presence of such operations is validated, e.g., by searching through *regular expressions* or traversing a *parser* generated *AST*. Thus, featuring a hybrid system between these strategies. Hybrid associations were extensively employed in our method, the strategy *test cases* is present in all associations and was used as a determinant of functional correctness, ensuring any evidence identified by other strategies is guaranteed to belong to a functionally correct program. *Parser* and *AST* strategies were also a quite common association, being used to validate whether given programming resources were present or not in students' source codes.

The infinite loop problem: as a proof of concept, the infinite loops problem was chosen as a particular case to demonstrate the potential of our automatic learning evidence identification method. Multiple techniques were implemented to act together on evidence identification, showing hybrid systems can be applied even for complex problems. Infinite loops occurrence is a situation related to the Halting Problem, regarding the completion of a program in a finite time given an arbitrary input. The Halting Problem was introduced by Turing (1936) just as it was proved to be unsolvable, however, without a general solution, the search for evidence of halting condition is plausible in specific scenarios. A scenario is shown concentrating on two runtime aspects:

- *Execution timeout*: programs developed by students, such as classroom activities in introductory programming courses, often have short execution time. Thus, very long unfinished executions (e.g., excessively above reference solutions execution time) may be an indication of an infinite loop;
- *Loop iteration count*: monitoring programs' execution by counting iterations performed in

loop statements allows identifying evidence of halted executions. A repetition structure that iterates excessively above expected can be considered an indication of an infinite loop.

The first aspect concerns a *Software Metric* strategy. A host process is responsible for initializing and monitoring student program execution, specifically in this case managing the execution time and providing commands for killing the program's process if necessary. Execution time is a critical factor to consider, some programs can take long execution times (or even unpredictable) and still provide the correct result. However, in this scenario, exercises developed by students in introductory courses generally have a short execution. Thus, the execution time metric should consider a period long enough not to impair correct time-consuming operations, but not so long to impact the performance of the evidence detection system.

To identify an adequate maximum execution time for our method, a ten-round benchmark was executed on 84 C-Language exercises solutions[7]. The resulting average execution time was 2.35 milliseconds; no execution exceeded 3.2 milliseconds. Thus, the arbitrary value of 5000 milliseconds (5 seconds) was considered a secure value for our experiments since correct exercises (based on our references) securely cannot reach it. Students' programs that exceed the secure period are then killed and considered potentially halted (evidence of infinite loop)[8].

The time-based software metric employed in the first aspect provides a clue about the student program's behavior, however, it is not possible to know exactly why the execution did not end. There are situations where a program is not necessarily in an infinite loop situation but may have similar symptoms, as Figure 8 exemplifies. The presented source code wrongly asks for two input values, and test cases are configured to provide only one. In this case, the input stream ends, but the program still waits for data causing an execution halt and being mistakenly categorized as an infinite loop by automatic mechanisms such as the timeout strategy previously described.

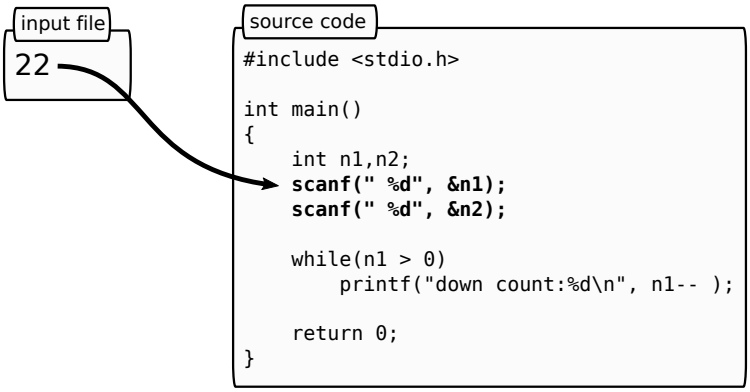

Figure 8: False-positive on infinite loop timeout strategy.

An alternative to avoid false positives on halt by input situations leads to the second aspect of assessment: identification of repetition loops and iterations counting. Iteration counting provides a hint that goes beyond program halt information, allowing loops behavior analysis from the program's execution to start the process termination (kill by timeout). Loop iteration

[7]Reference solutions relating to the seventh experiment mentioned in Section 4.

[8]Execution time is hardware dependent, a new benchmark is recommended when changing the execution environment.

counter implementation presented here is based on *Code Mutation* and *Execution Traces Analysis* strategies, executed as the following methodology:

1. Convert student code to an intermediary representation by applying a parser to generate an AST;
2. Traverse tree looking for C-Language repetition structures (consider *for*, *while* and *do-while* statements);
3. For each repetition structure, mutate the code by adding a global controller variable declaration (before and outside the loop) and an increment instruction (inside the loop);
4. Optionally: mutate the code by adding output print instructions inside the loop (permits analyzing execution traces through *stdout* logs);
5. Convert the AST intermediary representation into a compilable code (parser's reverse process);
6. Execute the mutated code inspecting the counter variables values. Timeout strategy can be applied to avoid getting stuck on halted processes. Mutated code execution can be done in two ways: (a) by running the program normally, capturing *stdout* output and searching for mutated print logs (e.g., with *regular expressions*); (b) by running the program through a debugger, adding breakpoints on control variables increment instructions, and inspecting memory to get variables valuation;
7. Apply a metric to classify *potential infinite loops*, e.g., student process was killed by timeout and at least one loop iterated more than a threshold value.

To exemplify our strategy, Figure 9 presents a sample source code and the corresponding mutated variation. In both original and mutated codes, the first repetition structure iterates forever while the second loop is never reached. The variable *i* on *while* statement should be decreased to avoid this condition. In the mutated code, it is possible to see the injection of two global control variables (*l_control_0* and *l_control_1*[9]). An increment instruction is added inside each loop, in this example associated with a standard output printing instruction (*printf*). These additions allow inspecting the program's runtime behavior both by collecting standard output or through a debugging process. Thus, a *potential infinite loop* metric can be applied.

To apply our strategy and inspect control variables valuation, the mutated program must be executed. Two conditions can be reached: the program's execution finishes in an adequate time (no infinite loop), and halting. When the second condition is reached, the halted process needs to be interrupted. Timeout logic can be applied to measure execution time and trigger process killing, however, this strategy is hardware dependent: faster computers can perform more iterations than slower ones in a determined period.

In most cases, hardware differences may be slight and irrelevant, but in favor of the method's invariance, an alternative strategy can be considered: analyzing *execution traces analysis* through

[9]Control variable names were simplified for this example, longer unique names are automatically generated to avoid override student variables.

source code

```
#include <stdio.h>

int main()
{
    int i = 10;

    while(i > 0)
    {
        printf("%d ", i);
        i++;
    }

    for(i=-10;i<0;i++)
    {
        printf("%d ", i);
    }

    return 0;
}
```

mutated code

```
#include <stdio.h>

int l_control_0 = 0;
int l_control_1 = 0;

int main()
{
    int i = 10;

    while (i > 0)
    {
        printf("l_control_0=%d\n", l_control_0++);
        printf("%d ", i);
        i++;
    }

    for (i = -10; i < 0; i++)
    {
        printf("l_control_1=%d\n", l_control_1++);
        printf("%d ", i);
    }

    return 0;
}
```

Figure 9: Code mutation example.

an automated debugging process. By using mutated source code information, the GDB debugger is configured to add breakpoints at each control variable increment instruction. A controlled debug process is then conducted to execute the mutated program until a certain number of breakpoints is reached (time-invariant metric). The breakpoints reached threshold corresponds to the number of loop iterations achieved in the whole program's execution.

As well as mentioned for the first aspect implementation, defining a "secure" threshold is challenging. We performed a manual analysis to find out the average and the maximum number of iterations executed in our reference solutions. Analyzing 32 source codes containing repetition loops, an average of 14.8 iterations per execution was identified, having 99 as the highest count. Thus, we chose to use the arbitrary value of 1000 iterations (about 10 times the highest value) as a threshold for distinguishing potential infinite loops. Applying our strategy in the mutated program, considering the established threshold, indicates the control variable *l_control_0* is probably inside an infinite loop, and *l_control_1* is never reached (0 iterations).

Given the strategy presented, two considerations are pointed out: (1) assessed program must be compilable and parseable, so *Compilation Analysis* and *Parser* are set as prerequisites; (2) a kill by timeout situation is also a prerequisite for the iterations count method, so programs that finish correctly are not penalized. Thus, as presented, the infinite loops case used multiple complementary strategies acting together, characterizing a *hybrid system*.

3.3 Learner Model

The strategies presented in the last section provide resources to identify evidence of programming skills. As the final step in our method, identified evidence are used to feed the learner model. For our research, the learner model is established to organize and present characteristics representing

skills, developed or not by students, necessary for the proper execution of programming tasks. Computer programming is an abstract domain where not all the skills required to be a good programmer are measurable quantitatively, e.g., there is no universal metric that can claim that someone has developed expertise in the "algorithm" concept.

Thus, measuring skills in the computer programming domain depends on elaborating a set of capacities to be used as a metric. In our context, the skill set presented in Figure 3 is employed. Characteristics of the knowledge representation mechanisms identified through our systematic literature review, as well as related research, were analyzed to define our learner model. Our analysis is described in the following.

Maschio (2013) already defined a skill visualization model in the form of an *overlay graph*, where the student's knowledge (the acquired skill subset) is highlighted over the entire domain (the whole skill set). Considering the automatic evidence identification method proposed in the present research, an analysis of Maschio's representation guided by features found on literature mechanisms was conducted and some improvements made are pointed out in the following.

Per source code detailed skill visualization: Distinguishing skills already developed from those that are yet to be worked on is essential for measuring student knowledge. The overlay graph permits to know (booleanly) if a particular skill has been developed or not, however, knowing details such as the exact programming exercise (or even more accurately the exact code snippet) where the student developed the skill can be of great value. The Heat Maps technique applied by Edmison & Edwards (2019) focuses on highlighting source code fragments. This feature inspired us to implement a complementary detailed log in which every identified evidence can point to a code snippet responsible for the skill assessment. Figure 10 shows an example log returned by our strategies to assess *functions*. It is possible to see the student's source code and a detailed log pointing to the identified evidence, the code snippet, and a function call analysis about parameters and return statement presence and correctness (e.g., void functions are not supposed to have a return).

source code

```
#include <stdio.h>

void print_year(int year)
{
   printf("hello time traveller, we are on %d.", year);
}

int main()
{
   int year;
   scanf(" %d", &year);
   print_year(year);
   return 0;
}
```

evidence evaluation result log

Evidence Found: function declaration (print_year)

```
void print_year(int year) {
    printf("hello time traveller, we are on %d.", year);
}
```

return type: void, **return provided**: false, **correct**: true
params count: 1
function call: print_year(year)
params provided: 1, correct: true

Evidence evaluation correct? true

Figure 10: Per source code detailed skill visualization.

Skill valuation on multiple source codes: Considering that students may submit several source codes to be assessed, identifying the same skill in multiple source codes is common, especially when the submitted source code set refers to a specific programming topic. In such a situation, the automatic system needs to decide which assessment to take into account (especially because results may be discordant). Koh et al. (2014) approach a similar situation: the maximum value of each skill is considered when multiple activities are selected. The maximum value metric is employed as the decision criterion on our learner model, so each skill is valuated with the higher result identified by strategies on the selected source code subset.

Timeline-based exercises subset selection: A selection mechanism is necessary as the model can be fed with information from multiple source codes. A method similar to that employed by Yamashita et al. (2017) was implemented: a timeline presents all source codes submitted by a given student and provides resources for selecting subsets. Timeline changes affect the valuation of the skills represented in the model.

Uncertainty treatment: Assessed aspects often have abstract nature, being challenging to accurately provide automatic assessment through source code analysis only. Students' are also unpredictable, sometimes following teachers' guidelines, and sometimes achieving solutions using uncommon programming resources and techniques. Unexpected behaviors can be challenging (and even impossible) to deal with in the automatic assessment environment – e.g., importing incompatible libraries, using unsupported/legacy syntax, submitting source code with incompatible encoding, and using operating system dependent resources. These issues make automatic assessment uncertain where accurate evaluation is not always possible. The learner model ideally should consider this limitation. According to Neapolitan (2003), Bayesian networks are graphical structures for representing relationships between variables (skills, in our context) and are capable of dealing with uncertainty by using probability theory.

Representing students' skills also requires considering the temporal aspect. Students may continuously submit new source codes along a course and evidence identified from different time intervals must be represented accordingly. The student model representation extends then to the concept of Dynamic Bayesian Networks[10] detailed by Neapolitan (2003). Thus, for the explored scenario, converting the graph model proposed by Maschio (2013) into a Dynamic Bayesian Network contributes to a better problem representation due to the network's capabilities to deal with an uncertainty environment and to deal with temporal changes.

Thus, the learner model adopted for our research is supported by the representation shown in Figure 11 and described as follows:

- Bayesian Network variables, also called "features" by Neapolitan (2003), model our full skill set. Skills are represented as *N1*, *N2*, and *N3* nodes;
- The edges represent direct influences between skills. These influences were extracted from Maschio (2013) where directed arrows represent relationships between skills (e.g., *A* indicates an analogy, *G* for generalization);
- Variable valuation (skill value) depends on evidence sets (e.g., the presence and correctness of an *if* conditional on student's code) resulting from automatic strategies application or

[10] Also known as Temporal Bayesian Networks.

probabilistically calculated by inference (described below). Any evidence of a variable is considered equally influential in the skill value calculation (e.g., "*switch*" and "*data type compat.*" in Figure 11 have equal weights).

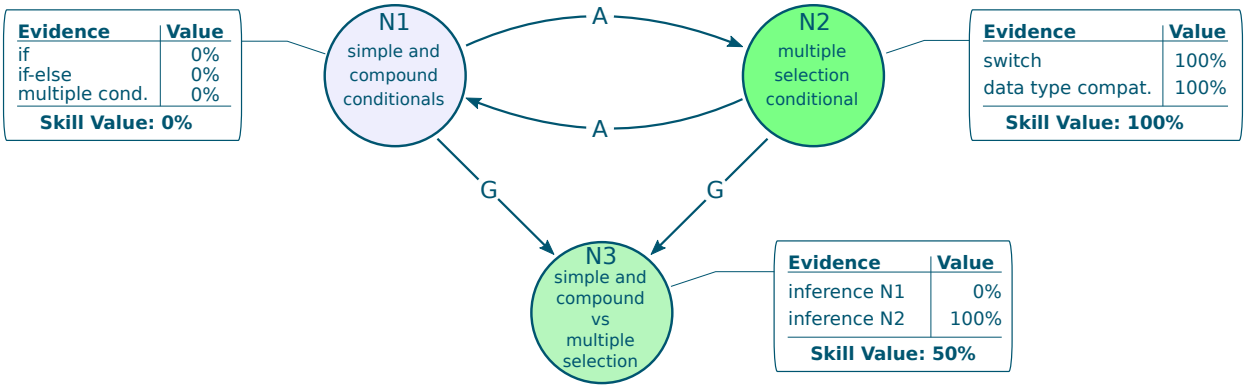

Figure 11: Dynamic Bayesian Network lerner model representation (subset extracted from conditionals-related skills).

Beyond the ability to handle uncertainties, variable valuation through inference is another useful resource provided by Bayesian Networks. The inference technique allows to propagate information from a variable through the network and use it as input for valuating subsequent dependent variables. A value propagation is exemplified in Figure 11, considering the valuation of three skills: (N1) *domain over simple and compound conditionals*; (N2) *domain over multiple selection structures*; (N3) *recognition of situations that favor using each structure*.

In Figure 11, the Bayesian Network considers dependencies between learner's skills, so the probability of success in (N3) has a strong connection with the predecessors (N1) and (N2). Thus, the evidence responsible for valuating skill (N3) assumes values from the predecessor nodes. Considering the node (N1) is not valuated, the node (N2) is fully valuated, and the evidence set has equal weights, (N3) valuation becomes 50%. Valuating network nodes by inference can be applied exclusively, where all evidence of a node are results of inferences, or even part of a hybrid mechanism, where an evidence of a node is the result of inferences while other evidence is identified by external automatic strategies.

As a final result, the student model can provide resources for representing students' skills in different perspectives: starting from a general view, where all skills are presented on the screen, followed by the visualization of the evidence used in each network node, and ending with the details of the code snippets where the evidence was identified. Figure 12 presents an example of visualization of these perspectives, emphasizing an evidence of using the multiplication arithmetic operator, related to the skill that concerns the use of *arithmetic expressions*.

4 Experiments and Results

Seven experiments were conducted to investigate different aspects of the A-Learn EvId method. Experiments were carried out both in controlled scenarios (source codes specifically designed for testing purposes) and in real scenarios (dataset built from real-world exercise solutions, formally

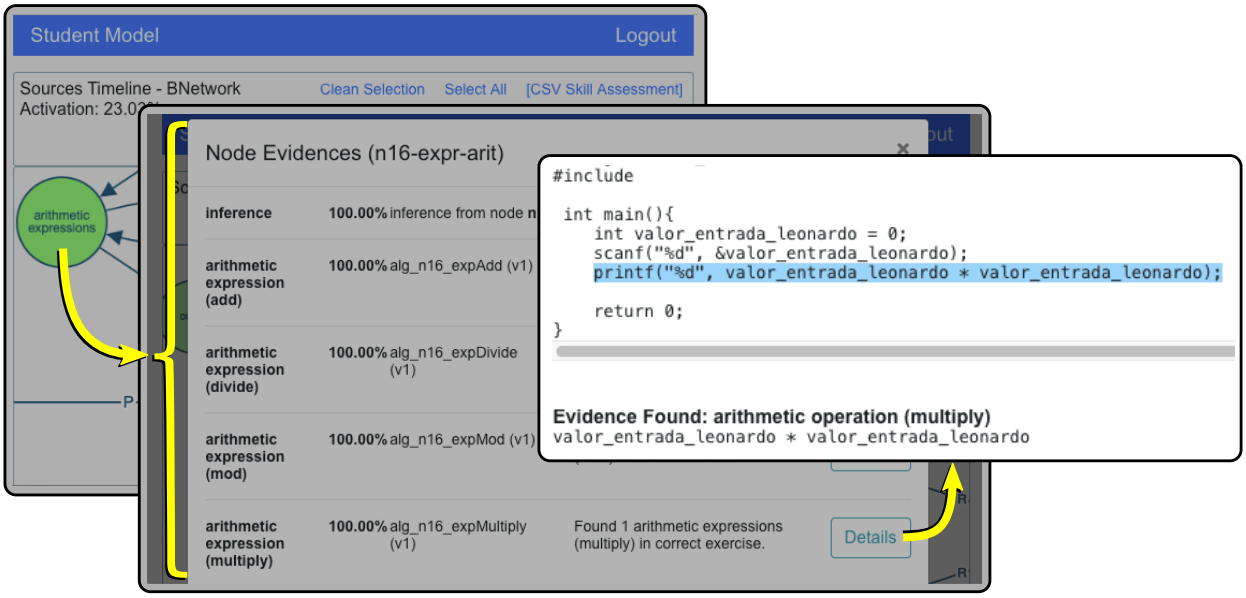

Figure 12: Different visualization perspectives in the Learner Model.

specified and collected). Table 4 summarizes our experiments according to the IMRaD format (Wu, 2011), where the structure is represented by four questions: **I**ntroduction – *why* experiments were elaborated? **M**ethod and Materials – *how* experiments were characterized? **R**esults – *what* have we found? **a**nd, **D**iscussion – *so what* does it mean? Dataset size (number of source codes) is shown in the first column. The complete data and discussion for each experiment can be found in Porfirio (2020).

In addition, two highlights are presented regarding the *Sixth* and *Seventh* experiments. The *Sixth* experiment specifically focused on analyzing student progress. Representing a more realistic environment, the experiment used a skill set based on topics commonly covered in real programming courses, showing that student progress can be monitored (e.g., between one exercise list and another), and that it is possible to identify when each skill start to be detectable. Figure 13 exemplifies a progress analysis where the vertical axis represents programming topics coverage and the horizontal axis represents ten exercise lists. The absence of learning evidence means the student did not submit a specific list of exercises.

Table 4: Experiments summary (IMRaD structure).

Experiment	**Why?**	**How?**	**What?**	**So What?**
First (Pilot Test) (1 source)	To investigate preliminary automatic source code analysis strategies.	*AST* and *parser* strategies were applied to a reference source code, results were then compared to human assessment.	*AST* and *parser* strategies can detect learning evidence of *constants* programming topic.	Strategies presented good results on reference source code. Application on student-made source codes still needed.

Experiment	Why?	How?	What?	So What?
Second (29 sources)	To investigate preliminary automatic source code analysis strategies in a real scenario.	*AST* and *parser* strategies were applied to student-made source codes, results were then compared to human assessment.	*AST* and *parser* strategies can detect learning evidence of *variables* and *constants* programming topics. Limitations were identified.	Experiment suggests automatic strategies can be feasible. Extended tests covering more programming topics and strategies still needed.
Third (Pilot Test) (29 sources)	To investigate static and dynamic automatic strategies to detect evidence of learning of input/output commands with different data types.	An experimental environment was built with four automatic strategies. An artificial dataset was employed. Results were then compared to human assessment.	92.39% of human cataloged learning evidence was also identified by automatic strategies. Implementation limitations were detected.	Static and dynamic approaches were successfully applied to detect evidence. Strategies worked well on extended programming topics set.
Fourth (113 sources)	To investigate whether strategies from the third experiment can also be accurate in a real scenario.	The third experiment was replicated with student-made source codes extracted from a real Intelligent Tutoring System.	Evidence identification capabilities in the real environment were observed to be similar to the controlled scenario.	Strategies were successfully applied for detecting learning evidence, but implementation limitations still exist.
Fifth (142 sources mixing data from Third and Fourth experiments)	To investigate using automatically identified evidence as data source for feeding the learner model.	A Dynamic Bayesian Network was fed with automatically detected evidence. An empiric analysis was conducted to detect changes in the model.	Student model changes according to evidence inserted in the network.	Detecting evidence from multiple source codes and filtering them in the network permits monitoring students' skills progress.
Sixth (3860 sources + 101 reference solutions)	To investigate evidence detection and students' progress monitoring considering skills commonly assessed in real programming courses.	A priority skill-set was established through syllabi analysis, exercise lists were applied to real students. Students' progresses were compared between exercise lists.	Student progress between exercise lists can be monitored and it was possible to identify when each skill began to be manifested.	Detecting (the lack of) progress can offer useful insights for both teachers and Intelligent Tutoring Systems as well as for students and their self-learning monitoring.
Seventh (4434 sources + 84 reference solutions)	To demonstrate skill-based assessment using automatic strategies as means of identifying functionally correct but conceptually incorrect solutions.	Desired skills were defined for each programming exercise and then compared to students automatically detected skills.	Skill-sets comparison indicated source codes that deviated from reference solutions.	Skill-based assessment proved to be a valuable resource for locating conceptually incorrect solutions built with subterfuges.

Finally, the *Seventh* experiment investigated the capabilities of skill-based assessment. Situations where subterfuges were used as means to achieve source code's functional correctness have been identified. Automatic search for learning evidence considering different programming skills has proved to be an interesting and effective method, providing indicative of potentially incorrect

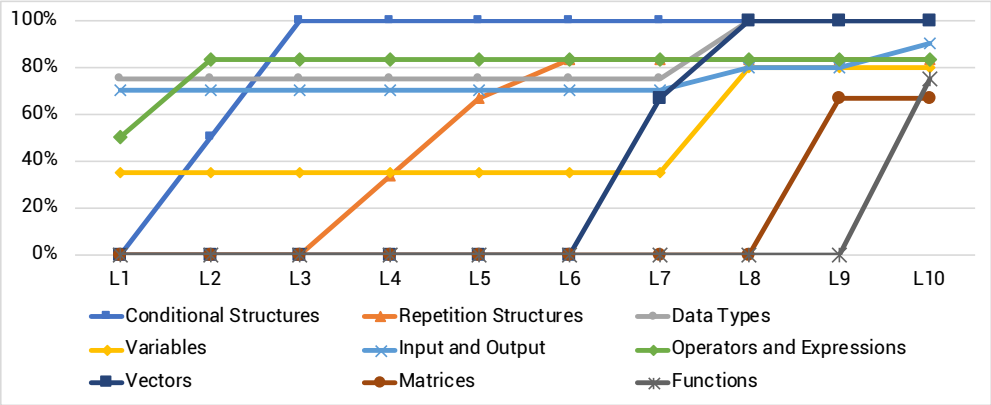

Figure 13: Example of a single student progress across the ten exercise lists.

solutions where manual assessment is required. Figure 14 exemplifies the experiment with a sample solution to the following problem: *read an integer vector and a floating-point vector, each with three positions. Subsequently, traverse the vectors with a single repeating loop and print the sets in parallel (Int1:Float1, Int2:Float2, Int3:Float3).* The assessment output points out the student did not employ any loop-related resource, solving the exercise in a *forced way* (with hard-coded vector indices). This type of solution would be accepted by simpler strategies such as *Test Cases*, but specifications of a desired skills-set (e.g., a higher valuation of loop-related concepts) prevents the solution from achieving a good score and indicates it for manual inspection.

student source

```
#include <stdio.h>

int main (){
    int ua[3];
    float f[3];
    int valor_1;

    scanf("%d %d %d %f %f %f",
          &ua[1], &ua[2], &ua[3],
          &f[1], &f[2], &f[3]);

    printf("%d:%.2f %d:%.2f %d:%.2f",
           ua[1],f[1],ua[2],
           f[2], ua[3], f[3]);

    return 0;
}
```

Assessment Output

Skill	Required (%)	Identified (%)	Skill Evaluation (%)
structuring and composition	100.00	100.00	100.00
effectuation	100.00	50.00	50.00
simple instructions	25.00	25.00	100.00
types of literals	25.00	25.00	100.00
variables	25.00	5.00	20.00
constants	75.00	8.33	11.11
variables vs constants	49.63	6.63	13.35
relational expressions	16.67	0.00	0.00
compound expressions	5.67	0.00	0.00
control structures	14.37	0.00	0.00
repetition structures	37.50	0.00	0.00
infinite loops	100.00	0.00	0.00
counted loops	50.00	0.00	0.00
conditional loops vs counted	25.00	0.00	0.00
arrays	66.67	66.67	100.00

Skill Based Assessment (Average): 32.96%

Figure 14: Source code skill-based assessment inspection (low score).

In another perspective, an excessively high score can also lead to identifying concept failures, where students employ very complex solutions deviating from teachers' goals – Figure 15 exemplifies this perspective. Initially, the student performed the initialization of two variables, *w* and *z*, and, right after, overwrote their values with the *scanf* data read command. Next, an additional unnecessary conditional were employed (*if* and *else* are already mutually exclusive). These flaws indicate the student, probably, used programming resources without fully understanding the concepts, especially with the *else* clause of the compound conditional.

student source

```
#include <stdio.h>

int main (){
   int w=0;
   int z=0;
   scanf("%d", &w);
   scanf("%d", &z);
   if(w==z){
      printf("sao iguais!");
   } else if(w !=z){
      printf("sao diferentes!");
   }
   return 0;
}
```

Assessment Output

Skill	Required (%)	Identified (%)	Skill Evaluation (%)
structuring and composition	100.00	100.00	100.00
effectuation	100.00	100.00	100.00
simple instructions	25.00	29.00	116.00
input	20.00	20.00	100.00
input vs output	10.00	10.00	100.00
types of literals	25.00	25.00	100.00
variables	5.00	25.00	500.00
constants	8.33	8.33	100.00
variables vs constants	6.63	16.63	250.94
relational expressions	16.67	33.33	200.00
compound expressions	5.67	11.33	200.00
control structures	8.74	17.49	200.00
conditional structures	16.50	33.00	200.00
simple and compound conditionals	33.33	66.67	200.00
simple and compound vs multiple selection	16.50	33.00	200.00
repetition structures	4.13	8.25	200.00

Skill Based Assessment (Average): 179.18%

Figure 15: Source code skill-based assessment inspection (high score.

5 Contributions

This doctoral thesis research produced conceptual, methodological and technical contributions. The main ones are highlighted as follows:

- A Systematic Literature Mapping (Porfirio, 2020) providing an updated and rigorous panorama of automatic programming source code assessment, presenting categorization schemes and important concepts, therefore informing this thesis and future research;
- A set of 37 standardized computer programming skills relevant to automatic identification;
- A set of 9 strategies, as well as their implementation and assessment, responsible for automatic identifying skill evidence from students' source codes;
- A learner model capable of representing students' knowledge (skills acquired) identified by the automatic strategies and monitoring knowledge evolution;
- The A-Learn EvId: Automatic Learning Evidence Identification method;
- Seven source code datasets totalling 8651 C-Language programs (8436 from real-world exercise solutions, formally specified and collected; 215 specifically designed for testing purposes), as well as method and tools sharing[11];
- Publications: Porfirio et al. (2016); Porfirio et al. (2017); Porfirio et al. (2018); Porfirio et al. (2020) (national award).

[11] Supplementary data, such as datasets and tools, can be found online at: https://bit.ly/doc_ctd.

6 Conclusion

Considering the central problem for this research, the challenge to provide assessment of the students' source codes in a continuous and timely manner and to identify the manifestation of new skills, the A-Learn EvId method was proposed and evaluated. Seven experiments were conducted, covering different aspects of the method, from the investigation of the automatic strategies capabilities to the representation of results in the student model. Results showed our method is promising, being able to automatically assess students' source codes, identifying multiple programming skills. Automatic strategies results were represented through our learner model, which provided resources for monitoring students' progress. The Dynamic Bayesian Network model showed up adequate to the exposed scenario, as it was able to represent the uncertainty environment generated by the automatic assessment, as well as to provide important resources for valuating skills by inference and monitoring students' progress. Therefore, experiments' results suggest that high-order cognitive skills can be automatically assessed in the computer programming context. However, efforts and research expansion are still needed to improve method's accuracy and reliability.

This research was built on the results identified in the literature and extends the state of the art by identifying more skills and applying more strategies, as well as hybrid systems, which are capable of working on complex problems such as the identification of infinite loop evidence. Also, the proposal, implementation, and demonstration of using automated strategies as a means for high-level, skill-based, assessment can be seen as positive impacts over the existing methods, especially when employing the resulting information to monitor the progress of students skills and detect potential concept flaws. Lastly, the contributions extend to the general context of Computer Science, where the acquisition of programming skills is a crucial activity for the vast majority of professionals. Thus, the development of resources that support the teaching of this activity tends to bring benefits and improve this process.

The A-Learn EvId method was applied to the computers programming domain, where investigating new strategies, elaborating new test scenarios, and performing application in real tutoring systems (and courses) can contribute to the evolution and improvement of the method. Still considering the context of computer programming, future research may refine the learner model, stressing the identification of evidence on relevant topics according to each course, methodology, or even teachers' preferences. Extending the learner model to include the object-oriented paradigm and making the method compatible with other programming languages are also possibilities to be explored. Intelligent Tutoring systems can also benefit from the contributions of our research by offering on-the-fly feedback and suggestions to students according to their mistakes and progress. Also, possibilities of generalization to other domains are pointed out. Theoretically, the method can be implemented for any domain of which automatic strategies are feasible, by defining input data, defining automatic strategies based on a skill-set relevant for the domain, and defining a learner model, also based on the chosen skill-set. Thus, future research can investigate possible applications and contributions of the method to other areas.

Acknowledgements

We dedicate this paper to professor Alexandre Direne (*in memoriam*), the initial advisor of this research. We thank the Andres J. Porfirio's Thesis Defence Committee members, professors Alexander Robert Kutzke, Andrey Ricardo Pimentel, Avanilde Kemczinski, Marcos Alexandre Castilho, and Patricia Augustin Jaques Maillard, for the critical suggestions and appreciation of this research. We also thank the RBIE Editorial Board for the invitation to submit the extended and revised version of this paper. The authors also thank their Universities, and the anonymous reviewers who contributed to this research.

Extended Awarded Article

This publication is an extended version of the best doctoral thesis award winner in the Alexandre Direne Contest for Theses, Dissertations and Undergraduate Work in Computers in Education (CTD-IE 2020), entitled *"Identifying Evidences of Computer Programming Skills Through Automatic Source Code Evaluation"*, DOI: cbie.wcbie.2020.01.

References

Ahmed, U. Z., Kumar, P., Karkare, A., Kar, P., & Gulwani, S. (2018). Compilation error repair: For the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training* (pp. 78–87). New York, NY, USA: ACM. doi: 10.1145/3183377.3183383 [GS Search]

Cui, B., Li, J., Guo, T., Wang, J., & Ma, D. (2010, Oct). Code comparison system based on abstract syntax tree. In *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)* (p. 668-673). doi: 10.1109/ICBNMT.2010.5705174 [GS Search]

Edmison, B., & Edwards, S. H. (2019). Experiences using heat maps to help students find their bugs: Problems and solutions. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (pp. 260–266). New York, NY, USA: ACM. doi: 10.1145/3287324.3287474 [GS Search]

Gerdt, P., & Sajaniemi, J. (2006, June). A web-based service for the automatic detection of roles of variables. *SIGCSE Bull.*, *38*(3), 178–182. doi: 10.1145/1140123.1140172 [GS Search]

Hettiarachchi, E., Huertas, M., & Mor, E. (2013, 12). Skill and knowledge e-assessment: A review of the state of the art. *IN3 Working Paper Series*. doi: 10.7238/in3wps.v0i0.1958 [GS Search]

Hettiarachchi, E., Huertas, M., & Mor, E. (2015, 01). E-assessment system for skill and knowledge assessment in computer engineering education. *International Journal of Engineering Education*, *31*, 529–540. [GS Search]

Ihantola, P., Ahoniemi, T., Karavirta, V., & Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (pp. 86–93). New York, NY, USA: ACM. doi: 10.1145/1930464.1930480 [GS Search]

Jackson, D., & Usher, M. (1997). Grading student programs using assyst. In *Proceedings of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education* (pp. 335–339). New York, NY, USA: ACM. doi: 10.1145/268084.268210 [GS Search]

Kautzmann, T., & Jaques, P. (2020). Modelo de identificação de unidades de conhecimento de programação em processo de aplicação durante a codificação. In *Anais do XXXI Simpósio Brasileiro de Informática na Educação* (pp. 982–991). Porto Alegre, RS, Brasil: SBC. doi: 10.5753/cbie.sbie.2020.982 [GS Search]

Koh, K. H., Nickerson, H., Basawapatna, A., & Repenning, A. (2014). Early validation of computational thinking pattern analysis. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education* (pp. 213–218). New York, NY, USA: ACM. doi: 10.1145/2591708.2591724 [GS Search]

Liang, Y., Liu, Q., Xu, J., & Wang, D. (2009, Dec). The recent development of automated programming assessment. In *2009 International Conference on Computational Intelligence and Software Engineering* (p. 1-5). doi: 10.1109/CISE.2009.5365307 [GS Search]

Maschio, E. (2013). *Modelagem do processo de aquisição de conhecimento apoiado por ambientes inteligentes.* Tese de doutorado, Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná (UFPR). [GS Search]

Morris, D. S. (2003, Nov). Automatic grading of student's programming assignments: an interactive process and suite of programs. In *33rd Annual Frontiers in Education, 2003. FIE 2003.* (Vol. 3, p. S3F-1). doi: 10.1109/FIE.2003.1265998 [GS Search]

Neapolitan, R. E. (2003). *Learning bayesian networks*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. [GS Search]

Pimentel, M., & Carvalho, F. d. S. P. (2020, 05). Princípios da educação online: para sua aula não ficar massiva nem maçante! *SBC Horizontes*. SBC Horizontes, maio 2020. ISSN 2175-9235. Disponível em: http://horizontes.sbc.org.br/index.php/2020/05/23/principios-educacao-online. [GS Search]

Porfirio, A. J. (2020). *Identifying evidences of computer programming skills through automatic source code evaluation.* Phd thesis, Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná (UFPR). [GS Search]

Porfirio, A. J., Maschio, E., & Direne, A. (2016). Modelagem genérica de aprendizes com Ênfase em erros na aquisição de habilidades em programação de computadores. *Anais dos Workshops do Congresso Brasileiro de Informática na Educação*. doi: 10.5753/cbie.wcbie.2016.1198 [GS Search]

Porfirio, A. J., Pereira, R., & Maschio, E. (2017). Atualização do modelo do aprendiz de programação de computadores com o uso de parser ast. *Anais dos Workshops do Congresso Brasileiro de Informática na Educação*, *6*(1), 1121. doi: 10.5753/cbie.wcbie.2017.1121 [GS Search]

Porfirio, A. J., Pereira, R., & Maschio, E. (2018). Inferência de conhecimento a partir da detecção automática de evidências no domínio da programação de computadores. *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação - SBIE)*, *29*(1), 1553. doi: 10.5753/cbie.sbie.2018.1553 [GS Search]

Porfirio, A. J., Pereira, R., & Maschio, E. (2020). Identifying evidences of computer programming skills through automatic source code evaluation. In *Anais dos Workshops do IX Congresso Brasileiro de Informática na Educação* (pp. 01–10). Porto Alegre, RS, Brasil: SBC. doi: 10.5753/cbie.wcbie.2020.01 [GS Search]

Rahman, K. A., & Nordin, M. J. (2007). A review on the static analysis approach in the automated programming assessment systems. In *Proceedings of National Conference on Programming 07*. [GS Search]

Rajala, T., Kaila, E., Lindén, R., Kurvinen, E., Lokkila, E., Laakso, M.-J., & Salakoski, T. (2016). Automatically assessed electronic exams in programming courses. In *Proceedings of the Australasian Computer Science Week Multiconference* (pp. 11:1–11:8). New York, NY, USA: ACM. doi: 10.1145/2843043.2843062 [GS Search]

Souza, D. M., Felizardo, K. R., & Barbosa, E. F. (2016, April). A systematic literature review of assessment tools for programming assignments. In *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)* (p. 147-156). doi: 10.1109/CSEET.2016.48 [GS Search]

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, *2*(42), 230–265. [GS Search]

Ullah, Z., Lajis, A., Jamjoom, M., Altalhi, A., Al-Ghamdi, A., & Saleem, F. (2018). The effect of automatic assessment on novice programming: Strengths and limitations of existing systems. *Computer Applications in Engineering Education*, *26*(6), 2328–2341. [GS Search]

VanPatten, B., & Williams, J. (2015). *Theories in second language acquisition: An introduction* (Second ed.). Routledge. [GS Search]

Wilcox, T. R., Davis, A. M., & Tindall, M. H. (1976, November). The design and implementation of a table driven, interactive diagnostic programming system. *Commun. ACM*, *19*(11), 609–616. doi: 10.1145/360363.360367 [GS Search]

Wu, J. (2011, 12). Improving the writing of research papers: IMRAD and beyond. *Landscape Ecology*, *26*. doi: 10.1007/s10980-011-9674-3 [GS Search]

Yamashita, K., Sugiyama, T., Kogure, S., Noguchi, Y., Konishi, T., & Itoh, Y. (2017). An educational support system based on automatic impasse detection in programming exercises. In *Proceedings of the 25th International Conference on Computers in Education, ICCE 2017 - Main Conference Proceedings* (p. 288-295). [GS Search]