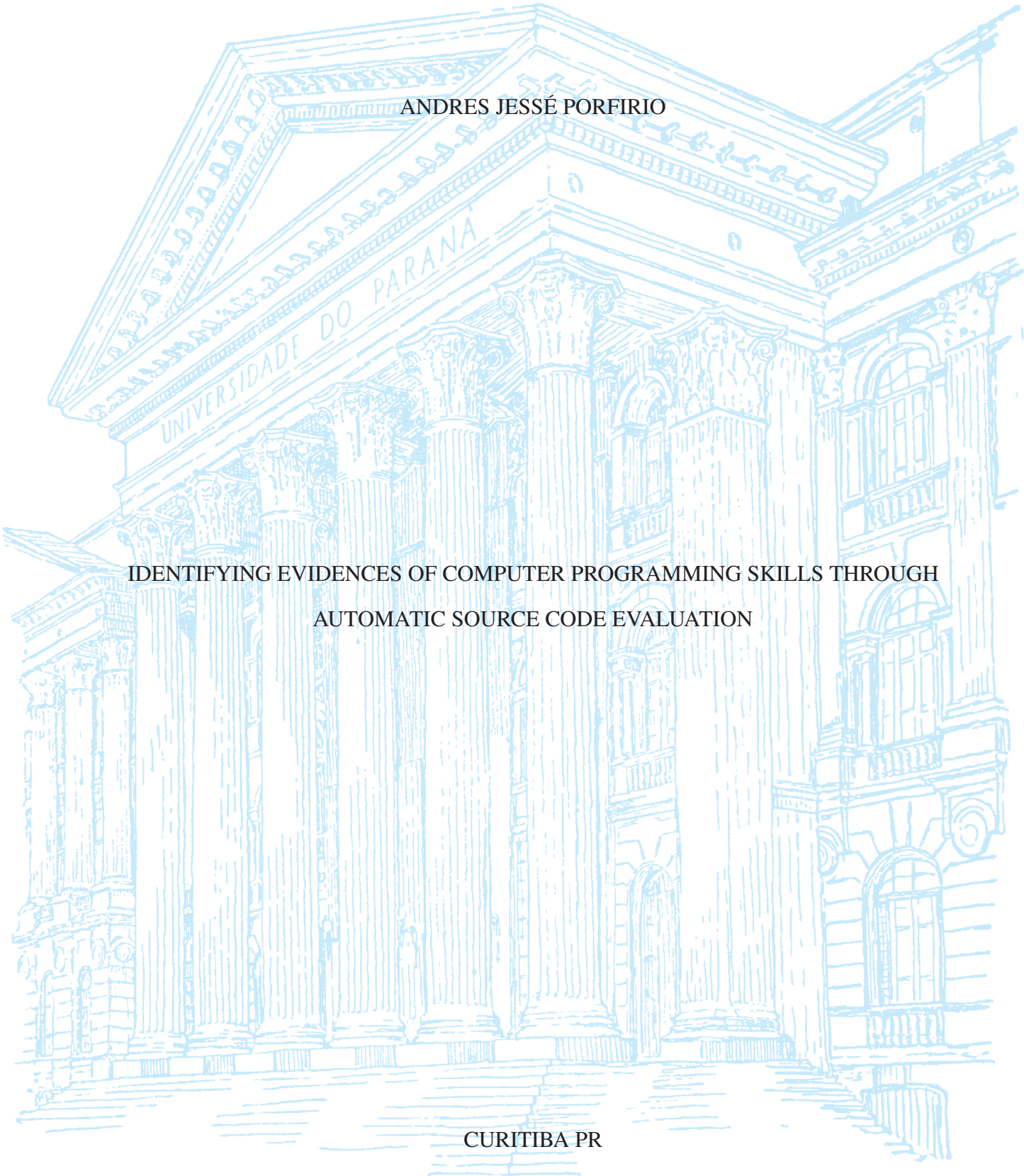
UNIVERSIDADE FEDERAL DO PARANÁ

ANDRES JESSÉ PORFIRIO



IDENTIFYING EVIDENCES OF COMPUTER PROGRAMMING SKILLS THROUGH AUTOMATIC SOURCE CODE EVALUATION

CURITIBA PR

2020

ANDRES JESSÉ PORFIRIO

IDENTIFYING EVIDENCES OF COMPUTER PROGRAMMING SKILLS THROUGH AUTOMATIC SOURCE CODE EVALUATION

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Roberto Pereira.

Coorientador: Eleandro Maschio.

CURITIBA PR

2020

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

P835i Porfirio, Andres Jessé
Identifying evidences of computer programming skills through automatic source code evaluation [recurso eletrônico] / Andres Jessé Porfirio. – Curitiba, 2020.

Tese - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2020.

Orientador: Roberto Pereira – Coorientador: Eleandro Maschio Krynski.

1. Programação (Computadores). 2. Código fonte (Ciência da Computação). 3. Algorítimo . I. Universidade Federal do Paraná. II. Pereira, Roberto. III. Krynski, Eleandro Maschio. IV. Título.

CDD: 005.131

Bibliotecário: Elias Barbosa da Silva CRB-9/1894

MINISTÉRIO DA EDUCAÇÃO
SETOR DE CIENCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA - 40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **ANDRES JESSÉ PORFIRIO** intitulada: **IDENTIFYING EVIDENCES OF COMPUTER PROGRAMMING SKILLS THROUGH AUTOMATIC SOURCE CODE EVALUATION**, sob orientação do Prof. Dr. ROBERTO PEREIRA, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVADO no rito de defesa.

A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 27 de Março de 2020.

ROBERTO PEREIRA
Presidente da Banca Examinadora

ELEANDRO MASCHIO KRYNSKI
Coorientador (UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ)

ANDREY RICARDO PIMENTEL
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

PATRICIA AUGUSTIN JAQUES MAILLARD
Avaliador Externo (UNIVERSIDADE DO VALE DO RIO DOS SINOS)

MARCOS ALEXANDRE CASTILHO
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

DocuSigned by:
Avanilde Kemczinski
9E588F92193F4EC...
AVANILDE KEMCZINSKI
Avaliador Externo (UNIVERSIDADE DO ESTADO DE SANTA CATARINA)

ALEXANDER ROBERT KUTZKE
Avaliador Externo (UNIVERSIDADE FEDERAL DO PARANÁ)

Esta tese é dedicada ao meu pai Miguel Porfirio Sobrinho (in memoriam). Jamais esquecerei suas últimas palavras, "boa sorte, tchau.", quando se referia à minha apresentação da qualificação do doutorado. Infelizmente o destino não me permitiu lhe dar a boa notícia da minha aprovação.

ACKNOWLEDGEMENTS

Dear reader, I ask your permission to write some words in my native language, Portuguese.

Agradeço aos meus pais, Miguel (in memoriam) e Noêmia, que sempre me incentivaram à traçar um caminho levando o estudo muito a sério, esse ensinamento será carregado para o resto da vida e com certeza será passado a diante;

à minha esposa Ana, que não mediu esforços para me dar o suporte necessário para que conseguisse atingir os objetivos. Foram muitos os momentos de trabalho árduo e incansável que suas palavras me tranquilizaram e mostraram que tudo se resolveria e valeria a pena. Amo você, esse doutorado não foi só meu, foi nosso! Deixo também um agradecimento especial à família que acolheu e sempre me apoiou, Cleides, Mauro, Orlando, e meus sobrinhos João e Lorena. Vocês foram fundamentais nessa conquista;

aos meus orientadores, Alexandre (in memoriam), Roberto e Eleandro. Não foram poucas as vezes que me deparei com situações aparentemente sem solução, mas seus sábios conselhos me encaminharam na direção certa. Sinto que meu tempo de convívio com o professor Alexandre infelizmente foi curto, mas ficaram lições que levarei para toda a vida. Professor Roberto, contigo aprendi a ser crítico e passei a adotar metodologias mais rígidas, me espelho muito no excelente pesquisador que você é. Professor Eleandro, quando fui seu aluno na graduação já sabia da sua extrema capacidade, fiquei extremamente grato por ter tido a oportunidade de manter essa parceria também no doutorado;

aos membros das bancas examinadoras de qualificação e defesa, que cordialmente apontaram e discutiram as fragilidades do meu trabalho, e assim permitiram que fosse melhorado. Sinto-me privilegiado de ter tido pesquisadores de tão alto nível contribuindo com ideias para o meu trabalho; A robustez e a força do aço vêm das condições extremas em que ele foi forjado;

aos demais professores, o mundo não é nada sem mestres! Agradeço imensamente todos os ensinamentos das pessoas que passaram pelo meu caminho. Foi uma experiência incrível voltar a ser aluno após tantos anos lecionando, o doutorado me proporcionou conhecer professores e pesquisadores brilhantes, que sempre serão exemplos para mim;

aos amigos, que sempre estiveram por perto me apoiando e garantindo que o convívio social e humano permanecesse. O trabalho é muito importante, mas também precisamos viver ao livre e limpar a mente de vez em quando. Deixo um agradecimento especial aos meus amigos Leonildo e Rafael, que contribuíram com revisões desta tese. Também aos colegas de estudo, Marcelino e Rodrigo, que sofreram juntos nas disciplinas (com muitas recorrências!) mas que foram fundamentais para eu conseguisse sucesso nessa etapa. Não deixemos que o tempo e as distâncias apaguem da memória os bons momentos que vivemos;

aos colegas professores e funcionários da UTFPR. O apoio institucional, a organização e colaboração de todos foi fundamental para a condução do doutorado. Deixo um agradecimento especial ao professor Fábio, que prontamente se dispôs a aplicar minhas ferramentas em sala de aula e assim contribuiu para a execução deste trabalho;

aos alunos das disciplinas introdutórias de programação da UFPR e UTFPR, onde as bases de dados foram coletadas. Agradeço o empenho na resolução dos exercícios de sala de aula, que foram essenciais para a elaboração dos meus experimentos.

Peço desculpas às pessoas que não foram mencionadas explicitamente, todos os que de alguma forma contribuíram estarão guardados para sempre em minha memória.

RESUMO

Esta tese é contextualizada no ensino de programação de computadores em cursos de Computação e investiga aspectos e estratégias para avaliação automática e contínua de códigos fonte desenvolvidos pelos alunos. O estado da arte foi identificado por meio de revisão sistemática de literatura e revelou que as pesquisas anteriores tendem a realizar avaliações baseadas em aspectos técnicos de códigos fonte, como a avaliação de corretude funcional e a detecção de erros. Avaliações baseadas em habilidades, por outro lado, são pouco exploradas e possuem potencial para fornecer detalhes a respeito de habilidades representadas por conceitos de alto nível, como desvios condicionais e estruturas de repetição. Um método de identificação automática de evidências de aprendizado é então proposto como uma abordagem baseada em habilidades para a avaliação automática de códigos fonte de programação. O método é caracterizado pela implementação de diferentes estratégias para avaliação de códigos fonte, identificação de evidências de habilidades de programação, e representação destas habilidades em um modelo do aluno. Experimentos realizados em ambientes controlados (bases de dados artificiais) mostraram que estratégias automáticas de avaliação de código fonte são viáveis. Experimentos conduzidos em ambientes reais (códigos fonte produzidos por alunos) produziram resultados semelhantes aos ambientes controlados, entretanto revelaram limitações relacionadas à implementação das estratégias, como vulnerabilidades à sintaxes inesperadas e falhas em expressões regulares. Um conjunto de habilidades foi selecionado para compor o modelo do aluno, representado por uma rede bayesiana dinâmica. Por meio de experimentos foi demonstrado que a alimentação do modelo com evidências resultantes da avaliação automática de códigos fonte permite o acompanhamento do progresso das habilidades dos alunos. Finalmente, as estratégias automáticas em conjunto com os recursos do modelo do aluno permitiram a demonstração da avaliação baseada em habilidades, que se mostrou um recurso valioso para identificação de soluções funcionalmente corretas, porém conceitualmente incorretas; quando o programa é funcionalmente correto, retornando resultados esperados à determinadas entradas, porém foi construído com recursos e conceitos incorretos.

Palavras-chave: Programação de Computadores, Avaliação Automática, Avaliação Baseada em Habilidades

ABSTRACT

This thesis is contextualized in the teaching of computer programming in Computing courses and investigates aspects and strategies for automatic and continuous evaluation of student developed source codes. The state of the art was identified through systematic literature review and revealed previous research tends to perform evaluations based on source codes technical aspects, such as functional correctness assessment and error detection. Skills-based assessments, in turn, are less explored although having potential to provide details of skills represented by high-level concepts, such as conditionals and repetition structures. A method for automatic identification of learning evidences is then proposed as a skills-based approach to automatic evaluation of programming source codes. The method is characterized by implementing different strategies for source code evaluation, identifying evidences of programming skills, and representing these skills in a student model. Experiments conducted in controlled scenarios (testing datasets) have shown automatic source code evaluation strategies are viable. Experiments conducted in real scenarios (student-made source codes) produced results similar to controlled scenarios, however, implementation-related limitations were revealed for some strategies, such as vulnerabilities to unexpected syntax and flaws in regular expressions. A skill set was selected to compose our student model, represented by a Dynamic Bayesian Network. Experiments have shown feeding the model with evidences resulting from source codes automatic evaluation allows monitoring students' skills progress. Finally, automatic strategies coupled with student model capabilities enabled demonstrating skills-based assessment, which showed a valuable resource for identifying functionally correct source codes, but conceptually incorrect; when a program is correct functionally, returning expected results to specific inputs, but it was built with erroneous concepts and resources.

Keywords: Computer Programming, Automatic Evaluation, Skills-Based Assessment

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ACM	Association for Computing Machinery
ALGOL	Algorithmic Language
API	Application Programming Interface
AST	Abstract Syntax Tree
CAP	Code Analyzer for Pascal
CEIE	Comissão Especial de Informática na Educação
CC	Cyclomatic Complexity
CE	Criteria for Exclusion
CI	Criteria for Inclusion
CPU	Central Process Unit
CSV	Comma-Separated Values
DSL	Domain Specific Language
GCC	GNU Compiler Collection
GDB	GNU Project Debugger
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ID	Identifier
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
ITS	Intelligent Tutoring System
JSON	JavaScript Object Notation
LOC	Lines of Code
MD5	Message-Digest Algorithm
NOC	Number of Classes
OCaml	Objective Caml
OSSL	Output Semantic-Similarity Language
PDF	Portable Document Format
PDG	Program Dependence Graph
PL/I	Programming Language One
REST	Representational State Transfer
RQ	Research Question
RUF	Ranking Universitário Folha
SciELO	Scientific Electronic Library Online
SQL	Structured Query Language
UFAM	University of Amazonas

UFCG	University of Campina Grande
UFG	University of Goiás
UFMG	University of Minas Gerais
UFPA	University of Pará
UFPE	University of Pernambuco
UFRGS	University of Rio Grande do Sul
UFRJ	University of Rio de Janeiro
UFSC	University of Santa Catarina
UnB	University of Brasília
UTFPR	Universidade Tecnológica Federal do Paraná
URL	Uniform Resource Locator
VHDL	VHSIC Hardware Description Language

CONTENTS

1 INTRODUCTION

Computer programming is one of the initial topics in Computer Science courses and, sometimes, one of the most complex from students' point of view (Ullah et al., 2018). Learning programming concepts introduces students to a completely abstract and difficult to assimilate world, but essential for the development of skills required in the field.

Research and Development of computer programming teaching support tools is a widespread topic in the literature. Tools to support programming teaching and learning typically provide resources focused on the needs of both students and teachers. In teachers scope, two of the major difficulties faced are evaluating individual programming exercises (Souza et al., 2016), and providing individualized timely feedback (Ihantola et al., 2010; Ullah et al., 2018). Correcting large amounts of source codes developed by a large number of students makes programming skills assessment a complex and exhausting task for teachers (Ullah et al., 2018; Rahman and Nordin, 2007). As a result, investigating methods and developing support tools is necessary to improve the teaching process.

Although the issue has come under investigation for decades (Liang et al., 2009; Rahman and Nordin, 2007; Souza et al., 2016; Ullah et al., 2018), identifying learning evidence based on automatic source code evaluation is still a challenge. Several source code aspects can be evaluated with different strategies (Souza et al., 2016), not always automatically possible. Also, this diversity of aspects leads to a disperse literature, where numerous methodologies are applied to problem-specific scenarios.

From systematic literature review, it was found many works dealing with automatic evaluation, but attempts to do this with a conceptual focus are rare as the majority focuses only on technical aspects such as functional correctness (Jackson and Usher, 1997; Morris, 2003) and error detection (Wilcox et al., 1976; Suarez and Sison, 2008; Bahlke and Snelting, 1986; Jadud and Dorn, 2015; Ahmed et al., 2018). Initial programming courses, however, have their syllabi focused on concepts and desired skills, not technical aspects, which are often conveyed through courseware, and evaluated in specific situations where students are supposed to succeed only if they have mastered certain programming resources usage.

In this thesis' context, the definition of skill is grounded in DeKeyser's skill acquisition theory (VanPatten and 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. Similarly, in the computer programming context, we assume that students acquire knowledge through the learning of concepts, and later manifest it by applying different programming resources in source codes. In our context, behavioral changes are marked by using previously unreported programming resources, thus suggesting evidence of new skills acquisition.

When it comes to automatic evaluation, assuming students have developed a skill if they have succeeded in a particular activity is valid in some scenarios, but, ideally, an evaluation system should refer directly to the concepts taught. (Hettiarachchi et al., 2013) present two types of assessment: knowledge-based and skills-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, knowledge-based assessment is related to simple questions and rarely give any insight into the thought process students used to elaborate their responses, while skills-based

assessment can be applied to evaluate cognitive skills and practical abilities (Hettiarachchi et al., 2015).

To exemplify skills-based assessment in computers programming context, consider the assignment shown in Figure 1.1, where students are asked to write a code able to receive an undetermined number of inputs. By manually analyzing the problem, it is possible to identify its solution requires capacities to deal with conditionals, repetition structures, input, and output commands.

Problem: Write a program that performs the reading of positive integer values until the user enters a value less than or equal to zero. At the end print the largest value entered.

source code

```
#include <stdio.h>

int main()
{
    int input, max=0;

    do
    {
        scanf(" %d", &input);
        if(input > max) max = input;
    } while (input > 0);

    printf("max value: %d \n", max);

    return 0;
}
```

Figure 1.1: Sample problem.

Taking advantage of source code evaluation strategies found in literature, several aspects can be automatically assessed on the solution presented in Figure 1.1, such as the absence of syntactic errors, the program correctness through compilation and execution processes, catching execution exceptions, among others. Considering a hypothetical automatic assessment system based on these strategies, Table 1.1 demonstrates a comparison with a skills-based manual analysis where it is possible to note different evaluation techniques allow different interpretations.

Although both approaches can evaluate students' programs, as it was exposed, technical aspects are not directly related to programming fundamental skills, requiring manual association, e.g., the mentioned sample code can be part of a repetition structures classroom test, where a percent of students' grading depends on program's functional correctness. In this case, the connection between the evaluated technical aspect with the repetition structures skill is imposed by the context where the activity was applied. Thus, it is assumed students understood a concept because were able to present correct solutions to a particular problem. Ideally, considering skill-based assessment, this kind of association could be pointed directly by the evaluation method, permitting any arbitrary source code produced by students could be considered a source of learning evidence for multiple programming skills.

Considering Hettiarachchi's descriptions (Hettiarachchi et al., 2013, 2015), assessing technical aspects is analogous to knowledge-based assessment. Table 1.1 technical assessment results provide an overview of some programming-related facts: the code contains (or not) syntax errors; the program is functionally correct (true/false); and, the program is vulnerable (or not) to certain unexpected situations. However, although useful on certain scopes, technical strategies shown in Table 1.1 can be considered simple and capable of providing only generic clues about students' knowledge. Skills-based assessment, however, aims to provide clues about students' high-order cognitive skills, representing small code units related to specific programming skills, such as variables, conditionals, repetition structures and matrices. Evaluating those specific code

Table 1.1: Technical vs skill-based assessment comparison.

Technical	**Skill-Based**	**Possible Interpretation**
Syntactic Error Check		Verify if students can write syntactically correct source codes.
Functional Correctness		Verify if students' programs can produce correct outputs given determined input-sets.
Exception Catching		Verify students' programs are prepared to unexpected behaviors (e.g., by inserting an wrong data type, such as a string instead of a number).
	Conditionals	Verify if students understand the concept of conditionals and correctly apply it in source codes to ensure programs' ending when a less than or equal to zero value is entered.
	Repetition Structures	Verify if students understand the concept of repetition structures and correctly apply it in source codes to guarantee multiple value inputs.
	Input and Output	Verify if students understand the concepts of input and outputs, and correctly apply them in source codes to read data and print the largest value entered.

units individually provides richer insights about students' reasoning and code construction details when compared to simpler and generic assessments.

Therefore, when automated, evaluation strategies focused on technical aspects are valuable to support student activities correction. However, they do not yet faithfully represent a conceptual view of the solution adopted and, according to (Hettiarachchi et al., 2015) definition, do not provide clues about the thought process students used to achieve their solutions. According to our systematic literature review, until the writing of this thesis, automatic assessment focusing on programming skills identification is a challenging and open problem, justifying the need for further studies on this topic.

1.1 RESEARCH PROBLEM AND OBJECTIVES

By delimiting the research scope, this thesis focuses on exercises from computer programming courses. The central problem addressed in this thesis is contextualized in the intersection between two activities: (1) automatically evaluating students' exercises; and (2) monitoring students skills development.

Automatic techniques can support teachers in exercise evaluations, reducing their workload. When it comes to monitoring the skills development progression, high workload and unfeasible individualized tutoring are also noted. Therefore, research for automatic assessment strategies and methods to facilitate tracking student progress are considered relevant for improving the teaching process. As a consequence, automated environments can improve students autonomy, allowing simple tests (e.g., exercise correctness check) to be directly resolved through tool-based interaction, with only the most punctual doubts and/or advanced concepts being reported to the teacher.

Considering the aforementioned activities, the main research questions that motivate this thesis are:

- *TRQ1: Can be high-order cognitive skills automatically evaluated in the computer programming context?*
- *TRQ2: Can automatically identified high-order cognitive skills be used for monitoring students' progress?*

Thus, the main objective of this thesis is to investigate a method for the automatic and continuous evaluation 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 review;
- Identify programming skills candidate to automatic evaluation;
- Identify a programming skill-set able to be automatically evaluated;
- Investigate strategies to automatically evaluate the identified skills;
- Implement the 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 evaluation capacity.

1.2 CONTRIBUTIONS

The following contributions are highlighted:

- Systematic Literature Review provides an updated and rigorous panorama of automatic programming source code evaluation, being useful for the present thesis and also for future references in this topic;
- Definition of a standardized computer programming skill-set relevant to automatic identification;
- Definition of a set of strategies, as well as their implementation and evaluation, responsible for automatic identifying skill evidences from students' source codes;
- Definition of 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 Evidences Identification method;
- Collecting a source code dataset from real students, from a real programming course;

- Sharing the method and tools to allow new data collections and experiments;
- All software developed for this thesis is distributed with an open-source license, as well as their respective documentation manuals[1] are made available to promote research transparency and its dissemination.

1.3 THESIS STRUCTURE

This thesis is organized as follows: Chapter 2 presents a systematic literature review about automatic assessment of programming exercises, aspects covered, strategies employed, evaluation methodologies and knowledge mapping techniques; Chapter 3 describes the methodology adopted for this research; Chapter 4 presents the experiments conducted to evaluate our method and discusses on their results; Finally, Chapter 5 presents our conclusion and direction for future research.

[1]Documentation manuals are available as appendices at the end of this thesis.

2 SYSTEMATIC LITERATURE REVIEW

A rigorous and comprehensive panorama of the literature is needed to situate researchers regarding the existing contributions in the field, their limitations, challenges and space for future research. This chapter presents the results of a systematic literature review conducted to draw a panorama for the field, showing what computer programming skills can (not) be identified by automatic source code evaluation, and what strategies have been employed to do so. Results provide an overview of the state of the art, identifying 43 different aspects of source code used as evidence of programming skills, and mapping 25 strategies used as automatic identification methods. Results also pointed out to challenges and research opportunities related to different aspects, such as methodologies to evaluate initiatives, resources for knowledge representation, and visualization mechanisms. Remaining of this chapter is organized as follows: Section 2.1 shows related literature reviews, Section 2.2 presents our review research method, Section 2.3 presents our findings, and lastly, Section 2.4 concludes the chapter.

2.1 RELATED WORKS

Research on methods for automatic evaluation of skills in computer programming has presented expressive contributions in literature, and other initiatives already tried to offer an overview and draw a panorama for the field, covering different periods of time and applying different methods. This section describes and discusses papers that present literature reviews on studies related to the automatic analysis of source code to identify evidences of skills in computer programming. Studies are presented in chronological order in the following.

(Ala-Mutka, 2005) contextualizes two main approaches for automatic evaluation of programming source codes: static and dynamic. While static approach consists of evaluating source codes without performing its execution, dynamic approach requires the source code to be executed. Ala-Mutka's review describes a set of aspects automatically identifiable through source code analysis, the strategies employed to do so, and examples of tools. In the dynamic approach, functionality, testing skills and efficiency aspects are mentioned. Among strategies listed to assess dynamic aspects are test cases, reflection and CPU metrics. In the static approach, coding style, programming errors, software metrics, and design aspects are listed. Lastly, among the static approach strategies the author lists compilers, LOC (lines of code) metric, cyclomatic complexity, and structural similarity (tree-based code comparison). In both approaches, the review mentions the presence of specific aspects and strategies applicable only in certain scenarios (e.g., dynamic memory management in C ++ language).

(Rahman and Nordin, 2007) describe a review limited to the static approach, presenting the programming aspects to be evaluated in students' source codes and the respective strategies responsible for the evaluation, such as programming style, indentation, structuring, comments, appropriate variables nomenclature, line spacing, variables scope, and use of constants. The *Style*++ tool (Ala-Mutka et al., 2004) is cited as a strategy for automatic evaluating these aspects, including syntactic and semantic error detection such as forgetting semicolons, unbalanced brackets, never-ending loops, and division by zero. Integrated Development Environments (IDE) and Code Analyzer for Pascal tool are cited as strategies for automatic error detection, and software metrics such as Cyclomatic Complexity and Number of Classes are also cited. The review also covers different methodologies for automatic source code evaluation, such

as structural similarities, non-structural similarity, keyword search, plagiarism detection, and diagrams evaluation.

Static and dynamic approaches were also presented by (Liang et al., 2009), whose review focused on methods for source code automatic correction. The authors describe the static approach in two steps: (1) transform students' code into an intermediate representation, such as a tree or a graph; and (2) analyze the resulting structure by comparing intermediate representations with reference solutions, searching for sub-elements (e.g., sub-trees), counting instructions, etc. Capabilities of this approach are cited, such as automatic identification of correctness to indicate whether a program is correct or not according to specifications, efficiency calculation (worst case metric) and source code quality evaluation (e.g., software metrics, unused variables, Cyclomatic Complexity).

Dynamic approach is described by (Liang et al., 2009) as a process based on compilation, execution, and analysis of students programs. Capabilities of this approach are evidenced by dealing with: i) functional correctness: the program execution behavior is evaluated by verifying if it returns the correct output for a given test set; ii) execution efficiency: evaluation of students' solution quality regarding processor and memory usage metrics; and iii) student testing skills: evaluation of test sets written by students. Challenges are also cited for dynamic approach, such as the need to deal with infinite loops, fatal errors, and malicious code execution. Finally, the authors point out to hybrid approaches, which benefit from both static and dynamic techniques.

In addition to the previous literature studies, (Ihantola et al., 2010) present a systematic review on automatic evaluation tools published between 2006 and 2010, adopting the static and dynamic categorization for them. Tools were further classified into four subgroups: (1) visualization tools, (2) automatic correction tools, (3) programming support tools, and (4) microworlds. The review is motivated by the following research questions: (a) *What are the features of automatic assessment systems reported in the literature after 2005?* and (b) *What future directions are indicated?*. The review methodology is detailed, inclusion/exclusion criteria, and preliminary studies are presented. Tools identified were categorized according to several criteria, such as programming language, teaching environments, functionality, possibility of task resubmission, and distribution license.

(Striewe and Goedicke, 2014) also published a review on techniques and systems for automatic source code evaluation. Study scope was delimited to cover Java, object-oriented, and online systems implementing the static approach. As (Liang et al., 2009), authors cite static approach capabilities, paying attention to details such as the identification of syntactically correct but improperly used commands (e.g., an *if* command never activated), uses of subterfuges (like using Java's default LinkedList in an exercise that requires the implementation of the data structure itself), and plagiarism detection. Data extraction performed by (Striewe and Goedicke, 2014) prioritizes the reviewed system's information, such as (1) name, (2) static evaluation strategy (authors highlight CheckStyle[1] and PMD[2] tools), and (3) use of bytecode analysis (authors cite the FindBugs tool[3]). Selected tools were evaluated based on the following aspects: source code analysis vs. bytecode analysis, intermediate representation (trees vs. graphs), single file or multiple file analysis (tools that can evaluate entire projects with multiple Java classes), and tool integration analysis (e.g., library or command-line integration).

Focusing on information about tools to support the correction of source codes, (Souza et al., 2016) present a systematic review to answer the following research questions: (1) *What assessment tools for programming assignments have been developed?* and (2) *What are the*

[1]CheckStyle Project. http://checkstyle.sourceforge.net.

[2]PMD Project. http://pmd.sourceforge.net

[3]FindBugs Project. http://findbugs.sourceforge.net.

main characteristics of the assessment tools for programming assignments?. Covering ACM, CiteSeerX, Compendex, Google Scholar, IEEE Xplore and ScienceDirect databases, search for papers was limited to the last 10 years prior to review publication. Results were filtered by applying inclusion and exclusion criteria based on paper's title and abstract. Selected papers were categorized into static and dynamic approaches. For the static approach, capability of identifying correctness, documentation quality, code style, originality (check of plagiarism) and cyclomatic complexity metric were mentioned. For the dynamic approach, ability to identify functional correctness, performance measurement, and verification of tests written by students were listed. In addition, papers were analyzed according to their evaluation type (manual, automatic, semiautomatic), approach (instructor, student-centered), specialization (tools specialized in programming competitions, questionnaires, software tests), programming language, and user interface type (graphical, command line).

Recently, (Ullah et al., 2018) presented a review of source code automatic evaluation systems that consider the common approaches (static, dynamic, hybrid) and is motivated by the following questions: (1) *Are existing systems or some of them widely used?*; (2) *Do they help in students learning?*; and (3) *Is it possible to standardize the specifications of these systems?*. Data extraction was focused on system's details such as name, approach, supported programming languages, automatic evaluation methods (e.g., test cases, regular expressions, keyword search), advantages and limitations. The paper presents a taxonomy for automatic evaluators, suggesting possibilities for standardizing specifications for this kind of system. Categorization was focused on static, dynamic and hybrid approaches: Static analysis includes systems using software metrics, programming style evaluation, error detection, keyword detection, plagiarism detection, similarity analysis (structural and non-structural), and diagrams analysis. Dynamic analysis deals with pattern matching and regular expressions. Finally, the hybrid analysis combines static and dynamic strategies.

Although literature presents relevant reviews focused on the automatic evaluation of skills in computer programming, existing reviews present some limitations and open space for extended and updated studies: (Rahman and Nordin, 2007) deal only with the static approach, and their research may be considered old, demanding update; (Liang et al., 2009) and (Ala-Mutka, 2005) studies updating are also required as they dates from a decade ago or more; (Ihantola et al., 2010) presented a tool-focused review, offering no details about methods and association with programming concepts, covering work published in a small period (2006 to 2010); (Souza et al., 2016) also focus on tools only, including automatic and manual evaluation systems; (Ullah et al., 2018), in turn, present a more recent work, showing characteristics and categorizations for the evaluation systems, however, the study focuses on the analysis of effects these systems cause in the students.

2.2 RESEARCH METHOD

This research is exploratory, reviewing and analyzing the state of the art on techniques to identify skills on computer programming based on automatic analysis of source code. A systematic literature review was conducted to select relevant papers and extract the necessary data. Based on the guidelines from (Petersen et al., 2008), the review process covered different stages: from defining the research questions and the inclusion and exclusion criteria, to performing data extraction and analysis. The main stages are described in the sequence.

Research Questions: Our review covers papers reporting research to identify evidences of computer programming skills from automatic source code analysis. Four main research questions are listed:

- *RQ1: What can (not) be automatically evaluated?* Expected results: to identify what aspects (concepts/learning cues) can be automatically identified, and gathering clues about limitations indicating aspects not passable to automatic identification;
- *RQ2: What strategies have been applied to automatically evaluate these aspects?* Expected results: a map of technologies, techniques, and strategies successfully employed. Also, to get information about negative results and experiences from experimented techniques;
- *RQ3: How techniques have been evaluated?* Expected results: to map methods and metrics used for evaluating strategies, to understand how the evaluation was characterized, and which was the audience/test datasets; and
- *RQ4: How is knowledge represented?* Expected results: to identify how measured knowledge is represented (tables, lists, maps, graphs, etc.). If available, check how do they represent student progress over time.

Search Process: for selecting scientific publications, relevant keywords must be defined and databases must be selected. The search string was created to retrieve papers containing the terms *programming*, *student*, *skill*, *evaluation*, and *automatic*, or their related terms, in paper title, abstract or keywords. Five of the most relevant digital libraries were selected: ACM, IEEE, Scopus, Scielo and CEIE. Libraries selection considered the following criteria: (1) parametrized search possibility; (2) availability for access in the academic environment[4]. Table 2.1 presents the search string adapted for each one, allowing search reproducibility and expansion.

Table 2.1: Parametrized search strings.

Database	**String**
ACM	(+programming +(learner student pupil participants) +(knowledge skill ability error) +(learning inference assessment evaluation identification analysis detection) +automatic)
IEEE	(programming AND (learner OR student OR pupil OR participants) AND (knowledge OR skill OR ability OR error) AND (learning OR inference OR assessment OR evaluation OR identification OR analysis OR detection) AND automatic)
Scopus	((programming) AND (learner OR student OR pupil OR participants) AND (knowledge OR skill OR ability OR error) AND (learning OR inference OR assessment OR evaluation OR identification OR analysis OR detection) AND (automatic))
SciELO	(programming) AND (learner OR student OR pupil OR participants) AND (knowledge OR skill OR ability OR error) AND (learning OR inference OR assessment OR evaluation OR identification OR analysis OR detection) AND (automatic)
CEIE	programming (learner OR student OR pupil OR participant*) (knowledge OR skill* OR abilit* OR error) (learning OR inference OR assessment OR evaluation OR identification OR analysis OR detection) automatic*

Inclusion and Exclusion Criteria: Publications retrieved by the search engines were subjected to a filtering process to select relevant papers and discard those not related to the

[4]Libraries subscription access provided by the Federal University of Paraná.

research objectives. This process was conducted by reading papers' title and abstract, and applying the criteria for inclusion (CI) and criteria for exclusion (CE) detailed in Table 2.2:

Table 2.2: Criteria for paper inclusion or exclusion.

Inclusion	Exclusion	Description
CI-01		Deals with automatic assessment of knowledge/Identification of skills.
CI-02		Deals with automatic evaluation of source codes.
CI-03		Deals with error identification in student-written programs.
	CE-01	Does not address the domain of computer programming.
	CE-02	Does not present a proposal for automatic source code evaluation.
	CE-03	The paper is duplicated.
	CE-04	The paper does not deals with automatic knowledge assessment.
	CE-05	Non available for full access.

Data Extraction: A data extraction form was created to support extraction process and its accuracy, as well as to favour results reproducibility (see Table 2.3). The first six attributes (ID, Base, URL, Title and Year) represent papers' metadata, while the following ones refer to the four research questions described previously. Data extraction was performed from the complete reading of the selected publications.

2.3 RESULTS AND DISCUSSION

Results are presented in the order in which the processes was performed. This section presents details about the process and the data obtained from papers search[5], filtering and preliminary analysis. The following sections present data extraction[6] for RQ1 to RQ4 and Section 2.4 concludes the review.

Table 2.4 displays the number of retrieved papers per database. Scopus returned most of the papers, however, the inclusion rate was considered low as only 9.16% were selected for data extraction. Scielo presented the best inclusion rate, however the sample was very small with only one paper. Therefore, IEEE database presented the best inclusion rate (around 23%) among the databases.

Figure 2.1 shows a chart representing the number of papers (vertical axis) selected for our review and their respective publication year (horizontal axis). The oldest selected paper was published in the 60's, however, the subject began to gain popularity from the 2000's, reaching 17 publications in 2016. Continuous lines represent the growing trend rate for published papers, showing that all databases have increased the number of papers over the years.

Inspired by publications timeline, programming languages adopted by each selected study were analyzed according to paper publication year. Data was grouped for decades, starting from the 60's to 2019. The first time interval was not representative as no paper specified the target programming language. The following period, 70's, has papers reporting using Cobol, Fortran and PL/I. The next period was the first to present more than one research on the same language: Lisp. In the 80's papers investigated ALGOL 60, Assembly, C, MODULA-2 and Pascal. The following 10 years presented a greater diversity, having papers investigating solutions for Ada, C++, Prolog, Lisp and Pascal. After the 2000's, several languages began to be explored,

[5]Papers search was performed in March 2019.

[6]Supplementary data to this chapter can be found online at http://bit.ly/doc_sreview_final.

Table 2.3: Extraction form.

RQ	Attribute	Description
	ID	Paper numerical identifier on current review.
	Base	Database where the paper is indexed.
	URL	Paper online address (when available).
	Title	Paper title.
	Year	Publication year.
RQ1	Aspects	Learning concepts/traits that can be identified automatically.
RQ1	Limitations	Aspects that are not identifiable or are identified with limitations. This attribute refers to limitations in aspects only, not covering limitations imposed by methods.
RQ1	Language	Programming language target of automatic identification; refers to the language of the analyzed source code, not the language in which the system was built.
RQ2	Strategies	Techniques and strategies that have been used successfully.
RQ2	Limited Strategies	Techniques that do not work or only partially work (have limitations). This field covers implementation and method limitations.
RQ3	What has been evaluated	What parameters were evaluated in the research/paper.
RQ3	How it was evaluated	What methods and tools were used for the research evaluation.
RQ3	Evaluation target	Who (or what) was the evaluation target audience: students, teachers, source code dataset.
RQ3	N	Size of test suite (number of individuals, projects, programs, etc.).
RQ4	Knowledge map	Describe technique used to represent students' knowledge (if applicable).
RQ4	Temporal progress	Described the how student knowledge progress is represented over time (if applicable).

Table 2.4: Execution of the search strings and filtering results.

Base	Results	Classified	Rate
ACM	233	47	20.17%
IEEE	206	48	23.30%
Scopus	327	30	9.17%
SciELO	1	1	100.0%
CEIE	3	0	0.0%
Total	770	126	16.36%

among them, the emergence and predominance of Java, followed by a growing interest in C and C++. The last analyzed period included the years between 2010 and 2019 when the number of publications focusing on C, C++ and especially Java was accentuated. This decade also marked the expressive debut of research with visual languages and Python. Appendix A.1 shows data used to generate the languages popularity presented in Figure 2.2.

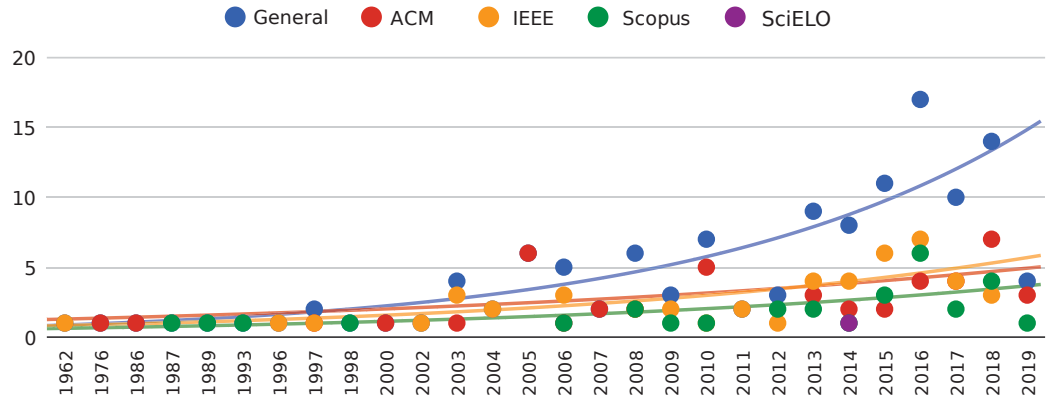


Figure 2.1: Publications timeline chart.

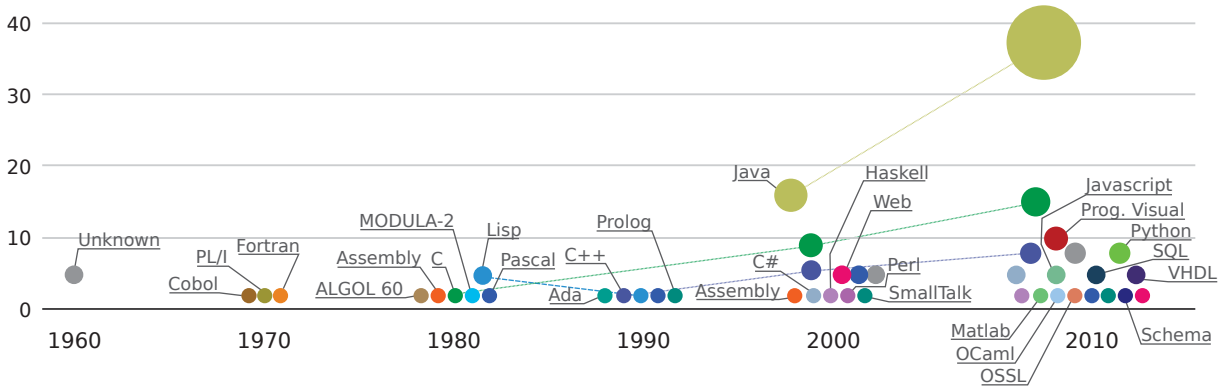


Figure 2.2: Most investigated programming languages chart.

After extracted, data were normalized and organized. The normalization process was conducted in an iterative way similar to the method applied by (Ihantola et al., 2010). Data grouping (categorization) started by evaluating a set of papers and initial categories were then established; later, new paper sets were submitted to evaluation. At the end of each iteration, categories were reviewed, included or merged with existing ones. The next section starts to describe the extracted data.

2.3.1 RQ1: What can (not) be automatically evaluated?

Related studies have already presented categorization for automatically evaluated programming aspects (Rahman and Nordin, 2007; Liang et al., 2009; Souza et al., 2016; Ullah et al., 2018). However, when analyzing papers, aspects not categorized by the aforementioned works were identified. Updating and extending the sets mentioned in the literature would benefit future data analyses, therefore, a new and more detailed categorization compared to those presented by previous works was created. This new categorization, with number of papers (total) related to each aspect and its percentage is shown in Table 2.5.

Dealing with source code automatic assessment, the most commonly evaluated aspect in the analyzed literature is *Functional Correctness*, which refers to executing students' programs, inserting input data and evaluating output correctness. This aspect is evaluated as a boolean parameter indicating whether the program is correct or not. Systems that evaluate Functional Correctness in both complete programs and code fragments were identified (Jackson and Usher, 1997; Morris, 2003). Functional Correctness evaluation is not limited to traditional methodologies

Table 2.5: Aspect category, total 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

(such as writing source code in an IDE, compiling and running it in a terminal), having been applied to web programming (Sztipanovits et al., 2008; Qian et al., 2008) and robotics (Siegfried et al., 2017).

Executing students' code allows to identify aspects other than Functional Correctness. Evaluating code performance concerning hardware resources usage, categorized as *Efficiency*, is approached in papers such as (Vesin et al., 2013; Muñoz De La Peña et al., 2012; Patel et al., 2015). *Test* aspect, in turn, evaluates students' ability to write tests for computer programs, being mentioned in papers such as (de Souza et al., 2014; Edwards, 2003).

Students' code automatic evaluation is addressed by some works. *Complexity* aspect refers to different analysis, such as cyclomatic complexity (Patel et al., 2015; Jackson and Usher, 1997) and spatial complexity (Spandana et al., 2018). *Style* aspect refers to code style evaluation, analyzing parameters such as indentation, comments, documentation, and clarity of the nomenclature used by the student (Rosenthal et al., 2002; Jackson and Usher, 1997; Patel et al., 2015). Regarding this aspect, different papers evaluate students' *Originality*, described here as an aspect related to solutions' creativity and exclusivity, also related to plagiarism detection (Xiaohong Su et al., 2016; Fonte et al., 2014; Rashid et al., 2016).

In addition to research focused on evaluation success, there is research focused on error identification. For the present review, the presence of error is defined as a lack of skill. Examples are publications dealing with the identification of *Lexical Errors* (Wilcox et al., 1976; Suarez and Sison, 2008), *Syntax errors* (Wilcox et al., 1976; Tiantian et al., 2009), *Semantic Errors*[7] (Bahlke and Snelting, 1986). We also found publications dealing with *Compilation Errors* (Jadud and Dorn, 2015; Ahmed et al., 2018) and *Execution Errors* (Wilcox et al., 1976; Delgado and De Barros, 2006). Finally, papers focused on identifying duplicity and inefficient

[7]Literature also references this type of error as *Logical Errors*.

code patterns, referred here as *Antipatterns* aspect, were also included (Ureel II and Wallace, 2019; Cardell-Oliver, 2013).

Abstract concepts, such as *Computational Thinking*, *Problem Solving Strategy* and *Algorithm* are preliminary to the study of computer programming and have automatic evaluation initiatives reported in literature (Kiesmueller et al., 2010; Gerdes et al., 2010; Koh et al., 2010). *Simulation* ability, either mental or tool-supported, is also subject of studies (Brusilovsky and Sosnovsky, 2005; Malmi et al., 2005). Understanding abstract concepts is usually followed by learning programming fundamentals. Automatic evaluation initiatives were found for the following concepts: *Types*, *Input and Output* (Moreno-León et al., 2017), *Variables*, *Conditionals*, *Strings*, *Homogeneous Structures* (single-type data-grouping structures, such as arrays/vectors and matrices) (Rajala et al., 2016), *Loops*, *Procedures* (Ota et al., 2016), *Constants*, *Functions*, *Heterogeneous Structures* (mixed-type data-grouping structures, such as structs), *Scope* (Turner, 2015), and *Recursion* (Hamouda et al., 2018). In addition to programming fundamentals, object orientation is addressed by research that deals with concepts such as *Classes* (Turner, 2015), *Methods*, *Constructors*, *Inheritance*, *Interfaces*, *Polymorphism* (Rajala et al., 2016), *Abstraction* (Moreno-León et al., 2017), and *Enumerators* (Turner, 2015).

Evaluation of specific topics in programming is also addressed. *Concurrency* covers topics related to concurrent programming, such as threads, deadlocks, parallelism and synchronization (Choi and Lewis, 2000; Oechsle and Barzen, 2007). *Code Reuse* detection, *User Interface* evaluation (Etzkorn et al., 1996), and *Events* handling (Ota et al., 2016) are also specific aspects addressed in the literature.

Although most papers (123) present only plausible possibilities and successful experiments, limitations related to aspects not subject to automatic detection were noted (*what can not be automatically evaluated?*). The following limitations were found: (1) (Marin et al., 2017) cite limitations to treat infinite loops; (2) (Rashid et al., 2016) mention constraints for automatic evaluation of graphical interfaces; (3) (Moreno-León et al., 2017) mention the impossibility of ensuring that a correctly used resource effectively means the student actually knows it (automatic evaluation general limitation).

In contrast, the literature shows that several aspects are common in articles dealing with automatic evaluation (*what can be automatically evaluated?*). The categorization presented in Table 2.5 reveals that the most popular aspects are those that deal with the automatic evaluation of the source code as a whole, pointing to generic results such as correctness and the presence of errors in the entire program. In turn, as the less common aspects are listed, the predominance of more specific programming topics is identified, reflecting details about small code units, such as functions, variables, and constants. Therefore, results may suggest the more specific the aspect, the more complex and less popular is its automatic evaluation.

2.3.2 RQ2: What strategies have been applied to automatically evaluate these aspects?

Strategies categorization and identification of the aspects presented previously were performed in an iterative process, expanding, modifying and updating existing clusters from the literature. The new categorization, with total count and percentage, is shown in Table 2.6 and described below. Nomenclature for the strategies is highlighted in italics.

When dealing with automatic source code evaluation, the most common approach is *Test Cases* to validate program functionalities (48 papers). Two methods were commonly applied: (1) (Staubitz et al., 2014) used both students' program execution and a reference program execution with the same input sets, comparing the outputs; (2) (Akahane et al., 2015) executed only students' programs with input sets and previously known outputs, which allowed comparing outputs with reference values. The first method can use random input sets, decreasing chances of fraud by

Table 2.6: Strategy categories with total.

Strategy	Total	Percentage
Test Cases	48	33.80
Tool Application, Unit Tests	22	15.49
Compilation Analysis	16	11.27
Structural Similarity	15	10.56
Execution Traces Analysis	14	9.86
AST, Software Metrics	8	5.63
Specialized Strategy, Memory Metric, Textual Similarity	6	4.23
CPU Metric, Reflection	5	3.52
Regular Expressions, Code Mutation	4	2.82
Classifier Application, Parser, PDG, Questionnaire	3	2.11
Debug Analysis, Flow Analysis, Competitive Evaluation, DSL, Model Tracing, Behavioral Similarity	1	0.70

students (Muñoz De La Peña et al., 2012). In both cases, students' program output may be correct but erroneously evaluated due to small format differences (such as extra blank spaces or line breaks). *Regular Expressions* strategy is suggested to bypass this problem and identify elements that correspond to the positive evaluation (Morris, 2003).

Similar to Test Cases but having more specific objectives, *Unit Tests* are also applied at runtime but on small units of code (such as methods or functions) rather than on the program main output. This practice is generally supported by test libraries, such as JUnit (Amelung et al., 2008) and xUnit (Le Ru et al., 2015). Similarly, *Reflection* technique is used to evaluate internal code elements, such as the presence of variables and data types validation (Morris, 2003).

Evidences of programming skills are also extracted by processes such as *Compilation Analysis* and *Debug analysis*. The first strategy concerns the automatic evaluation of compiler logs (Yamashita et al., 2017), using modified compilers to facilitate data collection (Yaganteeswarudu, 2016). The second strategy, employed by (Murray, 1987), deals with the automation of debugging tools. In addition to these methods, *Execution Traces Analysis* extracts information resulting from the program's execution, such as exceptions, error codes, and terminal outputs (Herout and Brada, 2015; Kim et al., 2016). Similar to Compilation Analysis, *Flow Analysis* is based on the program's execution behavior estimation. (Gerdt and Sajaniemi, 2006), the only paper identified in this category, used compilation techniques to construct a flow graph.

Data from executing Programs can be used to evaluate the code written by students regarding their performance. *CPU Metric* and *Memory Metric* strategies are based on hardware resources analysis, such as CPU time and memory use. Approaches that perform runtime measurements (Patel et al., 2015; Rosenthal et al., 2002) and mathematical estimates (Spandana et al., 2018) were also identified. Other papers also use different *Software Metrics* for source code analysis, without requiring its execution. Methods such cyclomatic complexity, statement count, LOC (lines of code), and NOC (number of classes) are examples of software metrics (Rahman and Nordin, 2007; Hung et al., 1993).

Tool Application refers to initiatives that use software systems for code analysis and comparisons, such as PMD and diff (Helmick, 2007), compilers and IDE (Conejo et al., 2018; Drasutis et al., 2010), software testing tools (Vesin et al., 2013; de Souza et al., 2014) and online environments (Maguire et al., 2017). Similarly, *Classifier Application* refers to initiatives that apply pattern recognition methods in students' code, such as (Kiesmueller et al., 2010) and (Kohn, 2019).

Textual Similarity, also referred as string matching, is a text comparison-based strategy that can be applied to analyze character by character a source code to identify exactly the same data (Truong et al., 2005), combined with search heuristics to identify fragments and keywords within a text (Jelemenska et al., 2016), or through strings approximation by returning a similarity percentage between two strings (Rojas, 2014). *Structural Similarity* strategy works for the same purpose, however, the comparison between two programs does not happen in students' source code, but rather in an intermediate representation constructed from their code. The intermediate representation aims to avoid details such as variables and methods nomenclature, preserving the code structure. Several methodologies were found in this category, for instance, comparison of block sets applied with visual programming languages such as Scratch[8] (Boe et al., 2013; Koh et al., 2014); code transformation metric, which calculates the amount of edits necessary for a given code to become identical to another one (Singh et al., 2013); and assembly comparison (Xiaohong Su et al., 2016).

Specializations of the Structural Similarity strategy were also found: *AST* (Abstract Syntax Three) stands for methodologies that convert the student code into an intermediate representation in the form of a tree, and then traverse it (Insa and Silva, 2018) or performs pattern recognition (Možina et al., 2018). (Insa and Silva, 2018) use *Parser* to create syntactic trees, and there are papers, such as (Wilcox et al., 1976) and (Beh et al., 2016), treating this method as an isolated strategy used to extract information from students' code. Similarly, the *Program Dependence Graph* (PDG) strategy uses graphs as an intermediate representation (Jamil, 2017). Other papers employed *Code Mutation* as an isolated strategy that, unlike code transformation metrics mentioned in structural similarity, acts as a mechanism for producing program variations, generally used to evaluate fragments of code. For example: given a functional program, a correct code fragment is replaced by student's fragment codes (Lee et al., 2018); student-produced test sets where the program is transformed to generate correct and incorrect variations to be submitted to tests: incorrect mutations are expected to fail and correct mutations are expected to succeed (Aaltonen et al., 2010).

Less common and specific strategies were also found in the mapped papers. *Questionnaire* were used in some papers, such as (Kumar, 2005) and (Braunfeld and Fosdick, 1962), to assess student programming skills through multiple choice questions. (Fonte et al., 2013) present *DSL* (Domain Specific Language) as a strategy for auto-grading programming source codes, a programming language specifically designed to define problems, inputs and outputs is used. The breakdown provided by the language allows executing students' programs to gather not only the output but also details such as data structures and types used. *Behavioral Similarity*, mentioned only by (Koh et al., 2010), refers to identification of behavior patterns in visual programming language. *Competitive Evaluation* was presented by (Muñoz De La Peña et al., 2012) and suggests using collective knowledge as an automatic evaluation method by ranking students' solutions. (Ramadhan, 1997), in turn, applies *Model Tracing* as a strategy for analyzing partial solutions and to identify the "path" students followed to get to the answer. Finally, *Specialized Strategy* was attributed to papers whose solution was tailored to specific problems, therefore, not applicable to other contexts and scenarios, e.g., theorem proving based assessment (Quan et al., 2009), and database Structured Query Language (SQL) automatic assessment (Ying and Hong, 2011).

Based on the selected papers, strategies were then classified according to the type of approach common used in the literature: static, dynamic, or hybrid (Figure 2.3). Assumptions for the categorization of static and dynamic strategies are the same in the literature: the primary criterion is the need to execute students' program, which characterizes the dynamic approach. However, this classification differs from previous ones about hybrid strategies. (Ullah et al., 2018)

[8]Scratch visual programming environment, website: https://scratch.mit.edu/

describes the hybrid approach as the combination of static and dynamic techniques, mentioning they complement each other. The taxonomy presented by the authors positions hybrid approach as an intermediate set, but no strategy was explicitly associated with it. We detailed strategies that operate in hybrid mode (i.e., *hybrid strategy*) as strategies that can be applied in both static and dynamic ways, and strategies that combine other distinct strategies and complementary strategies (i.e., *hybrid system*).

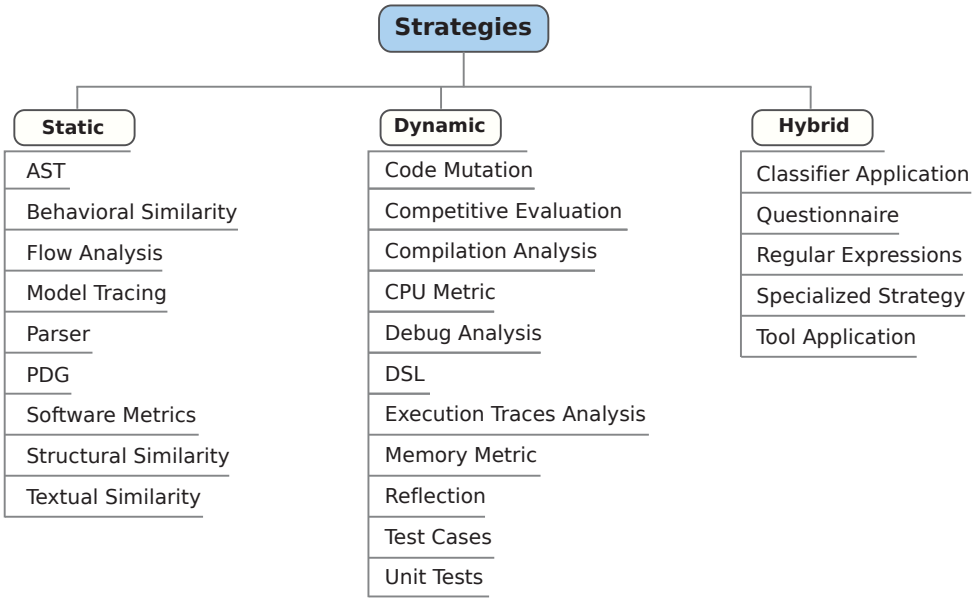


Figure 2.3: Strategies grouped by approach type.

For instance, *Classifier Application* was considered a hybrid strategy as we found papers adopting both static approach, considering code fragments only (Kohn, 2019), and dynamic approach, classifying execution logs (Kiesmueller et al., 2010). *Regular Expressions* strategy was used both for locating information in the source code (static style evaluation presented by (Ureel and Wallace, 2015)) and for extracting data from students' program output (dynamic) (Morris, 2003). *Questionnaires* are also examples of a hybrid strategy: (Kumar, 2005) uses it in a static approach where questions are generated by a tree traversal algorithm, and (Banerjee et al., 2015) describes programming based *Questionnaires* where students' answer is a source code that must be automatically compiled and executed against test cases. *Specialized Strategies* and *Tool Application* were also considered hybrid strategies as they are strictly dependent on the application scenario, being able to be static as well as dynamic.

Besides that, based on the literature reviewed, limitations for strategies were found and classified into two different groups: *method limitations*, when limitations are part of the method and the way it was designed to work; and *implementation limitations*, when the method itself has the potential to be fully functional but, for any reason, it has not been fully implemented. Method limitations are considered more critical as they may impact the viability of a given strategy for a specific problem.

From the 126 selected papers, 21 mentioned limitations for the methods they applied: papers applying specific tools and classifiers reported restrictions in their application (Edmison and Edwards, 2019; Kohn, 2019); (Aaltonen et al., 2010) cite limitations related to Code Mutation strategy as the method only works when students' program succeeds in unit tests. (Traynor and Gibson, 2005) indicate that, in some cases, using Code Mutation for generating programming questions with fully random codes can be confusing to students. For the Test Cases strategy, (Skupas and Dagiene, 2010) mention the method is not able to evaluate partial solutions, and

for the Model Tracing strategy, (Ramadhan, 1997) alleges the method can be too interventionist depending on the way it is applied, negatively impacting students when solving exercises.

Implementation limitations are presented in many different ways, usually characterized by the absence of features for a given system, requirements out of research scope or suggested as future work. From the 126 selected papers, 20 mentioned limitations of implementation, such as: problems related to the conversion of characters in digits (Kim et al., 2016); inability to evaluate exercises using input via files (Truong et al., 2005); not dealing with specific features of C++ programming language, e.g., *union*, *typedef*, *#define* and *templates* (Turner, 2015); and fault-prone implementation, such as dynamic strategies that do not return results when students' programs crash in runtime (Xiaohong Su et al., 2016).

Finally, answering RQ2 (*what strategies have been applied to automatically evaluate these aspects?*), Table 2.6 shows that *Test Cases* (48 papers) is the most commonly used strategy, followed by *Tool Application* and *Unit Tests* (both mentioned in 22 papers). Test Cases strategy is commonly applied mainly due to its simplicity of implementation and to be potentially applicable to any executable program regardless the programming language in which it was built (generic solution). In contrast, less common strategies may be dependent on specific scenarios and impose restrictions. For example, DSL strategy forces students to develop programming assignments in a specific language. It is also noticeable that more generic strategies tend to provide generic and less detailed assessments when compared to specific strategies, such as test cases boolean result vs. a parser that can provide details about code internal instructions. Therefore, we may argue that less common strategies can also provide benefits, such as the evaluation of small units of code, so the choice of strategy is dependent on the evaluation objectives (generic or specific).

2.3.3 RQ3: How techniques have been evaluated?

The third research question aims to characterize the way how authors evaluate their initiatives, proposals and strategies. Data collection for each paper was guided by three sub-questions: (1) *what was evaluated?*; (2) *how was it evaluated?*; and *what was the target audience or object of the evaluation?*. Figure 2.4 presents the categorization defined for these sub-questions, and the remainder of this section describes each one, individually.

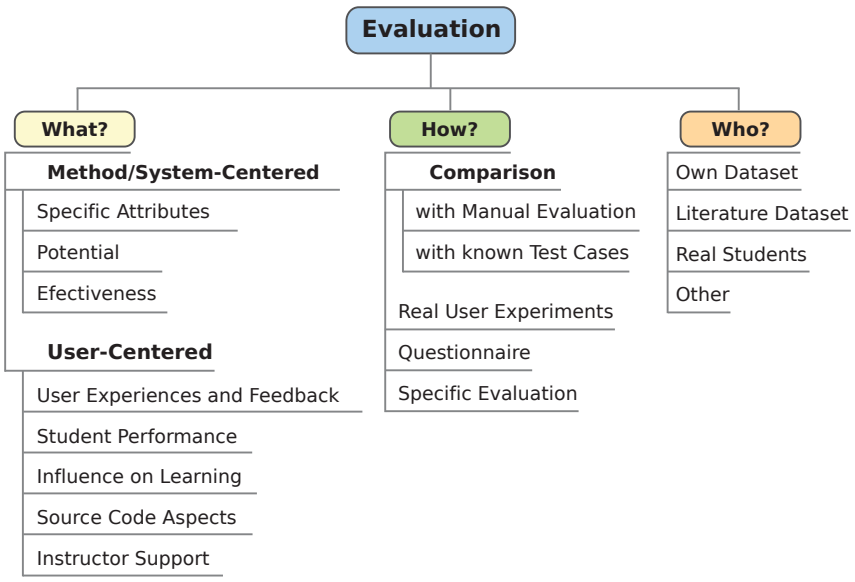


Figure 2.4: Categories for evaluation methodology.

The first sub-question refers to the purpose of the evaluation conducted in each paper. Evaluations centered on implementation (system/method focused) and user-centered (student, instructor, programmer) were the most common. Other papers do not present evaluation but examples and scenarios, listing evaluation as future work, or even evaluating other aspects than the ones related to the identification of programming skills via source code analysis. These papers were categorized as *Not Evaluated*.

Implementation-centered evaluations focus on aspects related to the method or the system only. Some papers investigate *Method or System's Potential*, verifying its applicability and feasibility, such as (Helminen and Malmi, 2010) and (Edmison and Edwards, 2019). Others, in turn, try to validate *Method or System's Effectiveness*, applying tests on part of (or all) the system's functionalities, verifying results produced by the method or the accuracy of the automatic evaluation system (Koh et al., 2014; Aaltonen et al., 2010).

Regarding user-centered evaluations, analysis of system's features and aspects from the user point-of-view were identified in papers such as (Farrow and King, 2008; Siegfried et al., 2017; Traynor and Gibson, 2005; Herout and Brada, 2015). This type of evaluation was characterized as *User Experiences and Feedback*. When dealing with student-focused evaluations, *Student's Performance* comparison is used by different papers as resource for validating the method itself (Hamouda et al., 2018; Yang et al., 2009). Also, collecting student information is used as a source to measure the *Method's Influence on Student Learning* (Cardell-Oliver, 2013; Brusilovsky and Sosnovsky, 2005). Also, analysis of students' *Source Code Aspects*, e.g., quality and structuring, was used by (Förster et al., 2018) as a metric for methodology validation. Finally, (De-La-Fuente-Valentín et al., 2013) and (Head et al., 2017) are examples of papers presenting instructor-centered methodologies, where *Instructor Support* refers to the potential of the proposal to support teachers in their activities and to reduce their workload.

Specific Attributes category was applied to papers that present an evaluation activity but apply problem-specific metrics. (Jadud and Dorn, 2015), for example, use error quotient metric in their method and, therefore, evaluate the method using the same metric. (Skupas and Dagiene, 2010), in turn, analyse incorrect results returned by their method in comparison to manual evaluations. Table 2.7 shows the total for each evaluation category. The total sum is higher than the number of reviewed papers because there are papers applying multiple types of evaluation.

Table 2.7: RQ3 sub-question (1): evaluation focus.

What was Evaluated?	**Total**	**Percentage**
Not Evaluated	46	33.82
Method or System's Effectiveness	35	25.74
User Experiences and Feedback	29	21.32
Student's Performance	13	9.56
Specific Attributes	6	4.41
Method's Influence on Student Learning	6	4.41
Method or System's Potential	4	2.94
Instructor Support	2	1.47
Source Code Aspects	1	0.74

The second RQ3's sub-question aims to know what methods and tools were used for research evaluation. A very popular methodology is *Comparative with Manual Evaluation*, where results from automatic correction mechanisms are compared with results from manual corrections by experts/teachers (Gerdt and Sajaniemi, 2006; Moreno-León et al., 2017). Similarly,

Comparative with Known Test Sets is also a common practice and encompasses comparisons with previous work results (Jadud and Dorn, 2015), datasets (with known result) produced by authors themselves (Marin et al., 2017), and comparative with results of consolidated techniques (Aaltonen et al., 2010).

Research conducting *Users Experiments* were identified. Such approaches collected data from real method/system users (Tiantian et al., 2009), by analysing empirical observations from teachers (Head et al., 2017), students performance evaluation (Braunfeld and Fosdick, 1962), comparisons using experimental and control groups (Hamouda et al., 2018; Yang et al., 2009), and interviews (Kiesmueller et al., 2010). *Questionnaire* is a specialization of this category and refers to the use of feedback collection questionnaires as instruments for method validation (Kamada et al., 2016). Finally, *Specific Evaluation* categorizes specialized techniques found in one paper only. Table 2.8 shows the total for each evaluation methodology.

Table 2.8: RQ3 sub-question (2): evaluation methodology.

How it was Evaluated?	**Total**	**Percentage**
Not Evaluated	46	37.40
Users Experiments	41	33.33
Comparative with Manual Evaluation	18	14.63
Comparative with Known Test Sets	18	14.63
Questionnaires	15	12.20
Specific Evaluation	3	2.44

The last sub-question for RQ3 refers to the target audience (evaluation with people) or the object being evaluated. Papers reported evaluation methodologies applied on *Own Dataset*, where authors produce their test sets (Morris, 2003; Jelemenska et al., 2016). Other papers used a *Literature Dataset*, usually test sets established by a reference work or databases extracted from Intelligent Tutoring Systems (ITS) running by third parties (Singh et al., 2013; Jadud and Dorn, 2015). Papers presenting experiments with students, conducted in real settings such as classrooms and courses, were categorized as *Students*. Finally, the *Other* category was applied to works that mention experiments with non-student users, such as teachers (Pinto, 2013; Head et al., 2017) and programmers (Etzkorn et al., 1996). Table 2.9 presents the number of papers for each evaluation, showing experiments with *Students* as the most common approach among the selected papers.

Table 2.9: RQ3 sub-question (3): target audience.

What was the Target?	**Total**	**Percentage**
Students	52	40.00
Not Evaluated	46	35.38
Own Dataset	18	13.85
Other	9	6.92
Literature Dataset	5	3.85

Details about datasets and participants were also extracted: (1) papers were grouped by target audience; (2) each subgroup was then divided into ranges according to the dataset size and number of participants. Papers classified as *Not Evaluated* as well as unspecified dataset size for evaluations were discarded (this explains differences in data compared to Table 2.9). Figure 2.5 presents the dataset size analysis: 68 papers (labeled as “Reference”) reported details about the

size of their datasets or the number of participants in their experiments. Evaluations with small numbers (less than 10 units) were found in *Students* and *Other* categories. In turn, big datasets and high number of participants were found in papers belonging to *Students* and *Own Dataset*. The most common range was the 100-1000 with 24 papers, while the less common one was the 0-10 range, suggesting tendency in literature to evaluate proposals with larger amount of data and participants.

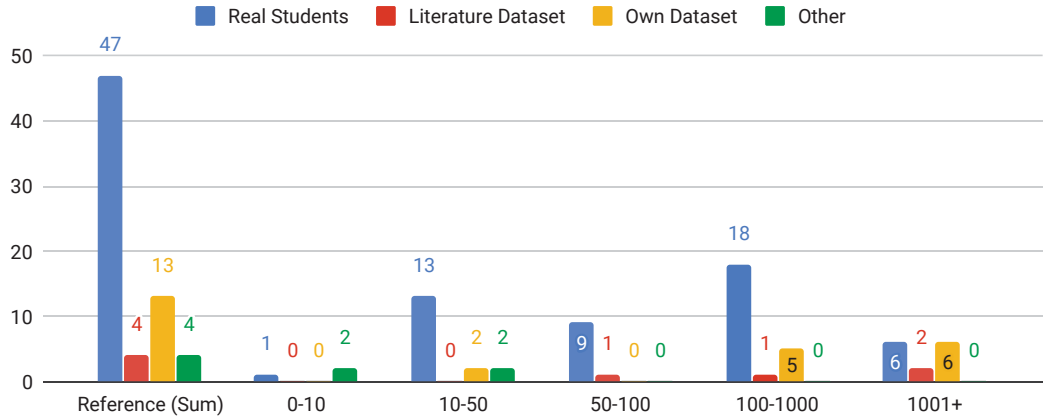


Figure 2.5: Dataset size and number of participants chart.

Considering the results obtained with the three sub-questions, it was possible to draw conclusions regarding RQ3 (*how techniques have been evaluated?*). Results demonstrated that 46 from 126 reviewed papers did not present evaluations related to automatic strategies. In the remaining papers, five evaluation strategies (*How?*) were identified, showing experiments with real users as the most common one (41 papers). Also, focus on effectiveness proved to be the majority, with 35 papers, when considering evaluation purpose. Evaluating with the target audience was also analyzed, and we found out the most common practice is to experiment the proposal with real students. Finally, size of datasets and participant groups were identified, revealing that evaluating with small test sets and participants is rare, and papers tend to evaluate using datasets and experiments with 100 to 1000 source codes or participants.

2.3.4 RQ4: How is knowledge represented?

The fourth research question investigated mechanisms for knowledge external representation and visualization. In this thesis' context, the term external representation refers to techniques for represent, organize and present knowledge[9] (Maschio and Direne, 2015; Cox and Brna, 1995). From 126 papers, only 5 papers presented some kind of student knowledge representation, and only 2 papers mentioned using resources to monitor student evolution over time (temporal progress). This was expected as most selected papers do not propose to explore such resources. The exceptions are listed in the following.

Research published by (Koh et al., 2010; Koh et al., 2014) use *Charts* to represent students' knowledge progress. In both papers, a 9-dimensional model (spider chart) is used, in which each dimension represents a programming concept identified by an automatic evaluation system. Each source code of a solution submitted by a student is evaluated independently, generating one chart per solution. (Koh et al., 2014) represents temporal progress in a chart that

[9]Recently, the term knowledge visualization has also been used in literature.

combines the information of several activities and shows, for each dimension, the maximum score a student reached with a specific source code.

Heat Maps use is described by (Edmison and Edwards, 2019) as a technique to visualize erroneous code fragments identified in students' source code, but temporal progress is not mentioned. Similarly, another paper presents a (non-temporal) representation as a *Knowledge Map* that relates programming concepts and skills with errors identified automatically (Haldeman et al., 2018).

Finally, (Yamashita et al., 2017) use *Timeline* as a mechanism for knowledge representation. The method is based on temporal information, making sequential source code records, compilation attempts, and error occurrences. Data comparison at different time points allows to analyze the evolution of students in programming over time.

Thus, in response to RQ4 (*how is knowledge represented?*), results demonstrated that most of the selected papers did not present details about the representation of knowledge, and indicated that using automatic assessment strategies as a mechanism for monitoring student progress is rare in the literature. It is believed that, within the scope defined for the present systematic literature review, papers dealing specifically with knowledge representation and progress monitoring were excluded due to the restrictions imposed by our search strings. The focus of search strings on terms related to automatic source code evaluation proved to be too restrictive for RQ4 purposes. Thus, it was identified that automatic source code evaluation and knowledge representation/progress monitoring should be treated as separate objectives, ideally addressed in different literature reviews.

2.4 CONCLUSION

This research presented a systematic literature review on aspects and strategies for automatic evaluation of source code as a means for identifying evidences of programming skills. Research protocol was presented, resulting in the selection of 126 scientific papers from ACM, IEEE, Scopus, Scielo and CEIE databases. Results revealed several different programming languages for which automatic identification was investigated, and suggested automatic evaluation has been approached since the 60s, having significant increasing in the number of publications in recent years.

The first research question (RQ1) identified aspects commonly evaluated in automatic source code assessment. A total of 43 aspects that could be automatically evaluated were detailed, and their frequency suggested *Functional Correctness* as the most popular topic in the selected literature, being addressed by 66 papers. Furthermore, generic aspects focused on evaluations of the whole source code are more common, while more specific aspects were less popular.

The second research question (RQ2) aimed to locate the most used strategies for automatic evaluation: 25 categories of strategies were found, and the most used one was *Test Cases* with 48 occurrences. Strategies were classified into static, dynamic, or hybrid approaches. As in the aspects analysis, strategies with generic focus were more common while more specific ones were less explored.

Results provided an overview of the aspects that can be evaluated automatically, the strategies most commonly used to support the assessment, and the limitations mentioned in the literature, revealing space for research and proposals to advance the state of the art and technique. Literature revealed limitations related to aspects and strategies: aspect limitations are rarely reported, while strategy limitations are more commonly found. We suggested the distinction between method and implementation limitations for their categorization, as highlighting the difference helps to understand better the nature of each one.

Subsequently, RQ3 addressed methodologies used by authors to evaluate their initiatives, proposals and strategies. Data revealed that experiments with real users, mostly students, are common practices to evaluate the effectiveness of the proposed methods. In addition, the fourth research question (RQ4) investigated knowledge representation and student progress monitoring mechanisms. Data revealed the great majority of the evaluated works did not present this type of content. The main cause to the absence of this information may be due to the scope defined for this systematic review, which included papers whose focus is related to the automatic source code evaluation only. Therefore, automatic source code evaluation and knowledge representation/progress monitoring may be better treated as separate topics, ideally addressed in distinct reviews.

The literature shows different attempts to use automatic strategies for source code evaluation. The vast majority of studies focus on applying one (or a few) strategies, providing specialized solutions to evaluate certain code aspects. Among the selected papers, we identified gaps regarding using hybrid approaches to mix different strategies and evaluate multiple code aspects simultaneously. We believe the more aspects are evaluated in a source code the richer feedback will be and, consequently, will provide more clues about students' programming skills. Therefore, although specialized solutions are sufficient for certain contexts, developing methods combining multiple strategies to evaluate multiple aspects and, thus, provide detailed unified feedback instead of specialized and isolated solutions is still needed.

3 A-LEARN EVID: AUTOMATIC LEARNING EVIDENCES IDENTIFICATION METHOD

Automatic evaluation of programming source codes is a widespread topic in the literature as shown in the Chapter 2 systematic review. Several methodologies and strategies employed in different scenarios were also identified. The A-Learn EvId method (Automatic Learning Evidences Identification method) is proposed in this thesis as a hybrid approach that employs static and dynamic source code analysis strategies to identify learning evidences, valuate programming skills, and feed a learner model. To conceive the method we take advantage of literature experiences regarding aspects automatically identifiable and strategies employed to do so.

Figure 3.1 presents our method overview. Basic workflow concerns submitting an input source code to evaluation, where several strategies are applied to collect evidences used as basis for feeding our skills-based learner model. Evidences can be collected from single or multiple strategies, and also from inference through combinations of previously evaluated skills.

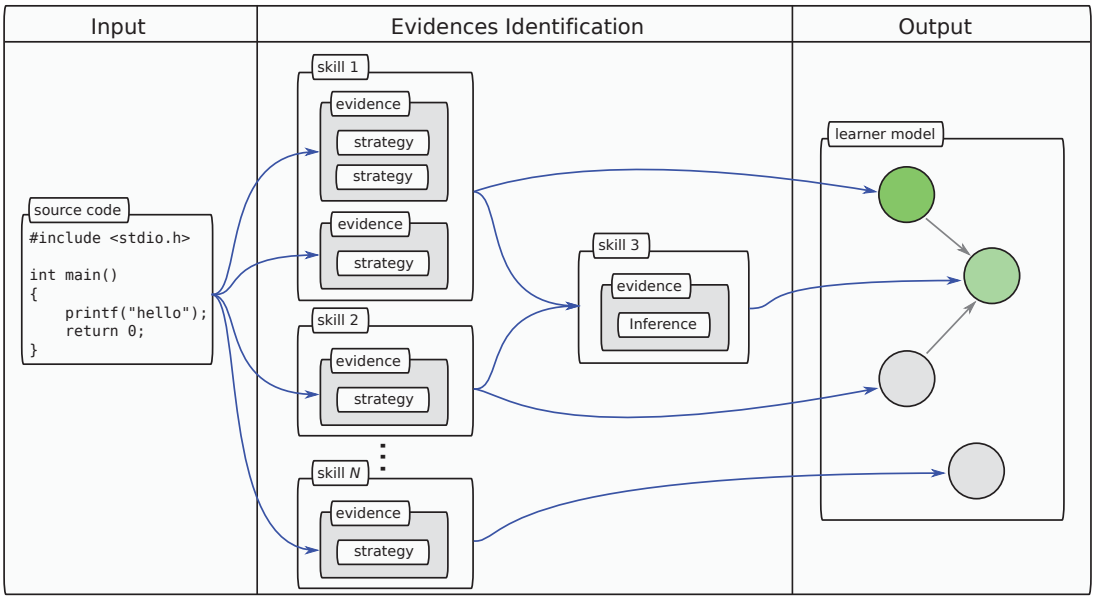


Figure 3.1: Method overview.

In line with Chapter 2 systematic literature review, next sections details our method. Regarding the first research question (*RQ1: What can (not) be automatically evaluated?*), Section 3.1 details the programming skills chosen for automatic evaluation. Then, Section 3.2 presents the strategies employed in learning evidences identification process (*RQ2: What strategies have been applied to automatically evaluate these aspects?*). Student skills representation (*RQ4: How is knowledge represented?*) is shown on Section 3.3, where the learner model is exposed. Finally, Sections 3.4 and 3.5 presents the functional prototype developed and the source code dataset used for evaluating the method (*RQ3: How techniques have been evaluated?*).

3.1 SKILL-SET DEFINITION

Chapter 2 systematic literature review revealed several aspects analyzed in students' developed programs. Most reviewed papers (61.11% of 126 papers) focused on single aspect evaluation, none of them approached more than 10 of the 43 aspects identified. Ideally, the A-Learn EvId

method should use as much information as possible to realistically evaluate 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 (Pimentel and Direne, 1998) and (Maschio, 2013) were used as starting point for our method and are described below.

(Pimentel and Direne, 1998) argument computer programming learning is a difficult task because of two reasons: (1) lack of knowledge of programming principles, and (2) lack of expertise. Therefore, the same authors highlighted the following skills necessary for the student to become a good programmer: (a) syntactic precision; (b) semantic precision; (c) identification of main structures in the source program (keyword search); (d) mental simulation of computer states during execution; (e) error catalog; (f) mental mapping of program structures; (g) precondition check; (h) problem analysis; (i) integration of sub-problems; (j) solution generalization; (k) reuse of known solutions; and (l) solution catalog.

The named skills represent a set of concepts and capabilities required or desired for an expert programmer. These concepts were taken up by (Maschio, 2013), which proposed an extension of this characteristics set. The author highlights the following additional skills: (m) resolution speed; (n) readability of written code; (o) solution optimization; (p) debugging capability; (q) definition of basic test cases; (r) building proper interface dialog; and (s) self-knowledge about metacognitive skills. Together, these 19 capabilities (from *a* to *s*) have been referred as overlying high-level skills.

Also, (Maschio, 2013) presents in detail another subset of computer programming skills, specifically focusing on the imperative paradigm. This subset has 41 skill categories, which cover various concepts needed for initial programmer training. (Maschio, 2013) elaborated a graph, available in Appendix A, which presents this subset of skills in an organized manner. Skills graph highlights programming concepts in an orderly way, with the simplest skills presented earlier and, as the flow progresses through the nodes, complex skills are achieved (often subject to prerequisites).

Considering (Pimentel and Direne, 1998) and (Maschio, 2013) research, to define a set of aspects for studying the evidence of skills automatic evaluation, due to the detailed description provided by the author, the subset of 41 skills elaborated by (Maschio, 2013) was chosen as reference. Thus, Section 3.1.1 presents a correlation between the aspects identified in the systematic literature review and the 41 skills subset of skills described by (Maschio, 2013).

3.1.1 Aspect to Concept Relation

To define a subset of aspects that would be candidate for implementation of automatic skills identification strategies, the first step correlate them with a previously established skills subset. The subset described by (Maschio, 2013) was used as basis to correlate the aspects identified in the Chapter 2 systematic literature review (*RQ1: What can (not) be automatically evaluated?*), thus indicating whether or not there is a similarity between the two programming skills groups. Results are shown in Table 3.1.

Table 3.1: Concept to aspect correlation.

Skills Graph Node (Maschio, 2013)	**Related Aspects (Chapter 2, RQ1)**
analysis	abstraction
structuring and composition	problem solving strategy
effectuation	simulation, tests

Skills Graph Node (Maschio, 2013)	**Related Aspects (Chapter 2, RQ1)**
simple instructions	problem solving strategy, variables, input and output
output	input and output
input	input and output
input vs output	input and output
attribution	
value changes	
types compatibility	types
loss of value	
types of literals	types
variables	variables
constants	constants
variables vs constants	variables, constants
arithmetic expressions	
relational expressions	
division by zero	execution errors
boolean expressions	
compound expressions	
control structures	conditionals, loops
expressions in control structures	
conditional structures	conditionals
multiple selection conditional	conditionals
simple and compound conditionals	conditionals
simple and compound vs multiple selection	conditionals
nesting	scope, conditionals
pipelining	conditionals
nesting vs pipelining	scope, conditionals
repetition structures	loops
counters and accumulators	loops, variables
infinite loops	loops, execution errors
counted loops	loops
conditional loops	loops
pre-evaluated	loops
post evaluated	loops
conditional loops vs counted	loops
pipelining (2)	loops
nesting (2)	scope, loops
nesting vs pipelining (2)	scope, loops

Correlation showed 80.49% compatibility between the skills described by (Maschio, 2013) and the aspects identified in the analysis of RQ1 (33 aspects from RQ1 contain corresponding skills in Maschio's set). Some skills, such as *counted loops* and *conditional loops*, were associated with a same generic aspect (e.g., *loops*), forming groups of similar terms.

Although the high correlation rate indicates Maschio's skill set is interesting for our automatic evidences identification method, a complementary study was designed to support the

subset selection decision. Focusing on the Brazilian scenario, a syllabus study was conducted and is presented in Section 3.1.2 aiming to verify which programming concepts are actually covered by Computer Science courses.

3.1.2 Syllabus Analysis

Considering the many aspects evaluated in computer programming tasks, defining which are candidates for automatic evaluation 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. Select two federal universities from each of the 5 Brazilian regions: 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. The results are shown in Table 3.2.

Table 3.2: Selected universities.

Region	**Abbreviation**	**University Name**
Central-West	UFG	University of Goiás
	UnB	University of Brasília
Northeast	UFCG	University of Campina Grande
	UFPE	University of Pernambuco
North	UFAM	University of Amazonas
	UFPA	University of Pará
Southeast	UFMG	University of Minas Gerais
	UFRJ	University of Rio de Janeiro
South	UFRGS	University of Rio Grande do Sul
	UFSC	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 there is no standardization 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 by synonyms.

“Introduction to Programming” is cited as a syllabus topic. We consider this topic as a main objective that is achieved by mastering the other nine. Therefore, we consider students must develop abilities related to these different topics and that the more developed their skills

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

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.

Table 3.3 shows a ranking of 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 subset) are highlighted. It is possible to note 7 of the 10 syllabus common topics has 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.

Table 3.3: 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	*

Returning to skills selection started in Section 3.1.1, we sought to determine whether the correlated topics are more relevant to be automatically identified. Thus, Maschio's skills subset in conjunction to the popular unrelated topics (*arrays*, *functions* and *matrices*) were taken as the *full skill set* (Figure 3.2)[4].

The full skill set is assumed as reference for the development of our automatic skills evaluation strategies. The ten topics shown on Table 3.3 were considered priority due to its

[2]Full table data is available on Appendix B. Supplementary research files are available online at http://bit.ly/doc_syllabus.

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

[4] (Maschio, 2013) specifies two skills for both *nesting* and *pipelining* topics. The nomenclature of the original work has been preserved; however, the distinction between the skills is highlighted: *nesting* and *pipelining* refers to conditionals, while *nesting (2)* and *pipelining (2)* relate to repetition loops.

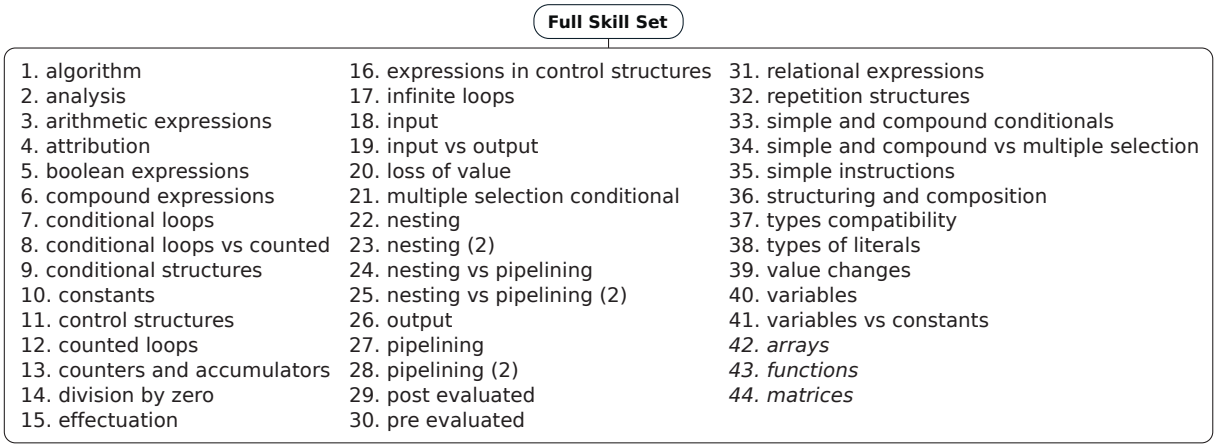


Figure 3.2: Full skill set.

popularity. Also, a feasibility analysis was performed in the full skill set to detect which skills can be automatically identified from the students' source codes. Classifying skills is important because some characteristics are strongly linked to the student's logical reasoning and perception, being difficult (or even impractical) to evaluate through automatic strategies. Two different skills from the full skill set are taken as examples:

1. *Multiple selection conditional* deals with multiple choice control structures (e.g., *switch-case* in C Language) and requires using a control variable whose data type needs to be countable; and

2. Conceptual *analysis* and abstraction of information based on problem description.

Regarding strategy implementation, the aforementioned skills differ in a very important aspect: how to evaluate students' skills. The first skill (*multiple selection conditional*) concerns using multiple choice conditional structures, more specifically dealing with the variable to be used as a comparison parameter for each case of the structure. In many languages, such as C and Java, there are limitations on the data types accepted by this conditional structure. Therefore, students must be aware of these details and avoid using incompatible types. Evaluating *multiple selection conditional* skill can be performed by verifying the data type of the variable used as comparison parameter. Thus, it is assumed that a student who always uses the correct data types has mastery over the mentioned skill.

The second skill (*analysis*) regards students' abilities required to interpret information provided in exercise description, e.g., by reading exercise statement and identifying the problem, possible processing steps, inputs, and outputs. Evaluating *analysis* skill depends on factors prior to the production of the source code, strictly dependent on the student's reasoning and experience. Because it is prior to program writing, abstract aspects such as *analysis* are not always expressed in the student's final code, making automatic identification challenging with strategies based purely on source code analysis. Although strategies focusing on abstract aspects exist and have been identified in the Chapter 2 systematic literature review, approaches such as employed by (Förster et al., 2018) and (Moreno-León et al., 2017) are uncommon and act in limited scenarios (e.g., only working with visual programming languages), without generalization possibilities mentioned by the authors (e.g., application to other programming languages).

Based on cited examples, evaluating different types of skills requires different approaches, some of which can be automatically performed while others are more challenging and may not

be feasible without human intervention. These arguments justify performing skills analysis to identify which of them can be automatically evaluated based on students' source codes.

Skills from the full skill set were categorized regarding potential for automatic identification and priority (Figure 3.3). *Challenging/not feasible* category represents skills that we have not been identified strategies capable of automating evaluation. The remaining skills were categorized as *automatically identifiable*, of which two subcategories were defined: *specific strategy*, which represents aspects identifiable by one (or more) strategies found in the literature; and potentially solved by *inference*, containing aspects challenging to be identified by automatic strategies but which assessment can be performed based on other skills previously assessed[5]. Finally, specific strategy aspects were categorized regarding investigation priority, aspects related to the 10 common syllabus topics were considered *priority* due to its popularity, the remaining aspects were considered *complementary*.

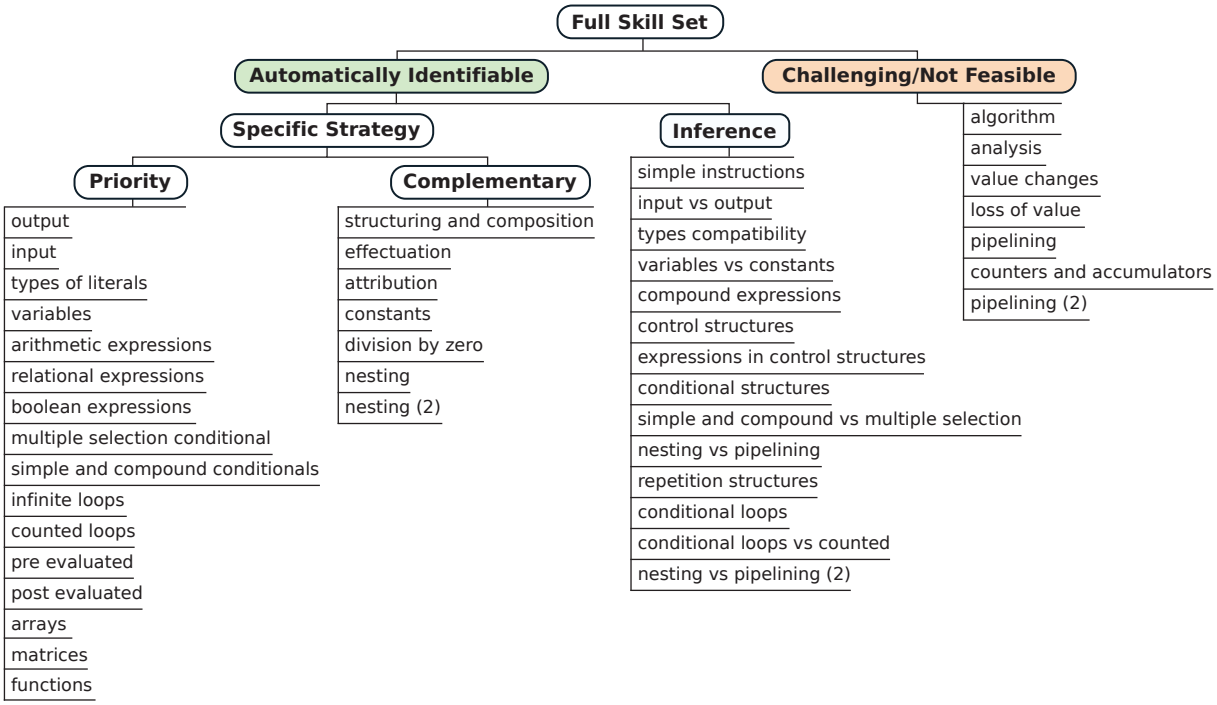


Figure 3.3: Full skill set categorization regarding potential for automatic identification and priority.

Thus, 37 skills of the 44 full skill set were selected for automatic assessment/strategy application. Skills excluded from the process are justified by the facts: (1) *algorithm* and *analysis* were considered abstract skills, strongly dependent on the interpretation of student thinking; (2) *value changes*, *loss of value*, *pipelining* and *pipelining (2)* were considered problem-dependent skills, hard to evaluate without knowing the program execution context and objectives, e.g., according to (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 if the loss of values was a student mistake or an intentional act, for example by reusing the same variable for another purpose; (3) *counters and accumulators* were also considered problem-dependent skills, however, (Gerdt and Sajaniemi, 2006) suggests 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 are listed as future work.

[5]Considering probabilistic inferences based on (Maschio, 2013) graph specification: prerequisites and generalizations. See Appendix A.

3.2 STRATEGIES IMPLEMENTATION

As mentioned in Section 3.1, detecting evidences for different skills requires different strategies, some with easier automatic detection and others impractical. Considering the *automatically identifiable* skills subset previously defined, this section describes the strategies employed on our automatic learning evidences identification method.

From the 25 strategies identified in the systematic literature review, 9 of them were chosen to compose the method and are highlighted in Figure 3.4. Hybrid systems between two or more of the chosen strategies were also applied. Strategies choice considered the skills subset to be evaluated, the analysis desired (static source code inspection and execution details evaluation), the availability of documentation regarding strategy implementation and, finally, adaption possibilities (specific strategies are not always adaptable, e.g., *DSL* can't be applied to source codes written in different programming languages; *Competitive Evaluation* can't be applied to single/isolated students). The remaining of this section explains and exemplifies each strategy employed in our implementation.

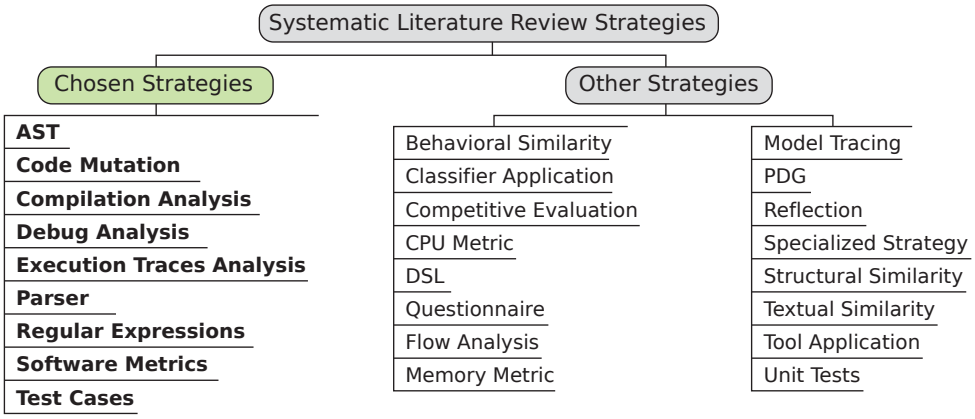


Figure 3.4: Chosen strategies.

3.2.1 AST and Parser

In the area of Computer Science, a parser works receiving input data, processing, and returning subsets of information, organizing them according to some criteria. Specifically, when it comes to computer programming, a parser takes a source code in a programming language, interprets it, and returns elements called *tokens*, which contains details of each instruction that makes up the computer program code.

Abstract Syntax Tree (AST) is a data structure commonly used to represent parser output. 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 finally (4) generation of the tree.

AST and parser strategies are demonstrated with Listing 3.1 source code. The code is written in C language, where it is possible to notice two functions: *test* and *main*. Each of these functions 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 Figure 3.5. Each function of the source code is stored in a tree node, whose children are instructions and blocks that compose it. Furthermore, parser ability to detect attributes of each element, such as names, arguments and literal types of each variable/function is highlighted.

Listing 3.1: Source code example.

```
1 void test(){
2   printf("test!");
3 }
4 int main(){
5   int a = 2;
6   test();
7 }
```

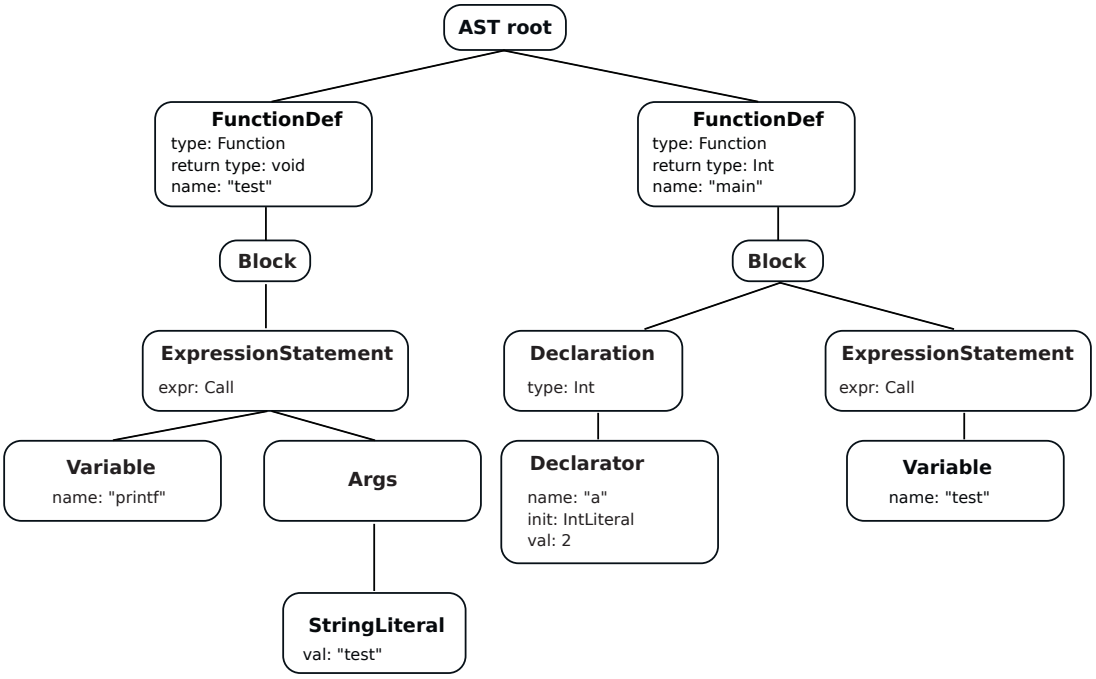


Figure 3.5: Parser execution resulting AST.

Thus, it is possible to perceive the parser strategy as a powerful tool able to identify internal code construction details. Considering parser's resulting tree, it is possible to implement heuristics to search for evidences 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.

The following 21 skills used AST and parser strategies: *output, input, attribution, types of literals, variables, constants, arithmetic expressions, relational expressions, division by zero, boolean expressions, multiple selection conditional, simple and compound conditionals, nesting, infinite loops, counted loops, pre-evaluated, post evaluated, nesting (2), arrays, matrices,* and *functions*.

In the listed skills, parser use was limited exclusively to generating syntactic trees, which were taken as base objects for finding evidences of certain programming resources usage (e.g., conditionals, loops, variables, among others). Parser-based strategy assumes if students use correctly a particular programming resource then there is evidence of learning related to a specific skill, e.g., correctly applying an *if* statement is evidence of learning *conditionals*.

AST and parser strategies are limited to analysing source code structural elements and cannot evaluate programs' execution. To evaluate programs' runtime behavior and outputs, strategies from the dynamic approach are required. *Test cases*, the most common technique in the literature reviewed (Chapter 2), provides resources for such assessment.

3.2.2 Test Cases

Test cases, although being a fairly simple strategy, are widely used and provide good results when evaluating programs' execution output. Evaluating a program with test cases requires the prior specification of input and expected output value sets. Strategy's application is performed by executing programs with the predefined input values, followed by the comparison of the outputs produced.

Considering a previously compiled executable program, operating systems allow redirecting[6] input data to 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. Listing 3.2 shows an example C-Language program that requires the user to enter two numbers and print the double of the first and triple of the second. Figure 3.6 exemplifies the execution of a test case based evaluation on the referred program.

Listing 3.2: Source code example.

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int number, number2;
6      scanf(" %d", &number);
7      printf("double:%d\n", number*2);
8      scanf(" %d", &number2);
9      printf("triple:%d\n", number2*3);
10     return 0;
11 }
```

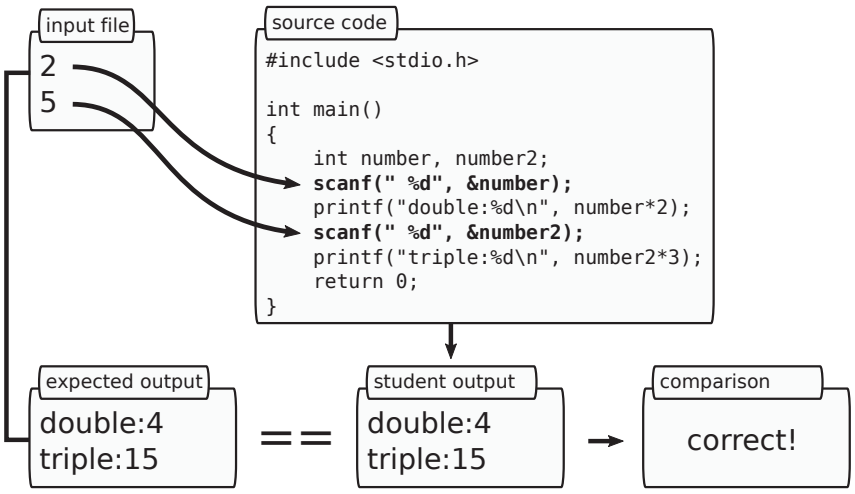


Figure 3.6: Test case application example.

Test cases strategy was applied to all skills evaluated in our method. Preliminary assessment of functional correctness ensures any evidence found by other strategies (such as

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

Listing 3.3: Source code with output precision error.

```
#include <stdio.h>

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

Listing 3.4: GCC output log example.

```
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);
         ^
```

parser) refers to a functionally correct source code. Thus, gathering evidences from different strategies contributes to the accuracy of our method.

Also, a special case is mentioned for the *effectuation* skill, which according to (Maschio, 2013) concerns to algorithm mental simulation, through testing, feedback, correction, and improvement. Feedback and correction processes are strongly benefited from the test cases approach, as students can perform self-assessments using predefined input/output sets, identify potential code flaws, and correct them before submitting the final version to the ITS.

Functional correctness, as identified in the Chapter 2 systematic literature review, is an important aspect to be evaluated in students' source codes, however, its capacity do not go beyond comparing program's final result. Checking programs' internal execution behaviour, such as valuation of variables during the execution process, is unfeasible with test cases. Such analysis requires a different strategy with deep runtime inspections, compilation and debugging processes analysis are cited as alternatives and are described in the next section.

3.2.3 Compilation, Debug and Regular Expressions

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[7] focused in transforming source codes into executable objects.

Compilers and debuggers 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 Listing 3.3, where a failure while printing the value of the variable *a* is present. 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. Listing 3.4 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.

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

Listing 3.5: Sample source code that generates execution error due to division by zero.

```
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 }
```

Listing 3.6: GDB output log example.

```
1 Program received signal SIGFPE, Arithmetic exception.
2 0x00000000004004ec in main () at source.c:7
3 7       int c = a/b;
```

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 evidences through programs' runtime analysis. The C-Language code shown in Listing 3.5 exemplifies using 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. Listing 3.6 shows a snippet of the debugging log, an "Arithmetic exception" error is pointed out in the operation performed in line 7.

The *Compilation analysis* strategy was employed to identify evidences for three skills: *structuring and composition*; *division by zero* and *infinite loops*. According to (Maschio, 2013), algorithm *structuring and composition* concerns, among other tasks, transcribing the algorithm in natural or formal language. In this case, a successful compilation is an evidence that students correctly transcribed their thinking to the algorithmic form respecting the programming language structure. Also, strategies developed for *division by zero* and *infinite loops* skills used runtime dynamic techniques, which required successful compilation as prerequisite.

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 amid text. *Regular expressions* strategy was employed to support data localization both in static mode (i.e., looking for data in the source code itself) and dynamic mode (i.e., acting on execution logs). The following skills used regular expressions in their strategies implementation: *output, input, arithmetic expressions, relational expressions, division by zero, boolean expressions, infinite loops, counted loops, pre-evaluated, post evaluated, arrays, matrices, functions*.

For demonstrating a particular case, the following section presents a set of strategies implemented to locate evidences of infinite loops.

3.2.4 Specialized Strategies: Dealing With a Particular Case

Infinite loops problem was chosen as a particular case to demonstrate the potential of our automatic learning evidences identification method. Infinite loops occurrence is a situation related to the Halting Problem, regarding the completion of a program in finite time given an arbitrary input. The Halting Problem was introduced by (Turing, 1936) just as it was proved to be unsolvable, however, although without a general solution, the search for evidence of halting condition is plausible in specific scenarios. A scenario is shown concentrating in two runtime aspects:

- *Execution timeout*: programs developed by students, such as classroom activities in introductory programming courses, often have short execution. Thus, very long unfinished executions (e.g., excessively above reference solutions execution time) may be an indication of infinite loop;
- *Loop iteration count*: monitoring programs' execution by counting iterations performed in loop statements allows identifying evidences of halted executions. A repetition structure that iterates excessively above expected can be considered an indication of 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 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 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[8]. 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)[9].

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, an example is shown in Figure 3.7. 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 execution halt and being mistakenly categorized as infinite loop by automatic mechanisms such as the timeout strategy previously described.

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

[8]Exercises solutions relating to the 10 default lists mentioned in Section 3.5.3.

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

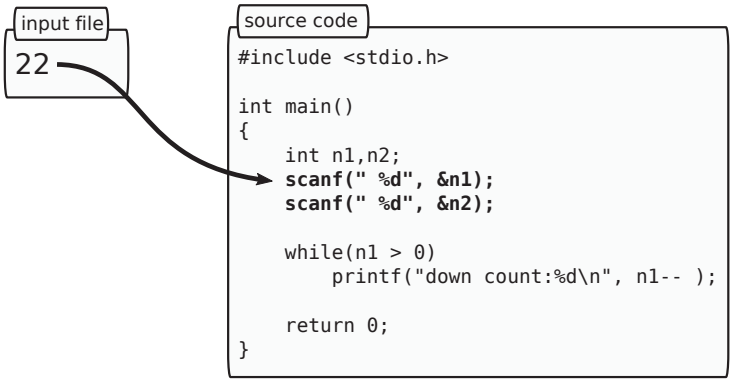


Figure 3.7: False-positive on infinite loop timeout strategy.

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 were killed by timeout and at least one loop iterated more than a threshold value.

To exemplify our strategy, Figure 3.8 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 injection of two global control variables (*l_control_0* and *l_control_1*[10]). An increment instruction is added inside each loop, in this example associated with a standard output printing instruction (*printf*). These additions allow inspecting program's runtime behavior both by collecting standard output or through a debugging process. Thus, a *potential infinite loop* metric can be applied.

[10]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 3.8: Code mutation example.

To apply our strategy and inspect control variables valuation, mutated program must be executed. Two conditions can be reached: program's execution finish in 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 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 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) evaluated 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*. The next section reinforces cases where hybrid systems were employed.

3.2.5 Hybrid Systems

Strategies 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 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.

Returning to the arithmetic exception identification example shown in Section 3.2.3, absence of division by zero errors cannot indicate success if students does not perform any division operation in their source codes. *Debug analysis* strategy must consider a pre-evaluating step where 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 on our method. Skills and associated strategies are shown in Table 3.4. 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. In addition, as a proof of concept, the largest association is observed for the *infinite loops* skill, where multiple techniques were implemented to act together on evidence identification even for complex problems.

Table 3.4: Hybrid systems.

Skill	**Strategies**
structuring and composition	test cases, compilation analysis
output	test cases, regular expressions, AST, parser
input	test cases, regular expressions, AST, parser
attribution	test cases, AST, parser
types of literals	test cases, AST, parser
variables	test cases, AST, parser
constants	test cases, AST, parser
arithmetic expressions	test cases, regular expressions, AST, parser
relational expressions	test cases, regular expressions, AST, parser
division by zero	test cases, regular expressions, AST, parser, compilation analysis, debug analysis
boolean expressions	test cases, regular expressions, AST, parser
multiple selection conditional	test cases, AST, parser
simple and compound conditionals	test cases, AST, parser
nesting	test cases, AST, parser
infinite loops	test cases, regular expressions, compilation analysis, execution traces analysis, software metrics, code mutation, parser, AST, debug analysis
counted loops	test cases, regular expressions, AST, parser
pre-evaluated	test cases, regular expressions, AST, parser
post evaluated	test cases, regular expressions, AST, parser
nesting (2)	test cases, AST, parser
arrays	test cases, regular expressions, AST, parser
matrices	test cases, regular expressions, AST, parser
functions	test cases, regular expressions, AST, parser

Previous sections presented a collection of automatic evidence location strategies applied to the students' source code evaluation problem. Collecting information from students' codes is an important step in their assessment, however, for teaching purposes evaluation results need to

be clearly presented. The following section describes the learner model adopted for representing student skills, where the automatically localized evidences can be applied and presented in an organized manner.

3.3 LEARNER MODEL

In the context of this thesis, the learner model is established to organize and present characteristics representing skills, developed or not by students, necessary for 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 computers programming domain depends on elaborating a set of capacities to be used as a metric. For the present thesis, the skill set defined on Section 3.1 is employed. Characteristics of the knowledge representation mechanisms identified through our systematic literature review (Section 2.3.4), 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* (available on Appendix A). Maschio's model presents his 41 skill set with details such as prerequisites, dependencies, analogies, and generalizations. Knowledge representation is done by the overlay technique, where 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 this thesis, an analysis of Maschio's representation guided by features found on literature mechanisms was conducted and some improvements were pointed out below.

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. Heat Maps technique presented by (Edmison and Edwards, 2019) focuses on highlighting source code fragments, this feature inspired us implementing a complementary detailed log in which every identified evidence can point to a code snippet responsible for the skill assessment. Figure 3.9 shows an example log returned by our strategies to evaluate *functions*. It is possible to see student's source code and a detailed log pointing 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).

Skill valuation on multiple source codes: Considering students submit several source codes to evaluation, identifying the same skill in multiple source codes is common, specially when the submitted source code set refers a specific programming topic. On such a situation, the automatic evaluation system needs to decide which assessment to take into account (especially because evaluations 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: Considering the model can be fed with information from multiple source codes, a selection mechanism is necessary. A method similar to 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 affects the valuation of the skills represented in the model.

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 3.9: Per source code detailed skill visualization.

Uncertainty treatment: Evaluated 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 evaluation environment, e.g., importing incompatible libraries, using unsupported/legacy syntax, submitting source code with incompatible encoding, and using operating system dependent resources. These facts 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 need to consider temporal aspect. Students continuously submit new source codes along course, evidences identified from different time intervals must be represented accordingly. The student model representation extends then to the concept of Dynamic Bayesian Networks[11] 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 network's capabilities to deal with uncertainty environment and to deal with temporal changes.

Thus, the learner model adopted in the present thesis is described below, supported by the representation shown in Figure 3.10:

- Bayesian Network variables, also called "features" by (Neapolitan, 2003), models the full skill set defined in Section 3.1.2. 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) collected from automatic strategies described on Section 3.2 or probabilistically calculated by inference (described below).

[11] Also known as Temporal Bayesian Networks.

Any evidence of a variable is considered equally influential in the skill value calculation (e.g., on figure, “switch” and “data type compat.” have equal weights). The evidence sets implemented to our method are available on Appendix C.

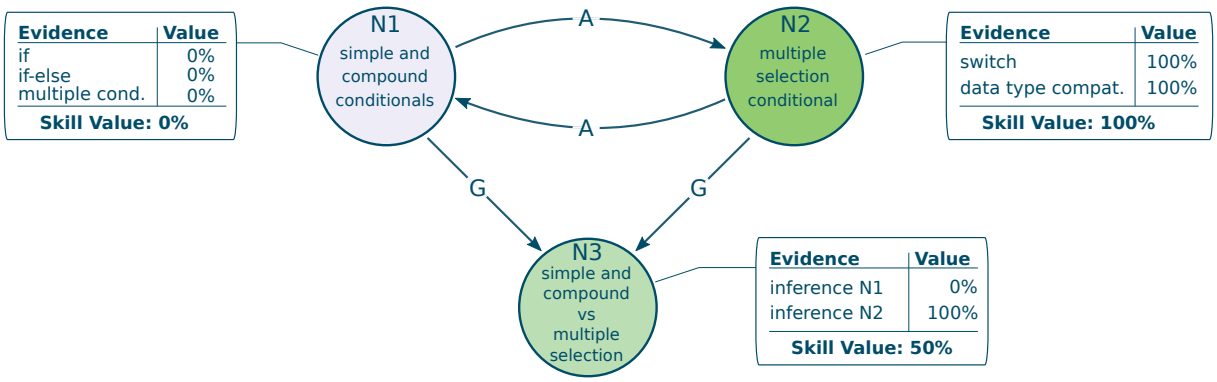


Figure 3.10: Bayesian Network inference.

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 on Figure 3.10, 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 the Figure 3.10, 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 evidences 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 evidences are identified by external automatic strategies. As stated in section 3.1.2, a subset of skills were categorized as “potentially solved by inference” due to its connection and dependence on other concepts, this subset uses the inference strategy aforementioned.

To conclude our learner model description, Figure 3.11 presents examples of skills and their respective valuation sources (when automatically identifiable, according to Section 3.1.2 classification). Skills are valuated by one or more evidences; evidences can be implemented with one or more automatic strategies; and, finally, each strategy analyses source codes and returns an evaluation regarding certain programming aspects (e.g., a percentage of success regarding the use of determined programming resources). Check Appendix C for full skill-set valuation details.

3.4 PRACTICAL APPLICATION: EVIDENCE MACHINE

Our method completeness is achieved by combining the automatic strategies exposed on Section 3.2 with the learner model detailed in Section 3.3, making it possible to perform analyzes on source codes submitted by students, identify learning evidence and apply them as input data for feeding the learner model. *Evidence Machine* is a standalone web application implemented for this purpose, whose characteristics will be detailed in this section distinguishing two perspectives: (1) management; and (2) visualization.

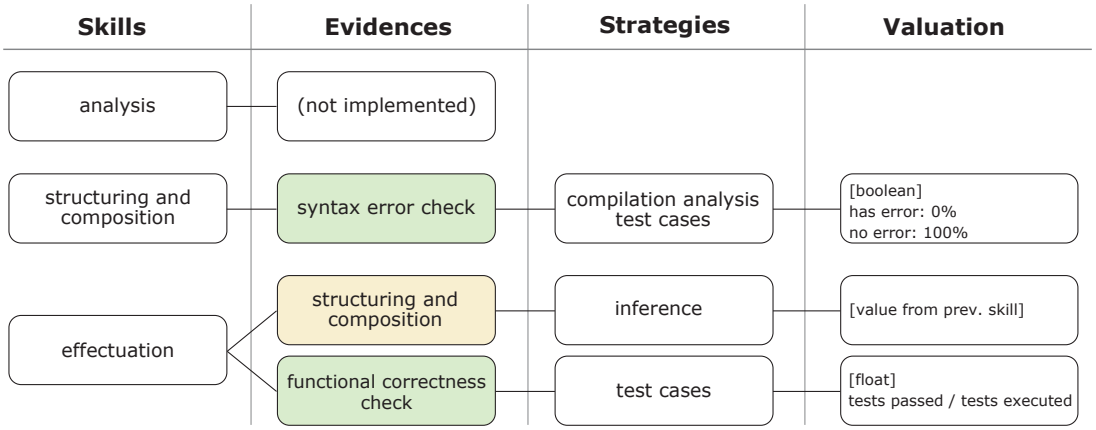


Figure 3.11: Skills valuation sample (Appendix C fragment.)

3.4.1 Management Perspective

Evidence Machine's management perspective aims to provide administration features to execute strategies on students' codes, collect and apply results as evidence for the Bayesian Network, among other functions.

Each learner model skill is associated with one or more strategies, represented by algorithms implemented in the form of scripts organized as a sub-module of the main tool. Implementation through scripts favors the extension of the strategies set since they act in isolation, not affecting the internal behavior of the Evidence Machine. Evidence identification is then performed by these algorithms which receive input source codes and returns percentages of success for a given strategy set. Algorithm result calculation is strictly dependent on strategy's implementation logic.

Considering multiple students, evaluating several source codes can generate heavy workload to the application server (e.g., evaluation of all evidence, in all exercises, of all students in a large class). Thus, Evidence Machine performs algorithms execution asynchronously in the background, massive processing must be scheduled and executed without teacher intervention after the generation of the algorithm execution queue. Two scheduling mechanisms were implemented: (1) full dataset schedule, all source codes submitted are added to execution queue; and (2) selective schedule, where the teacher can navigate through students, select specific algorithms and source codes, and add them to execution queue.

Also, isolated algorithm execution is available from the tool's administrative panel, enabling teachers to execute algorithms in small programs, such as, demonstrating classroom programming capabilities, applying test cases to isolated codes, and evaluating individual source codes. Lastly, user management (student enrollment) and algorithm execution logs viewing features are also available in the management perspective.

3.4.2 Visualization Perspective

Automatic feeding of the learner model by the result of the automatic strategies is one of the main contributions of our method. As discussed in Section 3.3, the learner model is represented by a Dynamic Bayesian Network whose valuation is affected by the student-written source codes timeline. Figure 3.12 presents the learner model implementation, showing: (1) the network activation (nodes valuation, considering the entire network); (2) the source codes timeline (real-time affects the network); and (3) a Bayesian Network fragment showing ten nodes in which green-color indicates higher valuation.

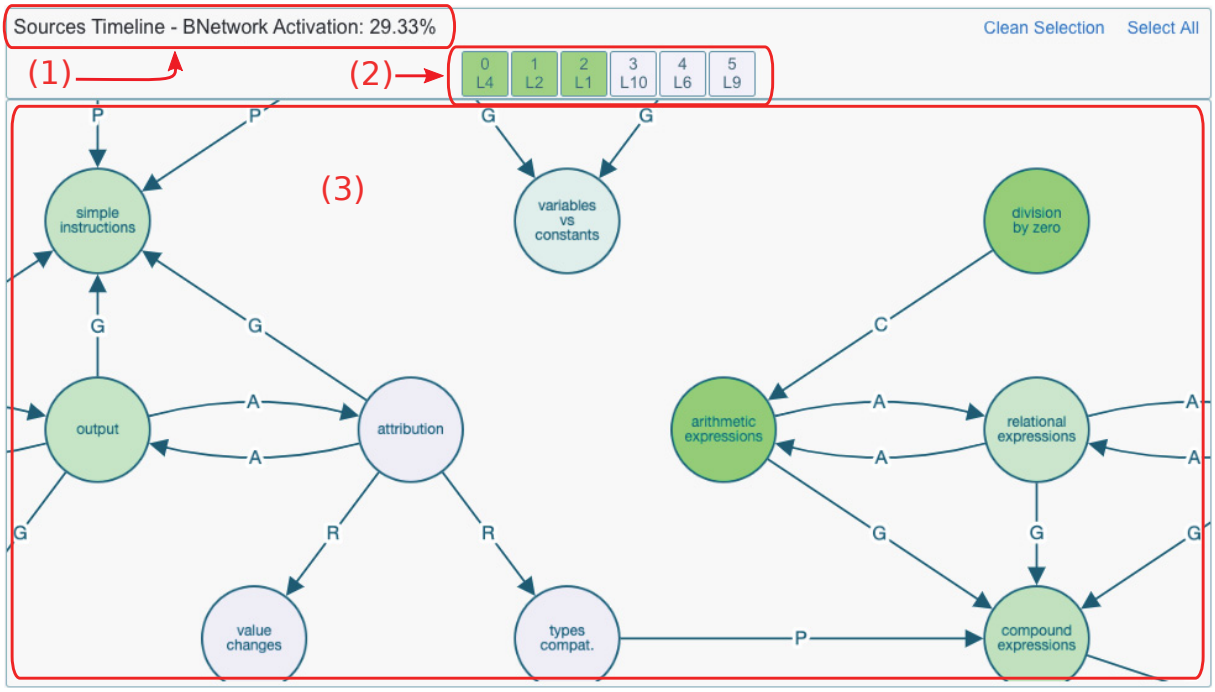


Figure 3.12: Evidence Machine learner model.

In addition, to inspect Bayesian Network details, a detailed view is available by accessing network nodes individually. Per source code detailed skill visualization permits to inspect each exercise answer and find out which evidences were extracted from it. By working with multiple sources of evidence (multiple source codes selected on the timeline, associated with inference mechanism) students' skills are valuated according to the maximum value metric (Koh et al., 2014), however, is important to identify exactly from where the valuation were extracted. Our approach is shown in Figure 3.13, where a Bayesian Node inspection view is presented, it is possible to note two evidences that affects the previewed node: (1) an inference from node *n2*; (2) an algorithm responsible for applying test cases strategy. Also, besides identifying the source of each evidence, a complementary view (shown with the *Details* button) allows analyzing algorithms execution logs (previously presented as the example from Figure 3.9).

To execute the algorithms and feed the learner model, Evidence Machine requires student source codes to be organized according dataset specification. The following section details this specification.

3.5 SOURCE CODE DATASET

This section presents the guidelines used to create the source code dataset used to verify the Evidence Machine effectiveness. The following sections expose the Evidence Machine dataset specification, ITS integration methods and a dataset collection tool for standalone use (without an integrated ITS).

3.5.1 Evidence Machine Dataset Specification

Evidence Machine dataset specification was created to allow the strategies execution with external, user-provided, source codes. A file/folder based structure was designed to ensure quick data manipulation and no-dependency installation. Figure 3.14 represents the dataset structure. Students and test cases are placed at the root level, students' folders can have any identification

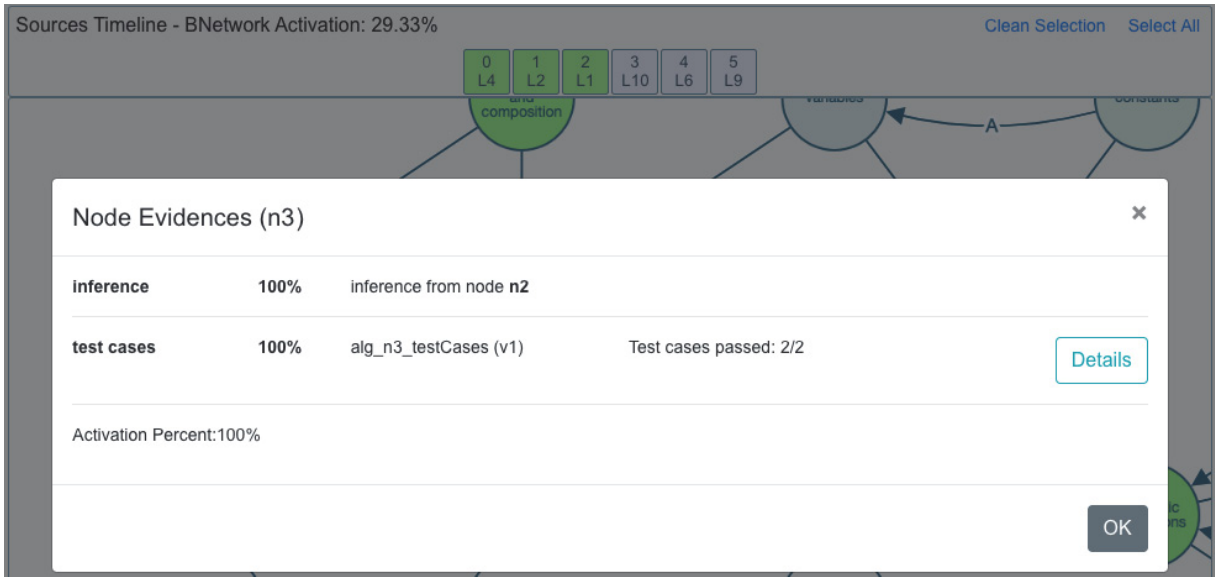


Figure 3.13: Evidence Machine Bayesian Network node evidence set.

method, however, for general purposes, it is suggested to use *S1* to *SN*, considering *N* the number of folders. Each student has exercise lists, strictly named from *L1* to *LN*. Each list has exercises, also strictly named according to the list and exercise numerical id, e.g., *L4E07.c* corresponds to the seventh C-Language program on the fourth list.

Test cases are organized by exercise lists also following the *L1* to *LN* nomenclature. Each sub-folder contains a PDF file that contains exercise specifications and sample input/output test cases. Each exercise must contain at least one test case, represented by the same nomenclature applied to the C-program file plus the test case identifier *M*, split into two separated files for inputs (*in*) and outputs (*out*).

Student exercise submissions (C-Language programs) must contain meta-data, represented by a comments header in the source code beginning. Meta-data header contains the attributes described in Table 3.5 and aims to provide essential information for the Evidence Machine execution, this information can be extracted from ITS or generated by external tools (e.g, standalone execution, described in Section 3.5.3). A sample answer source-code submission, with meta-data, is shown on Listing 3.7.

Listing 3.7: Sample source code with metadata.

```
1  //answer_id:313737303038302f4c312f4c314530322e63
2  //exercise:L1E02.c
3  //correct:true
4  //programming_lang:C
5  //date:2019-03-22
6  //originalfile_md5:E668609A260CA22F1FF7C4495E10A025
7
8  #include<stdio.h>
9
10 int main(){
11    return 0;
12 }
```

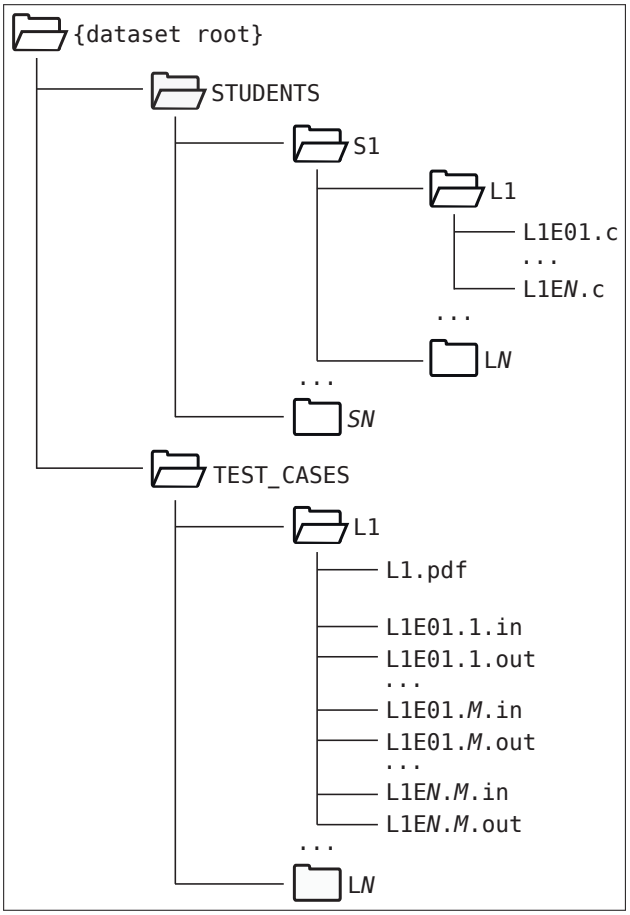


Figure 3.14: Dataset file/folder structure specification.

Table 3.5: Answer meta-data attributes.

Attribute	**Description**
answer_id	answer unique identifier, created from file path, relative to *STUDENTS* folder, converted to hex (eliminates special characters). E.g.: a sample file *S1/L1/L1E01.c* generates the id *53312f4c312f4c314530312e63*. Hex to String and String to Hex methods can be found on most of programming languages and can be applied for this conversion.
exercise	exercise strict name, e.g., *L02E04.c*.
correct	correctness flag, result of test cases application, boolean: *true* (passes all tests) or *false* (at least one test failed).
programming_lang	programming language in which the exercise was solved, e.g., *C*.
date	exercise submission date in *YYYY-MM-DD* date format.
originalfile_md5	student file md5 sum, used for plagiarism detection, prevents two students from sending exactly the same solution. This attribute must be generated before meta-data addition.

3.5.2 ITS Integration Methods

In some cases, creating a dataset may be unnecessary, such as when an ITS is already in use, or when students submit exercise resolutions on specific platforms. Evidence Machine can be applied to any system capable of providing student source codes that meet the required specification detailed in Section 3.5.1. To provide a way of using the Evidence Machine in

that cases, two forms of integration have been elaborated: (1) by calling API (Application Programming Interface) methods as an external service; (2) by extending the system with dataset adapters.

The first method, API based, was designed to be used when it is possible to make changes in the ITS. Evidence Machine can work as a standalone RESTful[12] web service, without end-user interface, fully operable through JSON (JavaScript Object Notation) based operations. ITS has full control of the dataset management and user interface, REST calls are performed in background transparently to the user.

With this method, the Evidence Machine is used as a completely isolated service, independent from the ITS. It is even possible to use a dedicated server for evidence processing, essential for large environments since this task can consume a lot of hardware resources over a long period. Also, this method gives the ITS programmer freedom as the student model and evidence search results can be handled in any desirable way. ITS is responsible for storing the evidence search results.

API method requires modifications to the ITS, usually done by creating plugins or modules. Figure 3.15 shows an example of the API method application. Evidence Machine is used as it is, without modifications (setup instructions available on Appendix D). A custom module, named Evidence Module, is created and added to the ITS. This module directly accesses to the database, reads information and converts it on JSON REST calls (REST API documentation is available in Appendix F). Evidence Machine server is then externally invoked (dashed arrows) by sending the student source code and evidence search parameters. Lastly, the JSON result is received by the ITS, which formats data for presentation.

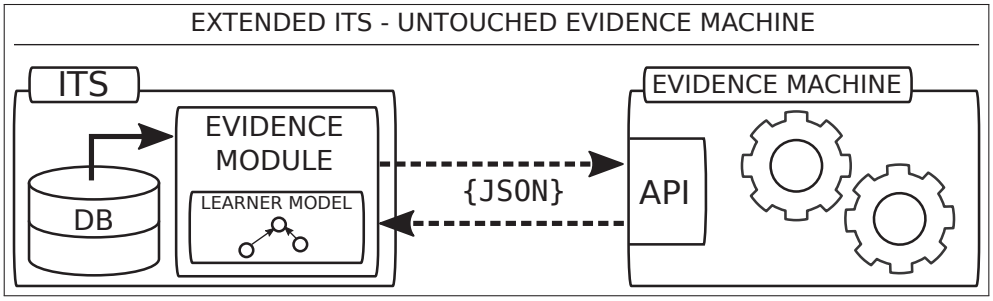


Figure 3.15: ITS integration: API method.

The second method was designed to be less intrusive to the ITS, requiring no modifications or modules. A read-only access to the ITS database still needed, but all modifications focus on the Evidence Machine, leaving the tutor system untouched. This method is useful when there's no access to ITS source code, when module/plugin creation is not available, or simply when tutor-side modifications are not desirable.

Evidence Machine, in this case, runs more actively, providing a full user interface that can be embedded or simply accessed as a regular website. The machine is also started as an isolated service, however, it must contain an adapter module directly connected to the ITS database. This adapter is responsible to read information and adapt it according to Section 3.5.1 specification, making the Evidence Machine work with ITS data instead of the regular file/folder structure.

Figure 3.16 shows an database adapter method example. The ITS remains untouched, tutor only provides an embedded web interface (e.g., a HTML *iframe* tag) or a simple hyperlink to the Evidence Machine server. Evidence Machine access ITS database, looks for students'

[12]Representational State Transfer (REST) Architecture.

source codes and test cases, adapts it, process and generates learner models. Evidence Machine locally stores processing results.

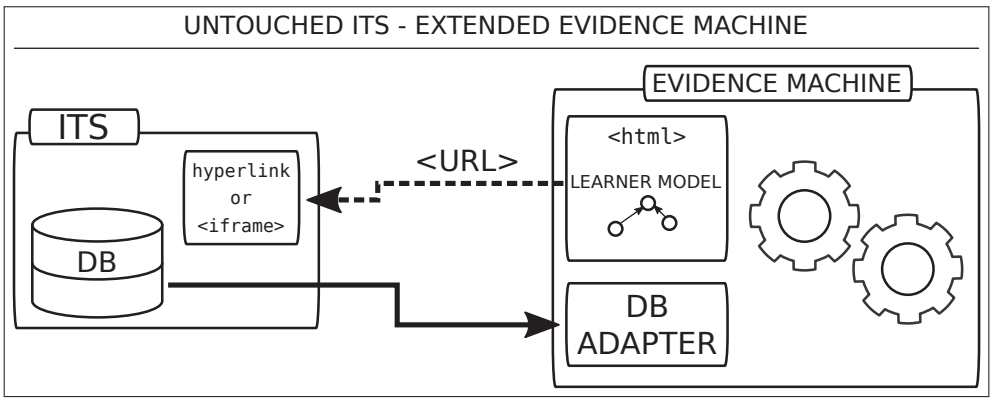


Figure 3.16: ITS integration: database adapter method.

3.5.3 Standalone Exercise Submitter

Evidence machine can be applied in more specific scenarios where there is no ITS running, however, collecting student-made source codes is still required. Exercise Submitter is a standalone dataset collection tool developed for this purpose. The following functional requirements guided the construction of this tool:

1. Tool must run remotely as a regular website, without user-side dependencies installation;
2. Teacher and students must be authenticated by username and password, both can change their passwords, the teacher can fully manage students;
3. Default exercise lists, with respective test cases, must be available right after tool initialization and exercises cannot be modifiable (favors the creation of standardized datasets, all students solve the same exercises);
4. Teacher must have exercises lists visibility and availability control;
5. Students must be capable of selecting an available exercise list and submit solutions;
6. Tool must be able to compile and execute student code, written in C language, with the default test cases, assigning correctness right after student submission;
7. All submitted solutions must be automatically organized in a file/folder structure according to the dataset specification described in Section 3.5.1.

The Exercise Submitter tool was built as a standalone, self-contained, web application. Installation and execution details are available in the Appendix E. Exercise Submitter functionalities will be described below, starting from the teachers and ending with students' perspective.

Right after username/password authentication, teacher is redirected to the main dashboard shown in Figure 3.17, where it is possible to access the several tool panels, check students' submissions and generate CSV (Comma-Separated Values) reports. Figure 3.18 also shows the charts-based students' progress overview (percentage values per student, and exercises list) available in the teacher's dashboard.

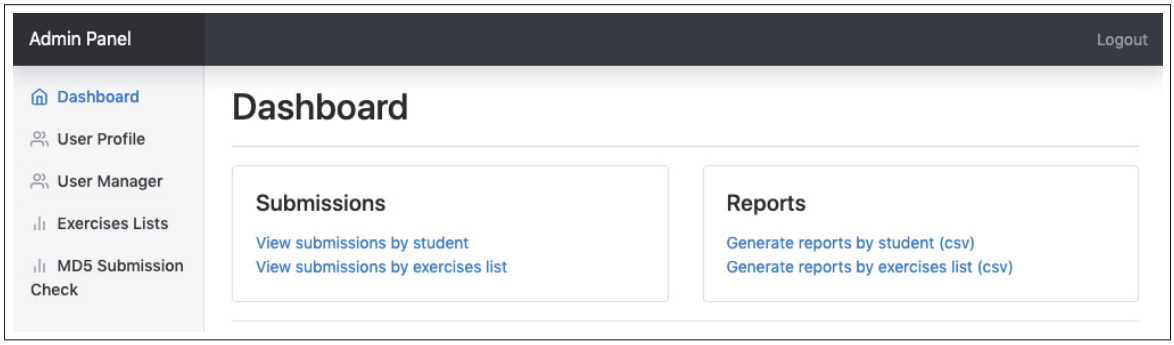


Figure 3.17: Exercise Submitter: teacher dashboard.

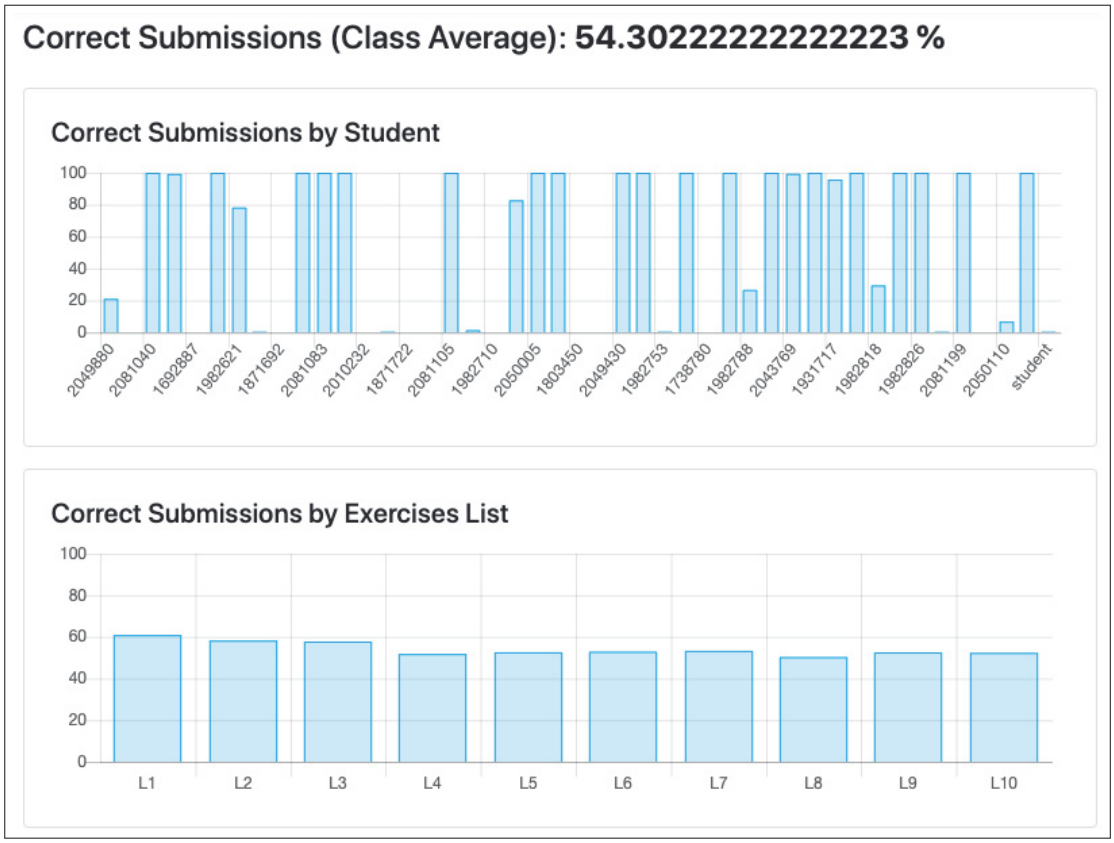


Figure 3.18: Exercise Submitter: teacher dashboard statistics.

Exercise lists management panel, shown in Figure 3.19, displays options to enable/disable submissions and change list visibility for the students. Ten default lists are available[13], labeled from L1 to L10, covering programming fundamental topics in consonance with the analysis presented in the Section 3.1. All exercise lists were written in Brazilian Portuguese as this is the students' native language. English translated topics shown on Figure 3.19 are: (L1) primitive data types, variables, operators, basic arithmetic expressions, input and output; (L2) conditional structures; and (L3) multiple choice structure.

Teachers' perspective allows tracking of exercise submissions by two panels: (1) *Submissions by Student* (Figure 3.20), with per-individual inspection, showing all lists and exercises of a single student; and (2) *Submissions by Exercises List* (Figure 3.21), where a single exercises list is focused, showing which students correctly submitted each exercise. In both cases,

[13]Default exercise lists are available online at http://bit.ly/doc_exsub.

Figure 3.19: Exercise Submitter: exercise lists management.

correct submissions are highlighted in green. The first visualization method permits a deeper look at a specific student, pointing lacks and potential difficulties faced. The second method shows a more general overview, allowing the location of exercises with high rates of failure (cases where few students had submitted correct solutions and extra attention must be taken by the teacher).

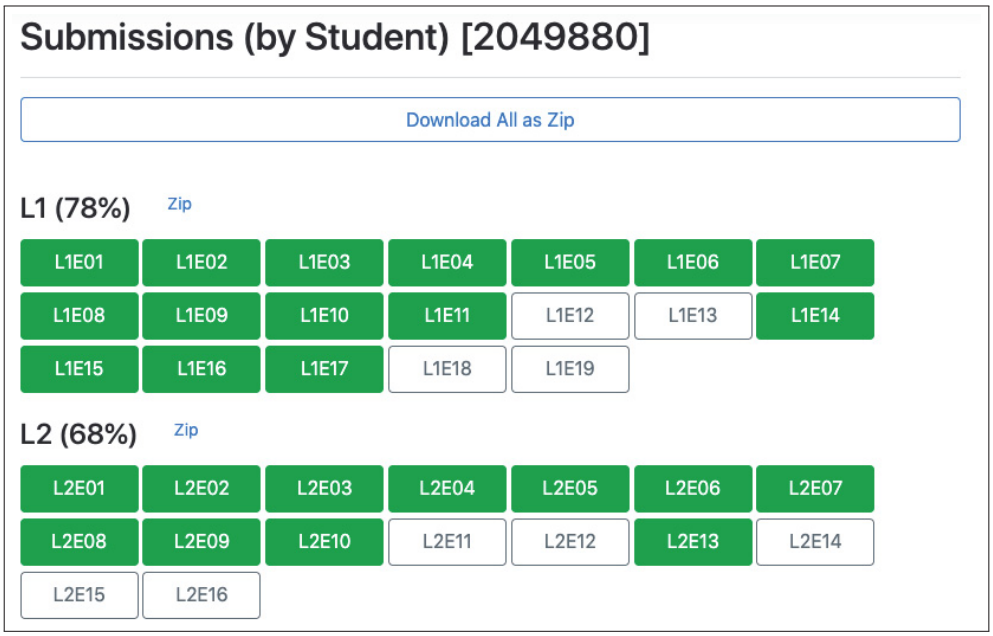


Figure 3.20: Exercise Submitter: submissions by student.

Once a submission is selected, teachers can view the student's code, correctness flag and last submission date (multiple submissions are allowed until the exercises list remains open, but the system stores just the last one). Figure 3.22 shows the submission visualization panel.

Additional panels were also made available: *User Profile* manages current user password and system language (Brazilian Portuguese is default, English translation is optional). This panel is available both for teacher and students; *User Manager* is a teacher-specific resource that allows creating students, password reset and login enable/disable; *MD5 Submission Check* permits to create an MD5 checksum skip, where specific exercises will not be checked against plagiarism (useful in cases where the exercise answer is too simple that multiple students can

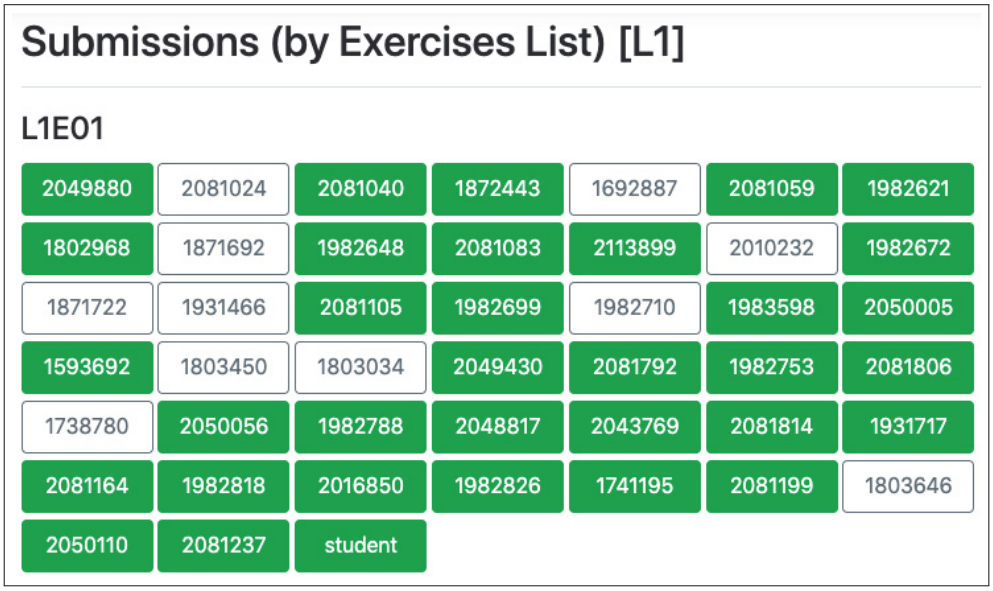


Figure 3.21: Exercise Submitter: submissions by exercise list.

Figure 3.22: Exercise Submitter: exercise source code.

achieve the same result, e.g., introductory *hello world*, *double of a number*, and *sum of two numbers* exercises).

Regarding the student perspective, the main dashboard is shown in Figure 3.23 highlights open submissions lists, student profile, and a self-progress overview chart. Chart points out the ten exercise lists with their respective correctness percentages as well as the average comparison metric.

Exercise lists panel, shown on Figure 3.24, leads students to code submission form. Each visible exercises list is shown, open submissions are highlighted in blue. Lastly, Figure 3.25 presents students submission form, where source code can be inserted and tested. All tries are automatically saved to the dataset, successful executions receives the comment-based header meta-data mentioned in Section 3.5.1, unsuccessful submissions results error feedback messages sent to the student, such as: invalid include file[14]; compilation error; and test case execution error.

The Exercise Submitter concluded the tool-set developed to support our experiments. The A-Learn EvId method was presented focusing on the computers programming domain.

[14]For security reasons, libraries inclusion was limited to the following files: *stdio.h*; *math.h*; and *string.h*.

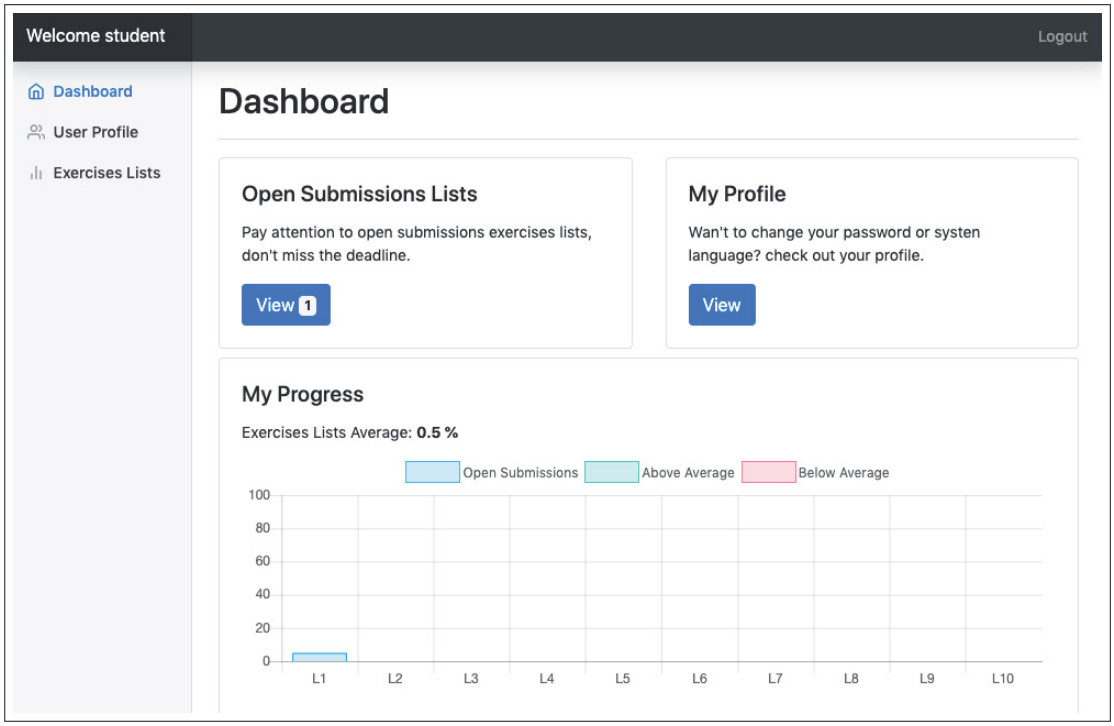


Figure 3.23: Exercise Submitter: student dashboard.

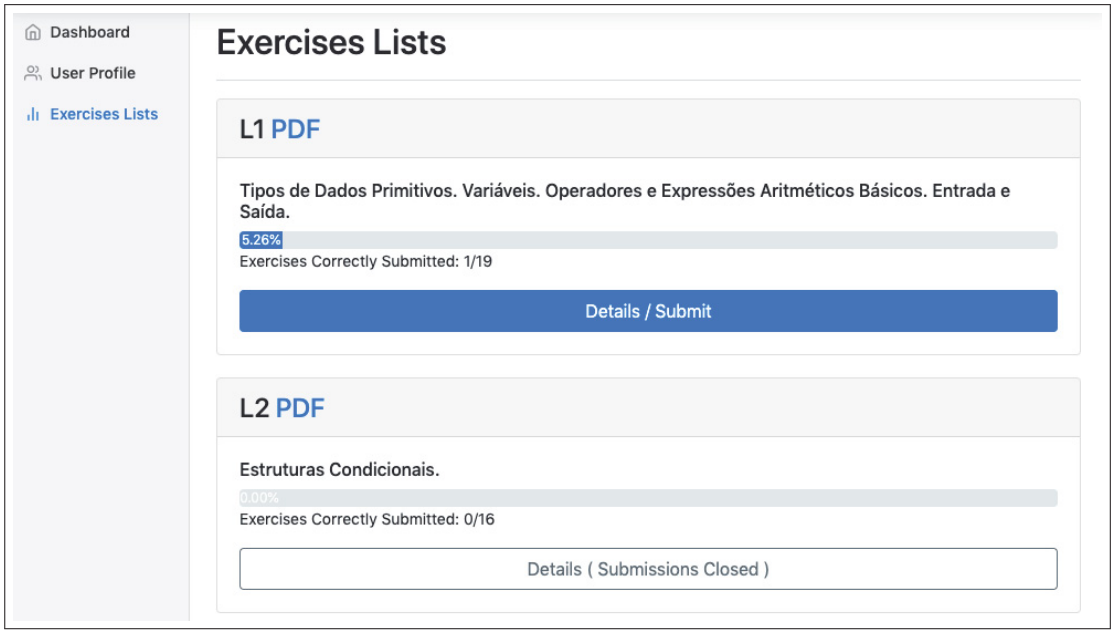


Figure 3.24: Exercise Submitter: student exercise lists panel.

Student-made source codes were taken as input data for the method. Programming skills were identified and selected to compose our learner model, implemented as a Dynamic Bayesian Network. Automatic source code analysis techniques were employed as strategies for identifying learning evidences in the input data. The Evidence Machine was presented as the practical application that makes up our implementation. Finally, the Exercise Submitter was presented as a dataset collection tool to allow testing our method in real-world source codes. The next chapter describes the experiments performed to investigate our method's capabilities.

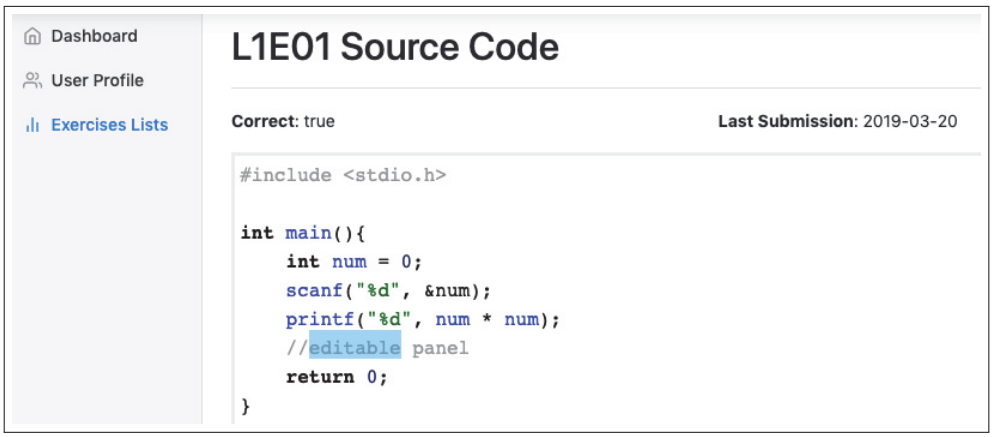


Figure 3.25: Exercise Submitter: student exercise submission panel.

4 EXPERIMENTS AND RESULTS

This chapter describes seven experiments performed to investigate the capabilities of the automatic learning evidences identification method and, thus, allow the objectives achievement and support argumentation about the contributions of this thesis. Some experiments were classified as pilot tests since they act as preliminary investigations for subsequent analyses (e.g., tests in a controlled environment). Experiments are structured according to the IMRaD format (Wu, 2011). Table 4.1 summarizes our experiments, where the structure is represented by four questions: *why* experiments were elaborated? (**I**ntroduction); *how* experiments were characterized? (**M**ethod and Materials); *what* we discovered? (**R**esults); **a**nd, *so what* does it mean? (**D**iscussion).

Table 4.1: Experiments summary (IMRaD structure).

Experiment	**Why?**	**How?**	**What?**	**So What?**
First (Pilot)	Investigate preliminary automatic source code analysis strategies.	*AST* and *parser* strategies were applied to a reference source code, results were then compared to human evaluation.	*AST* and *parser* strategies can detect learning evidences of *constants* programming topic.	Strategies presented good results on reference source code. Application on student-made source codes still needed.
Second	Investigate preliminary automatic source code analysis strategies in real scenario.	*AST* and *parser* strategies were applied to student-made source codes, results were then compared to human evaluation.	*AST* and *parser* strategies can detect learning evidences of *variables* and *constants* programming topics. Limitations were detected.	Experiment suggests automatic strategies can be feasible. Extended tests covering more programming topics and strategies still needed.
Third (Pilot)	Investigate static and dynamic automatic strategies to detect evidences 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 evaluation.	92.39% of human cataloged evidences were also identified by automatic strategies. Implementation limitations were detected.	Static and dynamic approaches were successfully applied to detect evidences. Strategies worked well on extended programming topics set.
Fourth	Investigate if strategies from the third experiment can also be accurate in real scenario.	The third experiment was replicated with student-made source codes extracted from a real STI.	Evidences identification capabilities in real environment were observed to be similar to the controlled scenario.	Strategies were successfully applied for detecting learning evidences, but implementation limitations still exist.
Fifth	Investigate using automatically identified evidences as data source for feeding learner model.	A Dynamic Bayesian Network was fed with automatically detected evidences. An empiric analysis was conducted to detect changes in the model.	Student model changes according to evidences inserted in the network.	Detecting evidences from multiple source codes and filtering them in the network permits monitoring students' skills progress.

Experiment	Why?	How?	What?	So What?
Sixth	Investigate evidence detection and students' progress monitoring considering skills commonly evaluated in real programming courses.	A priority skill-set was established through syllabi analysis, exercise lists were applied to real students. Students' progress 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 ITS as well as for students and their self-learning monitoring.
Seventh	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.

Additionally, Table 4.2 relates each experiment/pilot test with the method aspect it is intended to evaluate. Initial experiments focused on investigating automatic strategies capabilities. Fifth and Sixth experiments intended to analyze features of our learner model, and the last experiment covered the capabilities of the entire method.

Table 4.2: Experiments and method aspects.

Experiment	Method Aspect Evaluated
First (Pilot)	Automatic strategies: controlled scenario
Second	Automatic strategies: real scenario
Third (Pilot)	Automatic strategies (extended set): controlled scenario
Fourth	Automatic strategies (extended set): real scenario
Fifth	Learner model feeding and progress monitoring
Sixth	Learner model progress monitoring: real scenario
Seventh	Entire method: skills-based assessment: real scenario

Tests and experiments detailed in this chapter take human evaluations as reference parameters. Manual assessments and databases annotations were performed by the author of this thesis, defined as the human evaluator. Relevant characteristics from the human evaluator are: 13 years of computer programming experience; 7 years of teaching programming at university.

4.1 FIRST EXPERIMENT: PRELIMINARY METHOD FEASIBILITY (PILOT TEST)

4.1.1 Introduction

A method feasibility experiment was designed to preliminary evaluate the potential of *parser* and *AST* strategies as mechanisms for automatic identification of student learning evidences. The initial experiment is a pilot test and explores the ability of the parser to segment and isolate information from programming codes. A parser application results in a token tree that can be traversed by programming algorithms. Throughout the traverse process, it is possible to analyze the characteristics of each token and detect information that can be used as evidence of learning.

4.1.2 Method and Materials

Experiment is characterized by applying a parser to a predefined source code to locate evidences of the *constants* programming topic. Parser results are then compared to human evaluation.

Listing 4.1 shows the C-Language source code developed by the human evaluator and applied as input for the parser. The code contains a function (*main*), whose instructions consist of the declaration of four elements, being one variable and three constants. Also, the code has two assignment expressions and a required return statement.

Listing 4.1: Parser Input Source Code

```
1  int main ()
2  {
3     int a = 2;
4     const int b = 10;
5     const int c = 23.5;
6     const int d;
7     a = 4;
8     c = 6;
9
10    return 0;
11 }
```

Considering the concept of *constants* described by (Maschio, 2013), which (among other characteristics) concerns the domain of syntax and declaration of constants in a programming code. The listed source code has characteristics that allow investigating whether or not the student understands the concept of constants and, ideally, those characteristics must be detectable by the parser. Under the analysis of the human evaluator, the variable *a* and the constant *b* were initialized correctly, the constant *c* should have been initialized to an integer value, but it was incorrect when using floating point and, finally, constant *d* was not initialized. Moreover, observing the assignment operations, it is noted an attempt to change the constant *c*, which indicates a failure in concept understanding.

4.1.3 Results

Parser execution in the mentioned source code results in the AST summarized in Table 4.3. An algorithm capable of evaluating two evidences necessary to automatically evaluate the programming topic *constants* was developed considering the following characteristics: (1) constant declaration and initialization must be correct according to the programming language specification, and; (2) constant premise requires its value cannot be changed once it was defined.

Table 4.3: Parser's application result.

Declaration	**Type**	**Name**	**Attribution Type**	**Initialization Value**
FunctionDef	Function	main		
Declaration	Int	a	IntLiteral	2
Declaration	Int (const)	b	IntLiteral	10
Declaration	Int (const)	c	FloatLiteral	23.5
Declaration	Int (const)	d		
ExpressionStatement	Assign	a	IntLiteral	4
ExpressionStatement	Assign	c	IntLiteral	6

In the specific example cited, there are serious code flaws: (1) incorrect initialization of constant *c*; (2) non-initialization of constant *d*; and (3) attempt to change the value of constant *c*. All these indications were automatically detected through analysis on parser's output presented in Table 4.3.

4.1.4 Discussion

The preliminary experiment suggested the parser strategy is capable of detecting characteristics related to variables and constants, of which code flaws could be detected automatically. Thus, this information can be used as part of the mechanism to detect evidences of learning. Although results indicate the parser can extract details from each code unit that composes the source code, this experiment considered single-source code analysis only and further investigation is needed. Investigating parser capabilities in real student-made source codes still needs to ensure its functionality under generic conditions.

4.2 SECOND EXPERIMENT: METHOD FEASIBILITY IN REAL SCENARIO

4.2.1 Introduction

Following the presented viability test, an experiment was designed to evaluate the method under real conditions. The present experiment aims to identify if the parser is still effective on generic real source codes, and if there are problems or unexpected behaviors generated by students' syntax/programming style. For this, the method was applied to source codes produced by real students in a real programming course.

4.2.2 Method and Materials

Experiment is characterized by applying the parser on real source codes followed by the automatic identification of learning evidences for *variables* and *constants* programming topics. Parser results are then compared to human evaluation.

FARMA-Alg (Kutzke and Direne, 2015) is a teaching environment focused on computer programming which features solution development directly in the tool. Thus, records are stored of all actions taken by the student during the construction of the solution, including source codes. FARMA-Alg's database is extensive and growing daily as the tool is in constant use in undergraduate program subjects at various universities. Thus, a fraction of the database, consisting of 29 anonymous programs[1], was selected based on the following criteria:

- All source codes must be related to single-exercise resolutions to avoid large disparities between solutions;
- Exercise solution should require the use of variables;
- Programs must be written in C-Language;
- Solutions should always be extracted from exercises classified as correct by the ITS because certain error types, such as syntax errors, makes the code unparseable (impossible to apply the *parser* strategy). (Novais et al., 2016) also cited this limitation for static approach;

[1]Source code datasets used in our experiments are available online at http://bit.ly/doc_datasets.

- Source codes must be obtained from two separate classes, one file per student to improve the diversity of solutions.

The following evidences were then evaluated by the human evaluator and, subsequently, by the parser: (1) variable declaration capability; and (2) variables initialization. On item (2), the following characteristics were considered: (a) the use of compatible data types during initialization; and (b) initialization using the result of a mathematical expression. The first evidence was evaluated by an algorithm that uses the parser to validate whether the assigned data type is compatible, e.g., if there is no loss of information or accuracy. The last evidence was only flagged by the parser. Compatibility assessment between the data type resulting from the mathematical operation and the data type expected by the variable were not automatically verified.

4.2.3 Results

Figure 4.1 presents a chart comparing the results of the human evaluator and the parser taking into account the identified number of declarations and initializations. Chart data were extracted from the method execution in 29 source codes of an exercise that required the use of 4 variables per solution on average.

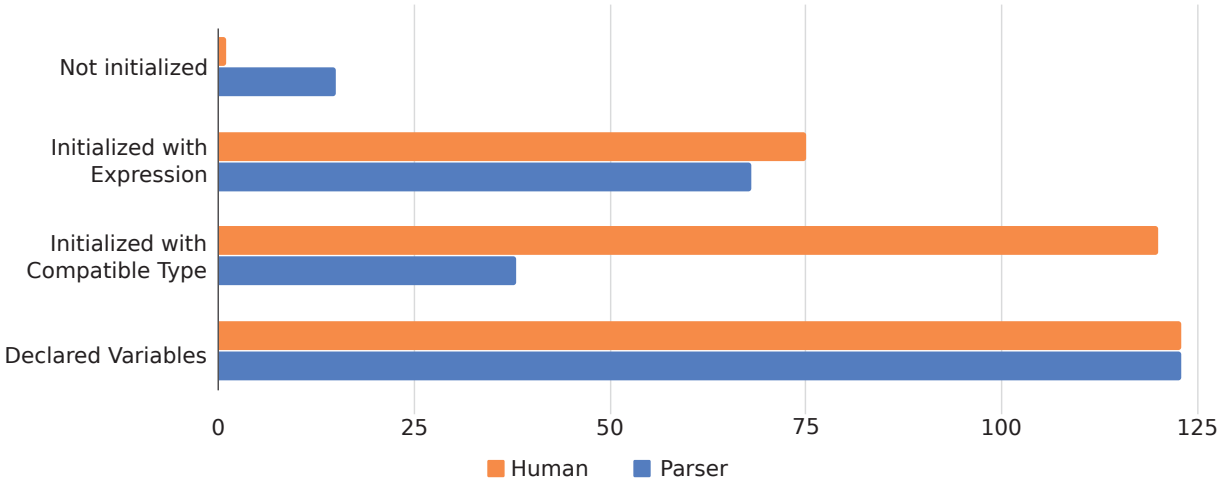


Figure 4.1: Parser vs human in real scenario.

Figure 4.1 shows that both the parser and the human evaluator detected the same number of variable declarations (121). It also shows that the identification of variables initialized with expressions was similar, but the parser did not detect 7 occurrences diagnosed by the human evaluator. Manual analysis of the data type used in initialization proved to be a deficiency of the parser algorithm, classified as implementation limitation because initializations with expressions are not fully evaluated by the algorithm (only flagged). Also, it is mentioned that 14 variables were incorrectly marked by the parser as not initialized.

4.2.4 Discussion

Parser's ability to act in real-world scenarios was evaluated, pointing to possible limitations on specific situations that can require complementary strategies. Besides that, C-Language source codes have been employed, however, any language capable of being interpreted by an AST parser can be contemplated with this evidence identification strategy, pointing out possibilities for more generic applications.

4.3 THIRD EXPERIMENT: CONTROLLED SCENARIO EXTENDED TEST (PILOT TEST)

4.3.1 Introduction

Method potential suggested by parser strategy experiments shown in the previous sections motivated the advance in research and the search for mechanisms capable of identifying other skills.

Automatic identification of variables and constants proved to be feasible, however, identification of more complex code units can be challenging and may require additional strategies. The present experiment is a pilot test to investigates if automatic strategies can be developed to identify learning evidences of input/output commands considering different data types and arguments. Also, if automatic strategies can go beyond static source code analysis, providing information also over runtime aspects.

4.3.2 Experimental Environment Definition

An experimentation environment was built as a web application composed of two modules: (1) user interface (front-end), which represents a Bayesian Network implemented with the Cytoscape.js library[2]; and (2) back-end module, responsible for strategies execution and results storage (evidence source for the Bayesian Network).

Bayesian Network was populated with a subset of skills, inspired by the skill set defined by (Maschio, 2013), based on concepts taught in early stages of programming courses, specifically in C-Language, considering the data types *int*, *float*, *char* and *char []*:

- *Types and Literals*: identification of declarations;
- *Input*: data read (*scanf*) and use of multiple arguments;
- *Output*: data print (*printf*) and use of multiple arguments;
- *Variables*: variables initialization;
- *Constants*: constants declaration and initialization, detection of constant value change try (student misconception);
- *Division by zero*: absence of division by zero when executing arithmetic expressions.

Given the subset of skills, strategies were developed to evaluate source codes and locate learning evidences. Strategies were elaborated as algorithms, which receive as input a source code and return as output a success percentage, which indicates if the student uses a certain programming resource correctly. Algorithm feedback also provides a detailed report of the localized evidences.

Strategies used for the composition of the evidence search algorithms were as follows: (a) location of patterns with *regular expressions*; (b) identification of code elements with *AST* and *parser*; (c) automatic *debug analysis* (*gdb*) associated with predefined *test cases*; and (d) catching exceptions with *execution traces analysis*.

[2]Cytoscape.js, graph theory (network) library for visualization and analysis. Website: http://js.cytoscape.org/

4.3.3 Method and Materials

The experiment applied automatic strategies to detect learning evidences for the six programming skills listed in the Section 4.3.2. Results were then compared to human evaluation.

The first testing step involves applying the method in a controlled scenario, contemplating a testing dataset consisting of source codes written specifically for this test. Source codes represent several situations, understood as common in the early stages of computer programming learning: declaration of variables and constants; input and output instructions; and arithmetic operations.

Twenty-nine C-Language programs were written by the human evaluator with objectives consistent with programming exercises applied to beginning students, such as reading and printing values, using different data types (*int*, *float*, *char*, and *char []*), numeric operations, conditional data printing, and data printing inside loops. Along with the source codes, test cases were written, containing input values and expected output. Such test cases are required for program feeding during automatic executions.

4.3.4 Results

The testing dataset made it possible to apply our strategies in source codes whose result is known. In the 29 testing programs, 92 evidences were inserted and manually cataloged, being classified according to the skills subset previously mentioned. Given the cataloged tests, the strategies were applied and evaluated. The result of each algorithm was then compared to the catalog constructed by the human evaluator. Figure 4.2 chart presents a comparison between the results of the automatic strategies and the manually cataloged evidences.

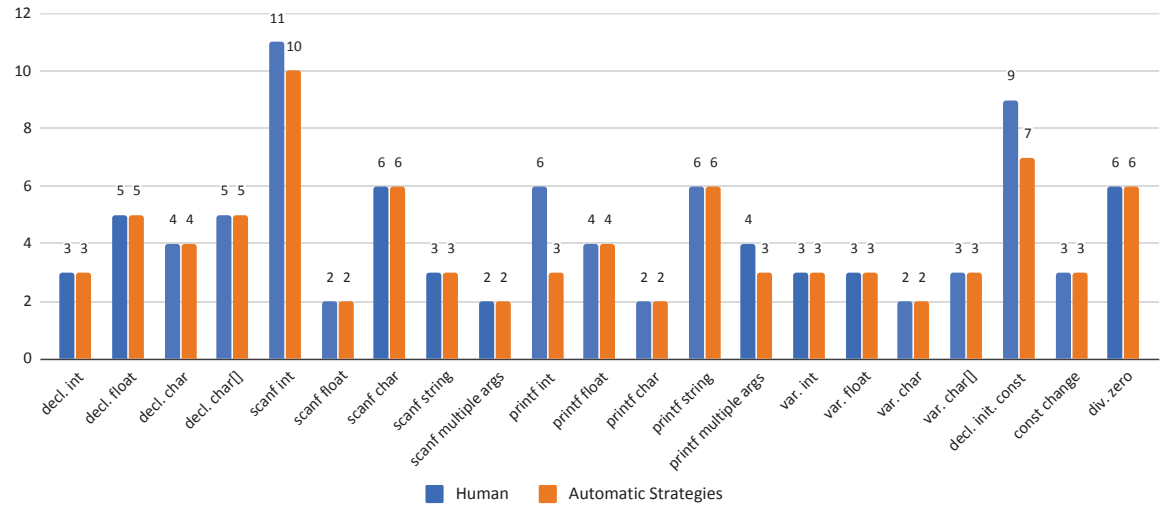


Figure 4.2: Extended experiment: controlled scenario results.

Our method was able to detect 85 of the 92 evidence inserted, totaling 92.39% accuracy. The highest error percentage occurred in the evidence related to integer data printing (*printf int*). A manual analysis revealed implementation-related issues as the strategy was unable to identify print command arguments when they are expressions and/or function calls. Thus, such arguments were automatically classified as incorrect, invalidating the evidence. Similar situations were found in the evidence *printf multiple args*. Also, the evidence *scanf int* presented an error related to an unhandled argument and, finally, the evidence *declaration and initialization of constants* suffered losses due to the mechanism not handling pointers.

4.3.5 Discussion

Automatic strategies were applied to detect learning evidences in a controlled scenario. Results suggest the method is accurate and promising as 92.39% of the manually cataloged evidences could be automatically detected by our strategies. Additional strategies were employed when compared to the first and second experiments, demonstrating capabilities to automatically detect evidences related to function calls, input and output commands with different combinations of arguments and data types. This shows strategies are not limited to detecting variables and constants. Also, the experiment demonstrated the potential of identifying also runtime aspects, such as division by zero, rather than only source code details provided by parser's static analysis.

4.4 FOURTH EXPERIMENT: REAL SCENARIO EXTENDED TEST

4.4.1 Introduction

Once results of the third experiment were obtained, the study was conducted to apply the method in a real scenario. This experiment replicates the tests conduced in the third experiment and aims to identify if the implemented strategies can also be accurate in real-world source codes. Also, check if real source codes can introduce problems/unexpected behaviors unseen in the testing dataset developed exclusively for the third experiment.

4.4.2 Method and Materials

This experiment was conducted on the experimental environment defined in Section 4.3.2, from which 113 source codes and their respective test cases were selected from FARMA-Alg's database (Kutzke and Direne, 2015). Criteria applied for sources selection were the following: (a) all codes should be extracted from the same class; (b) only students that submitted at least 4 solutions were selected, thus avoiding having multiple solutions of the same exercise and, consequently, increasing the source codes diversity; and (c) only submissions rated as correct were allowed (as mentioned before, avoids implementation related parser fails).

As performed on the third experiment, selected source codes were subjected to a manual evidence cataloging process. This process resulted in a total of 1020 manually localized evidence. Due to the nature and objectives of FARMA-Alg exercises, some features were not found in student source codes: declarations with data type *char*; reading and printing instructions with data types *char* and *char []*; and constants. A manual analysis revealed that FARMA-Alg's exercise lists mostly required the use of numeric data types and the use of constants was not mentioned (not a problem, but a characteristic of the exercises registered in the learning environment, which focuses on students who are new to programming.).

4.4.3 Results

As shown in Figure 4.3, results from applying the method in real environment reflected the evidence identification capability observed in the third experiment (Section 4.3). From the 1020 evidences, 971 were successfully located, totaling 95.2% of precision in automatic evidence identification.

An analysis of erroneous cases was performed, which revealed situations similar to those presented in the first experiment. In addition, errors were detected in some unforeseen situations during the deployment process, such as: failures in regular expressions responsible for detecting input commands (*scanf*) caused by syntactic disparities (e.g., white spaces between commands);

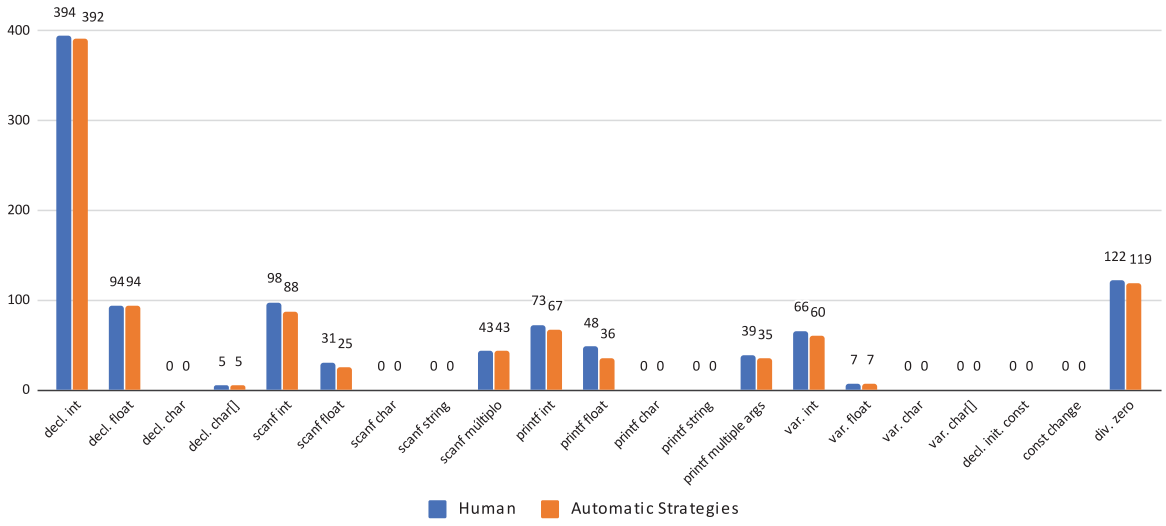


Figure 4.3: Extended experiment: real scenario results.

split assignment operations (/=) were not recognized as divisions; problems identifying input and output commands whose arguments are vector positions.

4.4.4 Discussion

Automatic strategies showed capable of identifying learning evidences in the real scenario. Results proved to be accurate and similar to the controlled scenario, encouraging strategies research expansion and application in real programming teaching support environments.

Evidence identification fails occurred and were classified as implementation limitations (as mentioned, unforeseen situations whose implementation was not prepared to deal with). Regular expressions were developed to match exact strings patterns, being vulnerable to small code changes. Although being useful for certain cases, care must be taken when considering evaluating generic source codes. Structural similarity-based strategies, such as AST and parser, are pointed as alternatives to avoid incorrect evaluations resulting from syntactic disparities.

4.5 FIFTH EXPERIMENT: LEARNER MODEL FEEDING

4.5.1 Introduction

Results achieved by the automatic evidence detection strategies in previous experiments stimulated the elaboration of a new experiment. This, in turn, aimed to verify the possibilities of using automatically identified evidences as data source for feeding the student model and for monitoring student skills evolution.

4.5.2 Method and Materials

This experiment was conducted with the experimental environment defined in Section 4.3.2, and started by constructing an evidence grouping mechanism with two criteria: (1) evidence grouping by source code and (2) source code grouping by student. Thus, data were organized into a hierarchy where students have source codes, which in turn contains evidences. Thus, it is possible to analyze the skill-set used by each student in each exercise resolution.

The same source codes were used as in third and fourth experiments. The clustering process resulted in 19 students, 6 corresponding to the third experiment controlled scenario database source codes, and 13 corresponding to the fourth experiment anonymous data extracted from FARMA-Alg's database (groupings were done by internal user ID, preserving student anonymity).

Evidence sets were then applied to the learner's model, presenting the skills to be mastered by the student throughout the learning process. A subset of these skills was selected for demonstration of the method, mirroring the evidence categories pointed out in the fourth experiment. All evidences were configured to have equal weights in the Bayesian Network.

Also, a mechanism for displaying and filtering source codes has been developed, allowing the selection of time intervals, which contains exercise solutions submitted by the student. This filtering mechanism can be viewed at the top of the Figure 4.4, where the blue spheres represent the source codes and the shaded area represents the active evidence range in the model. At the bottom of the mentioned figure, there is a fragment of the skill Bayesian Network built according to the Section 3.3 specification.

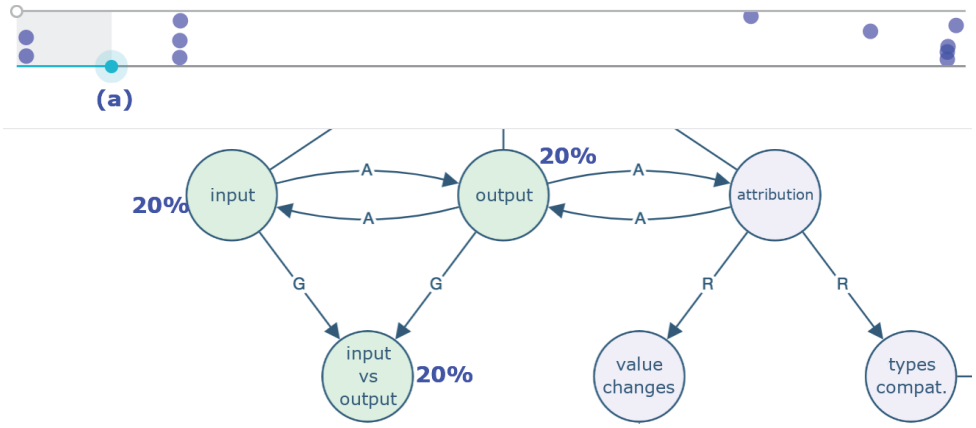


Figure 4.4: Learner model fed with evidences from two source codes.

4.5.3 Results

Results analysis considers Figures 4.4 and 4.5. Both figures represent data from the same real student, but at different time intervals. The state of the model in Figure 4.4 presents sets of evidence from two source codes, located to the left of the marker *(a)*, which infer 20% valuation on three nodes (*input*, *output* and *input vs output*). It is possible to see in Figure 4.5 changes in the state of the model: at the top, the *(a)* marker has shifted to the right, encompassing three new source codes, which add new sets of evidences responsible for increasing the valuation percentage in the three mentioned nodes.

An empirical analysis of the 19 generated models was performed. We observed evidence clusters and whether changes in the time interval reflect on changes in the network valuation state. In line with the example presented, in all cases the mechanism was able to group the evidence by source code and display the state of the model according to the selected time interval.

4.5.4 Discussion

The present experiment successfully employed automatic strategies for identifying basic programming resources, followed by applying this information to feed a learner model prototype. Results suggest the employed method can be used for continuous monitoring student progress by

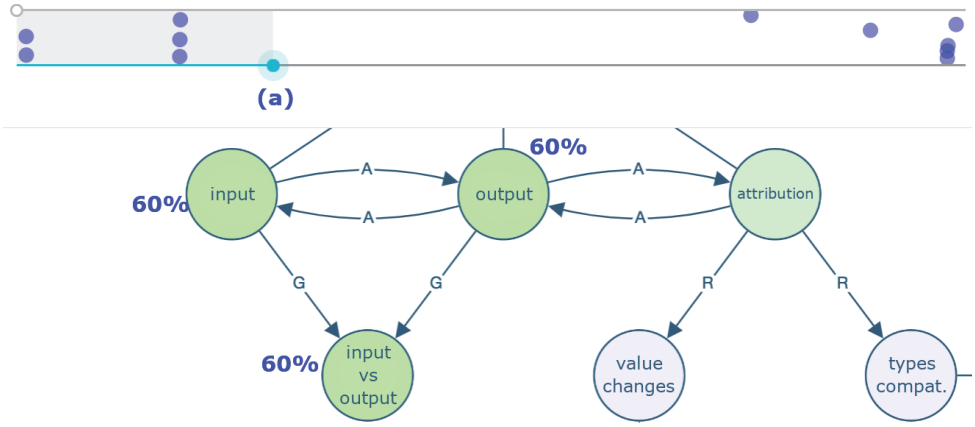


Figure 4.5: Learner model fed with evidences from five source codes.

identifying evidences from individual source codes associated with a timeline-based grouping mechanism.

Evidence grouping and filtering method proved to be efficient, providing instant visualization of different states of the student model following the concept of Dynamic Bayesian Network from (Neapolitan, 2003). Although the experiment was based only on basic programming topics and a limited number of strategies, results are encouraging to expand research, implement new strategies and represent most advanced programming skills in the learner model.

4.6 SIXTH EXPERIMENT: ANALYSING STUDENT PROGRESS IN PRIORITY SKILLS

4.6.1 Introduction

In the real scenario from the fourth experiment, we noted that some programming resources were not employed by students and, consequently, not evaluated in the automatic versus human comparison. This fact occurred because the chosen FARMA-Alg's exercise subset simply did not require these programming resources to achieve a correct solution. A new experiment was designed to provide analysis of a more diverse scenario, focusing on applying automatic strategies to detect evidences and monitor student progress considering skills commonly evaluated in real programming courses.

4.6.2 Source Code Dataset

Considering FARMA-Alg's exercise subset limitations from fourth experiment, we decided to create a new dataset to cover all skills classified as common in real-world courses. Using the Exercise Submitter tool (Section 3.5.3), new source codes were collected as no existing dataset was found with the necessary information for applying the strategies for the *priority skills* subset. The dataset created for this experiment follows Section 3.5 guidelines and contains 3860 source code files, written by students as answers to 101 programming exercises. Exercises were grouped into 10 lists (subsets of exercises), and source codes were written in C-Language by 39 students of an introductory programming course, part of a bachelor's degree in mechanical engineering from the federal university *Universidade Tecnológica Federal do Paraná (UTFPR)*. A "reference student" whose answers were provided by the teacher was also included in the dataset, besides that, real students were not identified, making the dataset completely anonymous. The following topics were covered by the ten exercises lists:

- L1: Primitive data types, variables, operators, basic arithmetic expressions, input and output;
- L2: Conditional structures;
- L3: Multiple choice structure;
- L4: Pre-test repetition structure: while;
- L5: Post-test repetition structure: do-while;
- L6: Counted loop repetition structure: for;
- L7: Uni-dimensional data structures: vectors;
- L8: Strings;
- L9: Bi-dimensional data structures: matrices; and
- L10: Functions.

Each source code was automatically annotated by Exercise Submitter with a metadata header which describes the following attributes: unique identifier, corresponding exercise, correctness flag, source code programming language, date of submission. Answer correctness was evaluated by previously defined *test cases* strategy when the student submits the solution. All answers are unique and any cloned source code was discarded.

4.6.3 Method and Materials

The syllabi analysis presented on Section 3.1.2 revealed ten common programming topics approached in real courses (Table 3.3). From these ten topics, the *priority skills* subset was defined as the basis for the current experiment, aiming to apply strategies and identify progress on students' skills valuation along the time. To recap, *priority skills* are: *output, input, types of literals, variables, arithmetic expressions, relational expressions, boolean expressions, multiple selection conditional, simple and compound conditionals, infinite loops, counted loops, pre-evaluated, post evaluated, arrays, matrices and functions*.

Evidence identification mechanisms employed for this experiment were implemented by using *test cases*, *AST*, *parser* and *regular expressions* strategies organized in *hybrid* combinations. Strategies description were presented in Section 3.2, also extended skills implementation details can be found on Appendix C.

Given the strategies built to cover the *priority skills* subset, each source code from the dataset was submitted to automatic analysis. Resulting data was stored and organized to discover how can the automatic strategies results be useful for analyzing evidence of progress in students' skills. For this experiment, progress is defined as the increase in the valuation of a given skill when new evidence sets are inserted. New evidence sets are added each time a new exercise list is submitted (the timeline step is marked by exercise lists).

Each strategy returns a value from 0 to 100 for the related concept, representing the correctness percentage a specific student applied the given programming resource. Considering students submit several exercises during the course, the same evidence may have different values in more than one source code. For this experiment, (Koh et al., 2014) metric (maximum value) was employed as the decision criterion.

4.6.4 Results

First analysis, shown in Figure 4.6, features a chart comparing the students' average progress with the reference solutions provided by the teacher. The vertical axis represents the percentage of programming skills covered by students according to the nine topics listed in Table 3.3. The horizontal axis represents the exercise lists assigned to students during the course. The figure shows students covered about 28% of the concepts after completing L1, and that coverage rate grew up as students progressed in the exercise lists. Moreover, the figure also shows students presented a coverage similar to the reference solutions, indicating students applied a similar subset of the concepts used by the teacher to solve the same problems.

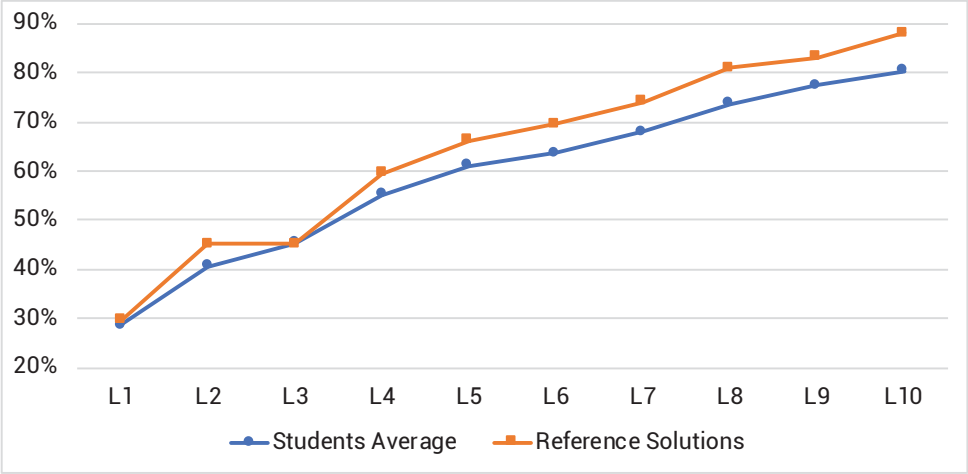


Figure 4.6: Students average knowledge comparison.

Figure 4.7 illustrates the individual progress of students, representing new concepts applied from the first to the second list of exercises. The blue-colored part of the bars represents the knowledge detected for L1, while the orange-dotted part represents new skills employed by the students when programming to solve exercises for L2. Comparing lists 1 and 2, all students but two (25 and 30) used resources related to new programming concepts, suggesting learning progress. The other two students (27 and 39) did not submit the source code for the first list and therefore presented no concepts for L1.

Figure 4.8 presents an example of the progress of a single student (represented by the identifier 6 in Figure 4.7), according to the ten lists of exercises. In each list, different concepts were applied. Concepts, such as *operators and expressions* and *data types*, were applied since the first exercises list; while more advanced concepts, such as *matrices* and *functions*, were only used in exercises from the last lists. The figure also shows how the student evolved when applying the concepts. Example: the student started using *vectors* by L7 indicating partial success, and evolved in L8 obtaining success for all the related evidence, suggesting improvements from one list to another.

The evidence search also allows the analysis of specific concepts, by specific students, in specific exercises. Figure 4.9 shows an example of a source code submitted for an exercise in List 9 (*matrices*). The right panel presents samples of evidence found by the strategies, describing what programming concepts the student applied and showing them in the source code. Table 4.4 presents the full set of evidences found in this example.

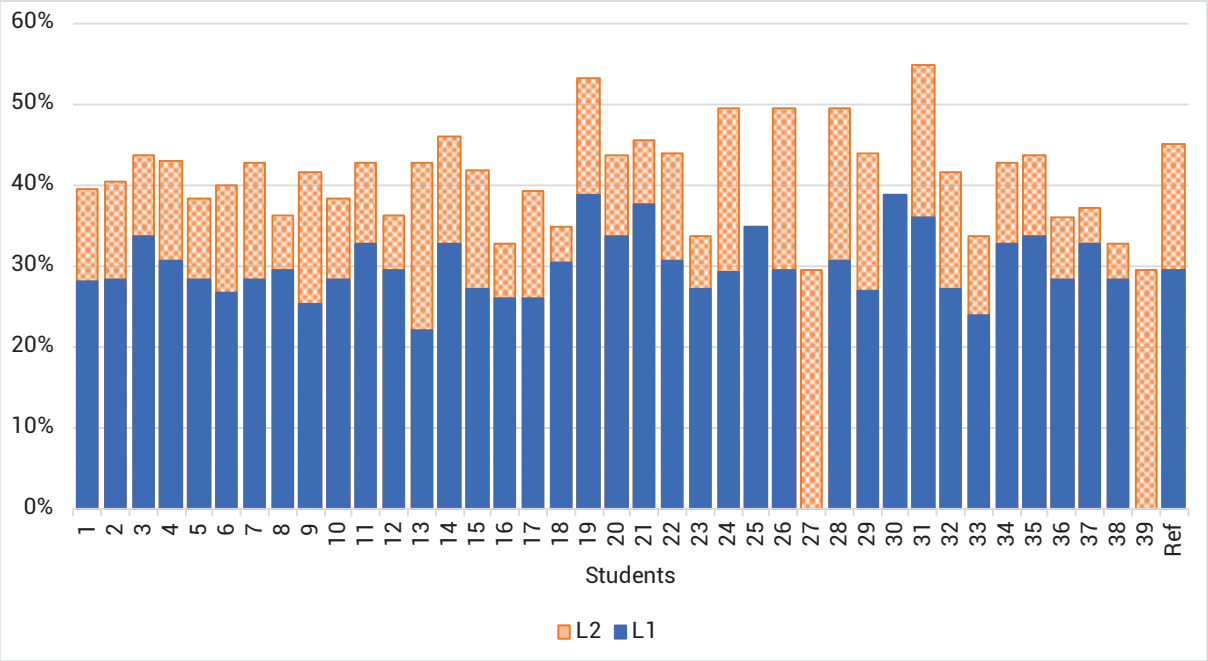


Figure 4.7: Students progress between two exercise lists.

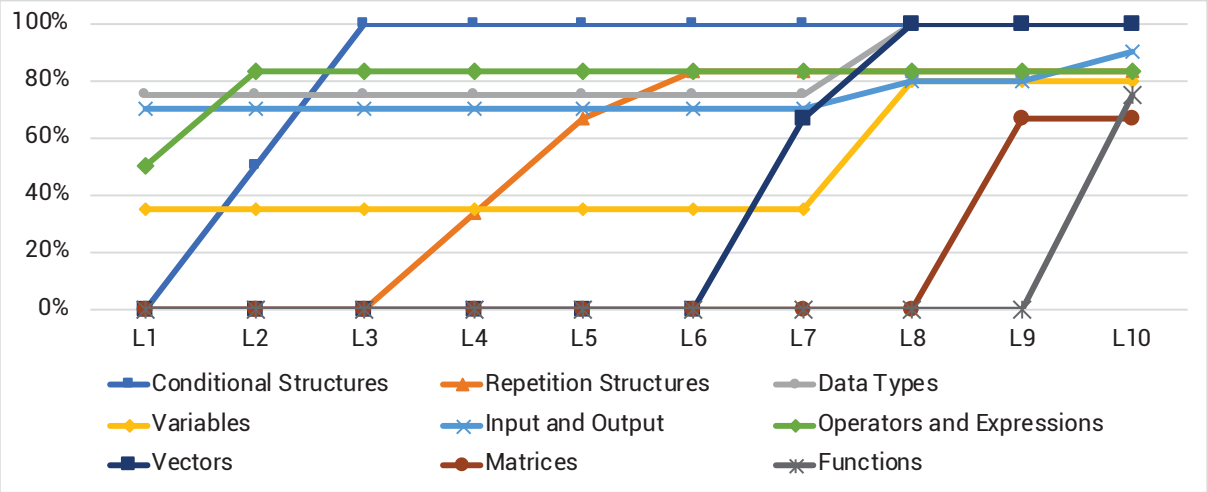


Figure 4.8: Example of a single student progress across the ten lists.

4.6.5 Discussion

Students' skills progress analysis was presented based on the most common programming topics covered in Computing courses of ten Brazilian universities. A set of strategies was applied to cover nine programming concepts and to identify learning evidence in students' source codes collected from a real introductory course.

Results were analyzed to verify how students apply programming resources as they progress through the course. Analysis suggests the automatic strategies can detect learning evidence for real-world common topics and are promising to support teachers when analyzing the source code produced by students. Strategies can map where specific concepts have been correctly applied by students. Furthermore, strategies offer details about the structure of the source code produced by the student, indicating which programming concepts were applied in each solution.

```
1  #include <stdio.h>
2
3  int main() {
4    int matriz[4][4], i, j;
5    int soma = 0;
6    for (i = 0; i <= 3; i++) {
7      for (j = 0; j <= 3; j++) {
8        scanf(" %d", & matriz[i][j]);
9        if (i > j) {
10         soma = soma + matriz[i][j];
11       }
12     }
13   }
14   printf("soma:%d", soma);
15   return 0;
16 }
```

Evidence Found: relational expression (less or equal)
i <= 3

Evidence Found: relational expression (more than)
i > j

Evidence Found: (int) matrix declaration
int matriz[4][4], i, j;
- size: [4][4]
- initialized in decl? false
- populated with scanf? true, in 8:0
  => scanf(" %d", &matriz[i][j]);
- occurrences in source: 3

Evidence Found: simple if
if (i > j) {
    soma = soma + matriz[i][j];
}

Figure 4.9: Source code and evidence search result.

Table 4.4: Evidence set found in Figure 4.9 source code.

Programming Topic	**Evidence Found**
Conditional Structures	*if*
Repetition Structures	incremental counted loop *for*
Data Types	declaration with type *int*
Variables	variable initialization evaluation with type *int*
Input and Output	*printf* with type *int*
Operators and Expressions	arithmetic expression *add* relational expression *less or equal* relational expression *more than*
Vectors	
Matrices	declaration and use of matrices of type *int*
Functions	

The learner model provided resources to monitor student progress across exercise lists, indicating whether there is progress or not in the evaluated skills. Detecting (the lack of) such progress can offer useful insights for both teachers and ITS as well as for students and their self-learning monitoring.

Besides that, analysis shown it is possible to monitor individual skills instead of the student grading as a whole. This information can be useful to detect programming topics that require special attention from the teacher, e.g., students can fail repetition structure exercise lists due to lack of skills in conditional structures, thus requiring reinforcement of programming concepts prior to loop related activities.

4.7 SEVENTH EXPERIMENT: HIGH LEVEL SKILLS-BASED ASSESSMENT

4.7.1 Introduction

Previous experiment results have suggested automatic skills identification is a feasible option as a support tool for correcting programming activities. An additional experiment was then designed to demonstrate, with more details, the possibilities provided by the skills-based assessment, thus reinforcing present thesis contributions. Therefore, the present experiment demonstrates the use of automatic mechanisms as a means of identifying functionally correct but conceptually incorrect solutions.

Figure 4.10 presents a hypothetical situation where the use of subterfuge allowed the student to build a source code capable of returning a correct result (functional point of view), but incorrect when considering the activity requirements (conceptual point of view). The exercise in question required the student to use a loop to print integers in the range from 1 to 10. Looking at the solution provided, the required resources were not employed as in the reference solution, however, the executions of both programs give identical results, being erroneously classified as a correct exercise when evaluated with the *test cases* strategy for example.

reference solution

```
#include <stdio.h>

int main()
{
    for(int i=1; i<=10; i++)
    {
        printf("%d ", i);
    }
}
```

student solution

```
#include <stdio.h>

int main()
{
    printf("1 2 3 4 5 6 7 8 9 10 ");
}
```

Figure 4.10: Solutions comparison: reference vs conceptually incorrect.

4.7.2 Method and Materials

Skills-based assessment was implemented as an Evidence Machine resource to investigate the use of automated strategies as mechanisms for identifying potential situations where subterfuges were used by the students. The experiment is described according to the following steps: (1) source code dataset definition; (2) definition of desired skills in each exercise; and (3) method application.

Source Code Dataset: the dataset used in this experiment was collected using the methodology detailed in the Section 4.6.2, however, the ten default exercise lists were revised and updated to better fit the course objectives, resulting in a total of 84 exercises (previously 101)[3]. Applying the new exercise lists resulted in collection of 4434 unique source codes from two real courses, totaling 71 students, plus 84 reference solutions.

Desired Skills: definition of desired skills aims to create a formal specification for the 84 dataset exercises, where each exercise is associated with a set of skills and their respective desired valuations (from 0 to 100%). The 37 skills defined in the Section 3.1 were considered, contemplating the following sets: *priority skills*, *complementary skills* and *potentially solved by inference*. Figure 4.11 presents an example of desired skill mapping, where each programming exercise has a set of requirements.

L1E01.c

effectuation	100%
input	20%
variables	5%

L1E02.c

effectuation	100%
input	20%
output	40%
variables	5%

...

L10E07.c

effectuation	100%
structuring and composition	100%
simple instructions	25%
functions	20%

Figure 4.11: Per-exercise desired skills, sample mapping.

Desired skills mapping allows the teacher to create conceptual requirements for each activity, providing topic-specific assessment and, implicitly, forcing the student to apply deter-

[3]Note: L1 to L10 topics and programming language were not changed. The major changes were concentrated on the removal and replacement of preliminary L1 exercises.

mined programming resources. Skills specification should reflect the objectives of each activity (what the teacher wants to evaluate in each exercise) and, ideally, should be part of the exercise elaboration.

For experimental purposes, desired skills mappings were created based on the 84 reference solutions. Automatic strategies were applied to identify evidence for the 37 aforementioned skills, this way valuating all skills for the 84 reference solutions. By using these solutions we assume, for this experiment, the student should, at least, use the same programming resources applied in the reference source codes.

Method Appliction: given the dataset, automatic strategies are applied to identify learning evidences according to the desired skills mappings. Results from automatic strategies represent skills identified in students' source codes. Identified skills are then compared to the desired skills, composing our skills-based assessment method. Figure 4.12 shows the visual representation of the conceptual correctness calculation used in the remainder of this experiment. Each skill has a required and an identified percentage, the rate between these values composes the skill evaluation. The skills evaluations arithmetic mean provides the source code skill-based assessment result (exercise conceptual evaluation).

Assessment Output

Skill	Required (%)	Identified (%)	Skill Evaluation (%)
skill_A	req_A	val_A	ev_A = val_A / req_A
skill_B	req_B	val_B	ev_B = val_B / req_B

Skill Based Assessment (Average): (ev_A + ev_B) / 2

Figure 4.12: Skill-based assessment calculation reference.

4.7.3 Results

Skills-based assessment were applied to the 4434 dataset source codes, results are shown in the Figure 4.13 chart. The vertical axis corresponds to the skill-based assessment average, the horizontal axis represents the exercise averages (considering all students). Most exercises (75%) achieved an average equal to or higher than reference solutions, while only 25% received lower ratings (indicating that not all skills identified in reference solutions were found in student solutions). Both lower and higher classifications are considered subterfuge inspection alert areas. Lower classifications can be evidence of lack of resources, indicating students potentially applied unappropriated programming resources to achieve the functional correctness. Higher classifications can also be evidence of conceptually incorrect solutions, indicating students have developed a more complex program, or used excessively unnecessary resources, when compared to the reference source code.

Figure 4.14 details the left segment (lower rating portion of the solutions) of the overall graph shown in Figure 4.13. This kind of view allows the teacher to obtain data that enables the discovery of activities that are, on average, causing the most confusion in students, where unexpected solutions are prevalent. In the chart, solutions with lower assessments are candidates for manual inspection, prioritizing those with the largest discrepancy to the reference source codes (from L9E02 to L5E06). Applying this same form of visualization in an individual context, considering single students instead of the general average, guided by the activities that presented more discrepant evaluations in the general context, it was possible to identify functionally correct but conceptually incorrect solutions. Some examples are presented below.

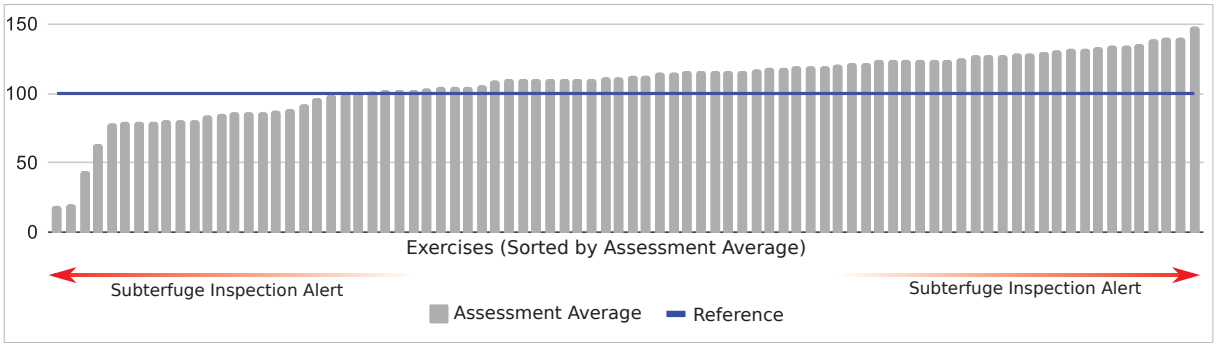


Figure 4.13: Skills-based assessment results (all students average).

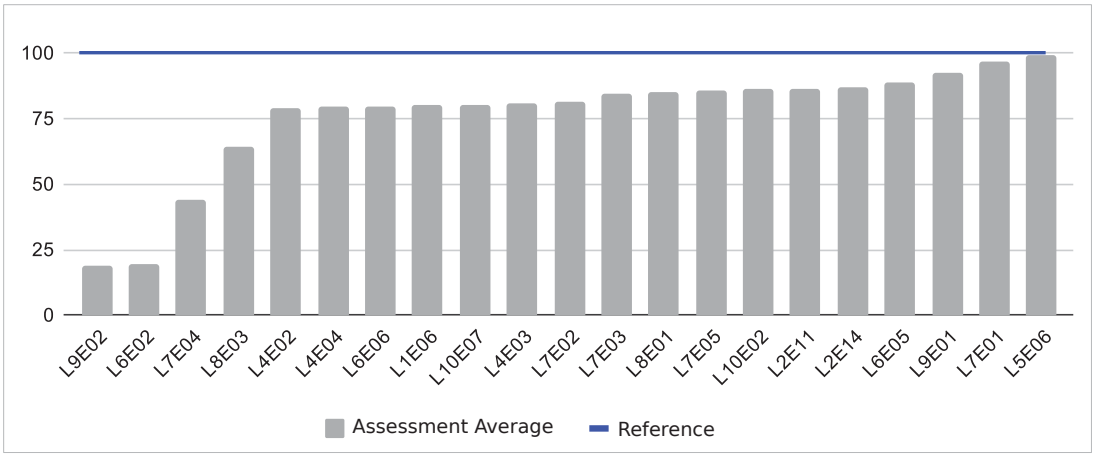


Figure 4.14: Skills-based assessment partial chart detailed view.

Figure 4.15 presents a sample assessment where the skills evaluation was classified under the reference line. Skills not identified (or with lower valuation) can point to possible failures and subterfuges (in this case the exercise required the use of a user-made printing function, not employed on student's solution). Also, skill evaluation can go over 100% when students use more programming resources that needed (not always a problem, but can point to exercises that require teacher's special attention: *why students are creating so complex solutions?*), in this case, the presence of an unused integer variable is highlighted.

student source

```
#include <stdio.h>

int main()
{
   int w;
   printf("Ufa! Este foi o ultimo exercicio! Acabei Computacao 1 =D");
   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	20.00	21.00	105.00
functions	25.00	0.00	0.00

Skill Based Assessment (Average): 76.25%

Figure 4.15: Source code skill-based assessment inspection (lower value).

Another example of the lack of skills is shown in the Figure 4.16, which consists of 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 the use of *test cases*, but the addition of conceptual constraints (such as the requirement for higher valuation of loop-related concepts) prevents the solution from achieving a good score and, consequently, be indicated 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 4.16: Source code skill-based assessment inspection (lower value, second example).

Looking at the right portion of the graph, functionally correct solutions, but built using more features than reference solutions were found. Figure 4.17 shows an example of a solution discovered by inspecting the activities of this group. The activity is designed with the following requirements: *create a program that reads three numbers, for each number print its double, use a function that takes as a parameter an integer and returns its double, the calculated value must be printed on the main function*. The student misinterpreted the problem's assignment since three functions were created with the same purpose (double calculation). Also, there was an erroneous attempt to use a user-made print function (*imprime_resultado*), whose behavior was not coded and the data printing is performed by an instruction that executes outside and after that function call.

Another example where excess skills led to identifying concept failures is presented in Figure 4.18. 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 ua(int v1){
  return v1*2;
}
int dobro2(int v2){
  return v2*2;
}
int dobro3(int v3){
  return v3*2;
}
void imprime_resultado(int resultado){

}
int main(){
  int v1, v2, v3, resultado;

  scanf("%d %d %d", &v1, &v2, &v3);

  resultado = ua(v1);
  imprime_resultado(resultado);
  printf("dobro de %d eh: %d\n", v1, resultado);
  resultado = dobro2(v2);
  imprime_resultado(resultado);
  printf("dobro de %d eh: %d\n", v2, resultado);
  resultado = dobro3(v3);
  imprime_resultado(resultado);
  printf("dobro de %d eh: %d\n", v3, resultado);
  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	37.00	148.00
input	20.00	40.00	200.00
input vs output	10.00	40.00	400.00
types of literals	25.00	25.00	100.00
variables	5.00	5.00	100.00
constants	8.33	8.33	100.00
variables vs constants	6.63	6.63	100.00
arithmetic expressions	16.67	16.67	100.00
compound expressions	5.67	5.67	100.00
control structures	1.87	1.87	100.00
functions	25.00	50.00	200.00

Skill Based Assessment (Average): 142.15%

Figure 4.17: Source code skill-based assessment inspection (higher value).

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 4.18: Source code skill-based assessment inspection (higher value, second example).

Deficiencies in certain skills can lead the student to construct inadequate solutions with the knowledge already acquired, often requiring even greater effort than would be employed in learning new skills. Also, assigning values to a large number of skills is not always indicative of good solutions. To exemplify this situation, the Figure 4.19 shows a sample solution to the following problem: *create a character array vetA and initialize it with the word "COMPUTACAO". Read a 10 character word vetB. Print all indexes where vetA and vetB have equal characters. Consider uppercase and lowercase as equivalents*. It can be seen the student made extensive use of character-by-character comparisons, ideally where a loop would be welcome. Also, the variable representing *vetA* was declared and not used, indicating failure in the main objective

of the exercise (vectors comparison). Thus, although the student employed a large amount of programming resources, the overall assessment average did not match the skills identified in the reference solution.

student source

```
#include<stdio.h>
#include<string.h>

int main(){
      char w[11];
      int i1, cont1=0;
      char veta_1[11]={'C', 'O', 'M', 'P', 'U', 'T', 'A', 'C', 'A', 'O', '\0'};

      for(i1=0;i1<11;i1++){
         scanf("%c", &w[i1]);
      }
      if (w[0]=='C'){
         printf("0 ");
      } if (w[0]=='c'){
         printf("0 ");
      }if (w[1]=='O'){
         printf("1 ");
      }if (w[1]=='o'){
         printf("1 ");
      }if (w[2]=='M'){
         printf("2 ");
      }if (w[2]=='m'){
         printf("2 ");
      }if (w[3]=='P'){
         printf("3 ");
      }if (w[3]=='p'){
         printf("3 ");
      }if (w[4]=='U'){
         printf("4 ");
      }
      (continue)
```

```
      if (w[4]=='u'){
         printf("4 ");
      }if (w[5]=='T'){
         printf("5 ");
      }if (w[5]=='t'){
         printf("5 ");
      }if (w[6]=='A'){
         printf("6 ");
      }if (w[6]=='a'){
         printf("6 ");
      }if (w[7]=='C'){
         printf("7 ");
      }if (w[7]=='c'){
         printf("7 ");
      }if (w[8]=='A'){
         printf("8 ");
      }if (w[8]=='a'){
         printf("8 ");
      }if (w[9]=='O'){
         printf("9");
      }if (w[9]=='o'){
         printf("9 ");
      }
      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	38.00	30.00	78.95
output	20.00	0.00	0.00
input	20.00	0.00	0.00
input vs output	20.00	0.00	0.00
types of literals	50.00	50.00	100.00
variables	50.00	50.00	100.00
constants	16.67	16.67	100.00
variables vs constants	33.25	33.25	100.00
arithmetic expressions	16.67	0.00	0.00
relational expressions	33.33	33.33	100.00
boolean expressions	33.33	0.00	0.00
compound expressions	28.00	11.33	40.48
control structures	35.49	23.12	65.13
conditional structures	33.00	16.50	50.00
simple and compound conditionals	66.67	33.33	50.00
simple and compound vs multiple selection	33.00	16.50	50.00
repetition structures	45.75	41.63	90.98
infinite loops	100.00	100.00	100.00
counted loops	50.00	50.00	100.00
conditional loops vs counted	25.00	25.00	100.00
arrays	33.33	16.67	50.00

Skill Based Assessment (Average): 64.15%

Figure 4.19: Source code skill-based assessment inspection: forced solution sample.

4.7.4 Discussion

The present experiment demonstrated how source code skills-based evaluation can be useful for detecting functionally correct but conceptually incorrect solutions. The conceptual evaluation aimed to identify situations where subterfuges were used by students to achieve functional correctness, deviating from the practices adopted in reference solutions. A dataset containing 4434 source codes was collected from the application of 84 exercises in computer programming courses. Reference solutions were developed and served as the basis for building desired skill sets for each exercise.

Comparing skills identified in students' source codes with desired skills proved to be a useful resource for finding solutions with potential uses of subterfuge. Through the analysis of results and sample presentation, it has been shown that solutions whose conceptual assessment differs greatly from the reference solution are potential targets for manual inspection and search for subterfuge. Also, situations were presented demonstrating that both the lack of desired skills and excessive valuations are potential candidates for finding misconceptions. The possibility of evaluating programming exercises based on conceptual constraints, represented as high level programming skills, is a promising alternative to the traditional approaches found in the literature, allowing the identification of situations where pedagogical invention may be needed.

Using automatic strategies as a source of skills valuation is linked to an environment of uncertainty that, although represented and dealt with by the Bayesian Network, does not guarantee perfect assessments for all cases. Manual inspection of source codes identified as

potential uses of subterfuges showed that, in some cases, the evidence identification strategy failed, thus causing a disparity between the student solution and the reference solution. These problems were categorized as implementation limitations as they were caused by unexpected student behaviors where parser-based strategies were not executed correctly (limitations already pointed out in Section 4.2 experiment). Thus, it is emphasized that the implementation of some automatic strategies requires maturation in order to improve the accuracy of the method.

The mapping of desired skills based on reference solutions aimed to present an example of a formal specification of the conceptual requirements of each exercise. However, it cannot be assumed that a single reference sample is admitted as the only correct example of a given activity, as in computer programming different trains of thought may allow the elaboration of distinct and equally correct solutions both functionally and conceptually. Thus, for experimental purposes, comparison with a single reference sample by exercise was useful for method's demonstration, however, it requires expansion when applied in real evaluations. Also, constructing desired skill sets for each activity can be treated as a personal matter of the teacher, giving the freedom to add conceptual constraints to the activities and evaluate specific skills in specific activities according to the course's needs.

4.8 CHAPTER DISCUSSION

Experiments were conduced to understand the capabilities and limitations of automatic evaluation in the computers programming context. Although the literature presents several aspects and strategies for automatic evaluation of source code, different research, applications, and scenarios may present different challenges. Our first experiment (pilot test) consisted of a preliminary study that evaluated, in a limited context, the possibilities of identifying learning evidences automatically. Applying static analysis suggested automatic strategies can be feasible.

The second experiment showed automating learning evidences search in real-world source codes is a feasible possibility. The evaluated scenario has shown syntactic diversity found in real source codes impacts the identification of evidence and can lead to strategy failure in specific situations. Applying different automatic evaluation strategies was necessary to improve the effectiveness of the method.

Our third experiment (pilot test) then extended automatic source code analysis to a larger set of aspects with strategies capable of handling input and output commands, different argument combinations, and data types. Automatic strategies presented good success rates when compared to human evaluation. The fourth experiment replicated previous experiment analyzes on source codes extracted from a real STI. Automatic strategies presented success rates similar to the third experiment, however, it was identified that small syntactic changes cause strategies failures based on pure text comparisons, such as regular expressions. Strategies based on structural similarity have been identified as alternatives to avoid this vulnerability. Despite flaws in some situations, none of the strategies have been discarded, but we pointed out their application may require complex implementation in order to cover all possibilities caused by syntactic nuances.

Considering our fifth experiment, automatically identified evidences were applied as a data source for feeding the student model and monitoring the progress of student skills. Results were encouraging, detecting evidences in different source codes allows the comparison between different states of the student model, showing whether or not there is progress in the valuation of skills according to the selected source codes. Using a Dynamic Bayesian Network as a student model proved to be efficient and able to represent the mentioned state changes.

The sixth experiment specifically focused on analyzing student progress. A skill set based on topics commonly covered in real programming courses has been established, thus

representing a more realistic environment when compared to the previous experiment. We found out that student progress between exercise lists can be monitored and it was possible to identify when each skill began to be manifested. Also, automatic evaluation results were consistent with reality, indicating an absence of learning evidence in cases where students did not submit a specific list of exercises.

Lastly, the seventh experiment demonstrated 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 solutions where manual assessment is required. Method's demonstration was performed based on reference solutions, however, specifying desired skills-sets can be an interesting resource for teachers, allowing them to evaluate specific topics without having to link the programming activity to the topic being evaluated (any generic source code is candidate to evaluation on any skill-set).

Our experiments demonstrated automatic strategies can succeed in identifying learning evidences for our selected skill-set. Our skills-set definition (Section 3.1) mentions three other sets: (1) nineteen overlying high-level skills started by (Pimentel and Direne, 1998) and complemented by (Maschio, 2013); (2) forty one skill categories detailed by (Maschio, 2013); and (3) ten programming topics identified in syllabus analysis (Section 3.1.2). How much of these skill-sets our strategies can identify is discussed bellow.

Overlying high-level set describes mostly skills strongly linked to students' logical reasoning and perception. Some skills, such as *mental simulation*, *mental mapping of program structures*, *problem analysis*, and *self-knowledge* are not directly linked to program writing and, consequently, not explicitly represented on students' final source code, thus challenging/unfeasible to be detected through our implemented automatic strategies. Similarly, skills such as *error catalog* and *solution catalog* are related to students' knowledge acquired and memory, concerning on developing solutions and correcting error through replying previously experienced situations. Finally, automatic evaluation of some skills may be feasible considering aspects identified in Chapter 2 systematic literature review, however were not covered in our scope, e.g., *semantic precision* relates to the *semantic errors* aspect, *reuse of known solutions* refer to the *code reuse* aspect, and *solution optimization* relates to *complexity* and *efficiency* aspects. Although overlying high-level skill-set being based on characteristics mostly not covered by our strategies, Table 4.5 shows three skills automatically identifiable by our implemented strategies[4].

Regarding the 41 skills described by (Maschio, 2013), 34 skills from the original set were classified as automatically identifiable and valuated by with our strategies (details exposed in Section 3.2). Seven skills not being automatically identified with our strategies: *algorithm*, *analysis*, *value changes*, *loss of value*, *pipelining*, *counters and accumulators*, and *pipelining (2)*. As in (Pimentel and Direne, 1998) skills-set, most of the non-identified skills relates to aspects dependent on students' interpretation and thinking, not always reflected on final source code, and sometimes making difficult to distinguish mistakes from intentional acts. However, results were considered successful as strategies worked for most of skills.

From common programming topics identified in syllabus analysis, 9 of the 10 topics were considered automatically identifiable (Table 4.6) and were employed in our sixth experiment. *Introduction to programming* topic regards to generic programming fundamentals, related to *algorithm* skill. We pointed this topic can be interpreted as the main objective achieved by mastering the other nine, however, in our experiments no automatic strategy was directly associated.

[4]Implementation details regarding our skills can be seen in Appendix C.

Table 4.5: Comparison between overlying high-level skills (Pimentel and Direne, 1998; Maschio, 2013) and our automatically identified skills.

Overlying High-Level Skill	**Our Related Skill(s)**
syntactic precision	structuring and composition
semantic precision	
identification of main structures in the source program (keyword search)	
mental simulation of computer states during execution	
error catalog	
mental mapping of program structures	
precondition check	
problem analysis	
integration of sub-problems	
solution generalization	
reuse of known solutions	
solution catalog	
resolution speed	
readability of written code	
solution optimization	
debugging capability	
definition of basic test cases	structuring and composition
building proper interface dialog	input, output, input vs output
self-knowledge about metacognitive skills	

Table 4.6: Comparison between common programming topics and our automatically identified skills.

Syllabus Topic	**Our Related Skill(s)**
conditional structures	control structures conditional structures multiple selection conditional simple and compound conditionals
repetition structures	repetition structures infinite loops counted loops conditional loops pre-evaluated post evaluated
data types	types compatibility types of literals
variables	variables
input and output	input output
operators and expressions	arithmetic expressions relational expressions boolean expressions compound expressions
arrays	arrays
functions	functions
introduction to programming	
matrices	matrices

Thus, concluding skills-sets discussion, overlying high-level from (Pimentel and Direne, 1998; Maschio, 2013) and 41 skills-set from (Maschio, 2013) have skills unidentifiable with our strategies. The main reason is attributed to the skills nature, which refers to students' thinking and perception, requiring evaluations that can go beyond the characteristics expressed in source codes. Also, representing generic skills such as *introduction to programming* in skills-sets subject to

automatic evaluation should be avoided when source code based strategies are employed. Ideally, generic skills should be decomposed in small units and associated with evidences identifiable through such strategies.

Automatic identification of learning evidences in computers programming is still a challenging task, but results suggest the presented method, and the demonstrated use possibilities, are interesting starting points for the elaboration of new automatic source code evaluation methodologies. Improvement of automated strategies for robustness and failure prevention can allow the creation of accurate and reliable automatic assessment tools for use in real courses, facilitating the evaluation of programming activities and teaching with large numbers of students. Also, benefits of automated assessment can extend to student's context, favoring autonomy in the development of activities and optimizing learning time as a rich and accurate feedback can be generated instantly.

5 CONCLUSION

This thesis presented challenges involving automatic evaluation of source codes and monitoring of the students' acquisition of programming skills. From a systematic literature review, the most common automatic evaluation aspects and techniques were exposed, where a predominance of methodologies based on technical aspects was identified.

Although existing literature reviews already point directions for source code automatic evaluation, discovering and classifying aspects and strategies showed to be a challenging task. Different authors tend to describe and structure their works in different ways, sometimes hard to make comparisons and provide accurate classifications, e.g., standardizing automatically evaluated aspects nomenclature was challenging since different papers focuses on different contexts, technologies, and programming languages, thus it was not rare to identify multiple authors dealing with the same problem but using different names. Nomenclature disparities were even worst when considering strategies, authors tend to name their methods, thus strategies employed are not explicit and identifying which strategy category better describes authors' methods was a challenge and required deep investigations. Some strategies however showed easy to identify, such as *test cases*, *regular expressions*, and *reflection* due to be techniques whose nomenclatures are already standardized for multiple technologies and programming languages.

Our systematic literature review is considered a strong point of the current thesis. Research protocol exposed can be applied to extend and update the state of art on techniques to automatically measure the knowledge of computer programming students. Categorizations defined for aspects and strategies can also be reviewed and updated in future studies. Distinct categorizations for aspects and strategies are also pointed out as interesting as this allow to differentiate *what is evaluated* from *how it is done*. Although our review identified several aspects and strategies, the correlation between these groups was not investigated. We point this correlation investigation as future work to identify which strategies are more commonly applied to evaluate each aspect.

In contrast to literature predominant methodologies, using automatic strategies for source code evaluation focusing on identification of high-level programming skills, rather than just technical aspects, was investigated. Taking advantage of previous works, along with the knowledge acquired through the systematic literature review and the analysis of real-world programming courses, a skill-set has been defined as a candidate for a conceptual point of view based automatic evaluation method.

Previous research already described skills-sets employed (or desired) in programming students' evaluations. We discovered some skills, such as listed by (Pimentel and Direne, 1998), are strongly dependent on students' reasoning and logical thinking. We identified such features are not explicit in students' final source code, causing automatic evaluation to be challenging/unfeasible through our strategies. We believe collecting information besides source code can make it possible to automatically detect evidences for such skills.

Considering our automatically identifiable skills-set, source code evaluation strategies found in the literature were used as a starting point for elaborating our automatic learning evidences identification method: A-Learn EvId. Our systematic literature review revealed 25 categories of strategies covering static, dynamic and hybrid approaches. Both specific and generic strategies were found. We identified specific strategies tend to provide detailed evaluations about particular topics but, as a consequence, are dependent on specific scenarios and can impose application restrictions. Generic strategies, in contrast, provide less details but can be applied

in wider scenarios. Nine strategies, mixing generic and specific, were selected to compose our method. Combining strategies in hybrid systems was proven to be effective since multiple aspects, specific or generic, can be evaluated and provide learning evidences for the same skill, increasing chances of correct assessment. Also, combining static and dynamic strategies permitted evaluating source codes from two complementary perspectives: internal code units inspection, and runtime program behavior analysis. As a result, our method identifies 37 skills automatically; in comparison, the analyzed literature (Chapter 2) points out a maximum of 10 aspects evaluated in a single study (Rajala et al., 2016).

Automatically evaluating programming aspects is a task surrounded by uncertainty. We discovered real student-made source codes can reveal unexpected situations that causes strategies failures. Students' creativity is unpredictable, they can follow teachers' directions and write source codes similar to desired, try by themselves until achieve a valid solution, or even go through strange ways mixing methodologies (e.g., by studying from several divergent sources such as books and websites). In all mentioned cases correct source codes can be achieved considering both conceptual and functional viewpoints, however, strategies may not be prepared to evaluate them properly, e.g., when students import an incompatible library found on Internet, or apply a syntax different than exposed by teachers. Strange behaviors such as mentioned could be found in our real-world datasets and negatively impacted our strategies, causing some learning evidences to not be correctly evaluated.

Strategies implemented to demonstrate our method showed implementation-related limitations, the following options are listed as starting points to improve its reliability and accuracy: (1) identify source codes (or code fragments) where strategies fail and extend implementation to cover such situations; (2) replace problematic strategies by failsafe ones, e.g., re-implement evidences that use *regular expressions* by applying structural similarity based strategies such as *parser*; (3) limit students' resources usage, e.g., by forcing them to write source codes directly on ITS interface, inhibiting unexpected resources/syntax usage. We believe limiting students' resources, although acceptable for learning environments, must be analyzed carefully to avoid disparities with industry-standard programming environments, forcing students' to re-learn how to program considering full language/technology features.

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 specified and collected). Results showed our method is promising, being able to automatically evaluate students' source codes, identifying multiple programming skills. Automatic strategies results were represented through our learner model. 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, resuming our first research question (*TRQ1*, from Section 1.1), experiments' results suggest that high-order cognitive skills can be automatically evaluated in the computer programming context. However, efforts and research expansion still needed to contour implementation-related limitations.

Strategies results, when applied as a feed source for the learner model proved to be an interesting alternative besides the literature common technical aspect evaluation. Applying automatically identified learning evidences as source for feeding the learner model can be considered a strong point of our research. Evaluating multiple aspects from a single source code provides information regarding the programming resources employed by the student during program's writing. Visualizing this information on the learner model gives an instant overview of skills manifested in a particular source code. By comparing multiple states of the learner model it is possible to detect which skills-sets are manifested in different source code sets. Also,

comparison permits to visualize whether students' progressed or not along the course and allow identifying programming topics needing special teaching attention. Thus, resuming our second research question (*TRQ2*, from Section 1.1), automatically identified high-order cognitive skills were applied for monitoring students' progress and experiments' results suggest feasibility.

Identifying multiple skills from source codes enabled skills-based assessment, allowing to evaluate specific topics on each programming assignment. Our skills-based assessment method has shown promising in addressing automatic evaluation of programming activities by providing conceptual feedback to teachers. Specifying conceptual constraints in programming activities provided resources for locating solutions built using subterfuges, which indicate potential concept-related failures and topics where pedagogical interventions can be applied.

Our skills-based assessment method is highlighted as a differential compared to the approaches found in our systematic literature review. Automated technical assessment is still a valuable resource, sufficient and effective in some scenarios, however, skills-based approach can be seen as a viable alternative, acting isolated or even complementary. Automating source codes evaluation on both technical or skills-based approaches is challenging, limitations and unexpected situations affects strategies regardless methodology adopted. We believe research expansion regarding automatic evaluation, and specifically in skills-based methods, can bring benefits and enrich the functionalities of current ITS, providing alternative ways of assessing students' skills and, consequently, making it easier to monitor individual progress in large classes.

Resuming the A-Learn EvId method, three steps were defined: (1) input source code insertion; (2) learning evidences identification through automatic strategies; and (3) apply strategies results as feeding source to a learner model. We detailed the method implementation by focusing on C-Language source codes, strategies based on static, dynamic and hybrid approaches, and a Dynamic Bayesian Network learner model. However, we point out the method can be generalized to other scenarios, acting on different programming languages, paradigms, and technologies. New skills-sets need to be defined to better represent the desired domain, different strategies can be employed, and the learner model must be updated accordingly.

Our implementation was presented as a working prototype. Known implementation limitations were exposed and discussed, thus, the practical viability of our propose is discussed in two perspectives: immediate and long term. Considering the immediate perspective, conclusions are based on our implementation results. Our method is viable for scientific experimentation, can be applied to build teaching support tools and to evaluate specific programming topics in C-Language source codes. However, real-classroom application should apply it as a complementary evaluation methodology. Strategies implementation-related limitations still unsolved and affect evaluation results, thus students' automatic grading is not encouraged yet. Concerning the long term perspective, automatic strategies improvement will increase reliability, enabling the construction of fully automated evaluation environments, potentially applicable to students' grading and contributing to teachers' workload reduction. Also, applying the method on programming teaching STI can allow automatic feedback improvement, skills visualization, and help in identifying topics where students have difficulty to progress. Identifying such topics can be useful for automatic suggesting programming exercises, study materials and contributing to students autonomy.

Regarding the research niche investigated, the present study resulted in updating and expanding the state of the art through systematic literature review. 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 evaluation methods, especially when employing the resulting information to monitor the progress of student 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.

5.1 FUTURE WORK

This section presents additional notes for future work.

The student model presented in Section 3.3, based on a Bayesian Network, has nodes (skills) influenced by multiple evidence. As mentioned, all evidence sets were set up with equal weights, however, refinement of these weights may be beneficial to the model, allowing less significant evidence to cause less impact on the network while the most important strategies may be responsible for a greater portion of the the nodes valuation. The skills graph (Appendix A) described by (Maschio, 2013) presents different connections between nodes, where relations such as prerequisites, generalizations, and analogies can be found. Investigating those relations can be a starting point for specifying the Bayesian Network weights.

The analysis of programming languages presented in Section 2.3 showed that there are emerging technologies, especially Java with increasing popularity. Thus, the possibility of transcribing automatic strategies to support emerging technologies is also highlighted, expanding the possibilities of application of the method and keeping the work updated against the technologies currently applied in the real world. Also, revising and updating the student model to make it compatible with object-oriented paradigm skills also becomes an interesting possibility.

Our sixth experiment compared students' progress between ten exercise lists. The absence of progress can be investigated to evaluate the effectiveness of exercise lists and to detect topics where students have difficulty progressing. Also, the lack of progress can be exploited as an opportunity for error correction and mediation (automatic or by suggesting pedagogical interventions to teachers). Also, dealing with progress monitoring, our model considered the maximum value metric from (Koh et al., 2014) study. Thus, once a single source code max valuates a given skill, this value will override evaluations from all other selected source codes in the timeline. Investigating complementary metrics can be an interesting topic to enrich knowledge detection, and visualization such as: harder exercises can have more influence; frequently identified evidences can have greater influence in skills valuation; rarely manifested skills can pointed lack in skills (*does a student really understood concepts or just employed a given programming resource "by accident"?*).

Considering the skill-based assessment experiment presented in Section 4.7, along with the argument that using a single reference solution for each exercise may not be optimal, the need to implement an exercise authoring tool focused on this type of assessment is highlighted. Provide the teacher with resources to design exercises and define desired skill sets without requiring the student solution to be similar to a predefined model will benefit the assessment process. Also, it can be considered that a single activity can be assessed from different perspectives (focusing on different skills), providing evidence of student learning and indications to pedagogical interventions. Finally, another possibility for specifying desired skills can be implemented by providing resources for teachers to feed the reference solutions set by accepting students' solutions (e.g., an unknown solution is submitted and classified as a potential subterfuge, teacher manual inspection can eventually detect correctness and add it to the reference solutions set).

The A-Learn EvId method was applied to the computers programming domain. However, possibilities of generalization to other domains are pointed out. Considering method's overview described in Chapter 3, three steps are highlighted: (1) defining input data; (2) defining automatic strategies based on a skill-set relevant for the domain; and (3) defining a learner model, also

based on the chosen skill-set. Theoretically, those steps can be implemented for any domain of which automatic strategies are feasible. As an example, the method may be feasible for evaluating English grammar domain: input data can be defined as student written texts; skill-set can represent grammar topics, such as verbs, prepositions, gender, and passive voice. Automatic strategies can be defined based on text mining techniques; and grammar topics can be represented in the learner model. Thus, future research can investigate if the method can also benefit other areas of knowledge.

REFERENCES

Aaltonen, K., Ihantola, P., and Seppälä, O. (2010). Mutation analysis vs. code coverage in automated assessment of students' testing skills. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 153–160, New York, NY, USA. ACM.

Ahmed, U. Z., Kumar, P., Karkare, A., Kar, P., and 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*, ICSE-SEET '18, pages 78–87, New York, NY, USA. ACM.

Akahane, Y., Kitaya, H., and Inoue, U. (2015). Design and evaluation of automated scoring java programming assignments. In *2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 1–6.

Ala-Mutka, K., Uimonen, T., and Jarvinen, H.-M. (2004). Supporting students in c++ programming courses with automatic program style assessment. *Journal of Information Technology Education: Research*, 3(1):245–262.

Ala-Mutka, K. M. (2005). A survey of automated assessment approaches for programming assignments. *Computer science education*, 15(2):83–102.

Amelung, M., Forbrig, P., and Rösner, D. (2008). Towards generic and flexible web services for e-assessment. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '08, pages 219–224, New York, NY, USA. ACM.

Bahlke, R. and Snelting, G. (1986). The psg system: From formal language definitions to interactive programming environments. *ACM Trans. Program. Lang. Syst.*, 8(4):547–576.

Banerjee, S., Ramanathan, C., and Rao, N. J. (2015). An approach to automatic evaluation of higher cognitive levels assessment items. In *2015 IEEE 3rd International Conference on MOOCs, Innovation and Technology in Education (MITE)*, pages 342–347.

Beh, M., Gottipati, S., Lo, D., and Shankararaman, V. (2016). Semi-automated tool for providing effective feedback on programming assignments. In *ICCE 2016 - 24th International Conference on Computers in Education: Think Global Act Local - Main Conference Proceedings*, pages 258–263. cited By 0.

Boe, B., Hill, C., Len, M., Dreschler, G., Conrad, P., and Franklin, D. (2013). Hairball: Lint-inspired static analysis of scratch projects. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 215–220, New York, NY, USA. ACM.

Braunfeld, P. G. and Fosdick, L. D. (1962). The use of an automatic computer system in teachin. *IRE Transactions on Education*, E-5(3 & 4):156–167.

Brusilovsky, P. and Sosnovsky, S. (2005). Individualized exercises for self-assessment of programming knowledge: An evaluation of quizpack. *J. Educ. Resour. Comput.*, 5(3).

Cardell-Oliver, R. (2013). Evaluating the application and understanding of elementary programming patterns. In *2013 22nd Australian Software Engineering Conference*, pages 60–67.

Choi, S.-E. and Lewis, E. C. (2000). A study of common pitfalls in simple multi-threaded programs. In *Proceedings of the Thirty-first SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '00, pages 325–329, New York, NY, USA. ACM.

Conejo, R., Barros, B., and Bertoa, M. F. (2018). Automated assessment of complex programming tasks using siette. *IEEE Transactions on Learning Technologies*, pages 1–1.

Cox, R. and Brna, P. (1995). Supporting the use of external representations in problem solving: The need for flexible learning environments. *Journal of Artificial intelligence in Education*, 6:239–302.

Cui, B., Li, J., Guo, T., Wang, J., and Ma, D. (2010). Code comparison system based on abstract syntax tree. In *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*, pages 668–673.

De-La-Fuente-Valentín, L., Pardo, A., and Delgado Kloos, C. (2013). Addressing drop-out and sustained effort issues with large practical groups using an automated delivery and assessment system. *Computers and Education*, 61(1):33–42. cited By 15.

de Souza, D. M., Oliveira, B. H., Maldonado, J. C., Souza, S. R. S., and Barbosa, E. F. (2014). Towards the use of an automatic assessment system in the teaching of software testing. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, pages 1–8.

Delgado, K. and De Barros, L. (2006). Diagnostic of programs for programming learning tools. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4140 LNAI:7–16. cited By 1.

Drasutis, S., Motekaityte, V., and Noreika, A. (2010). A method for automated program code testing [automatizuoto programos kodo testavimo metodas]. *Informatics in Education*, 9(2):199–208. cited By 1.

Edmison, B. and 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*, SIGCSE '19, pages 260–266, New York, NY, USA. ACM.

Edwards, S. H. (2003). Rethinking computer science education from a test-first perspective. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 148–155, New York, NY, USA. ACM.

Etzkorn, L. H., Davis, C. G., Bowen, L. L., Etzkorn, D. B., Lewis, L. W., Vinz, B. L., and Wolf, J. C. (1996). A knowledge-based approach to object-oriented legacy code reuse. In *Proceedings of ICECCS '96: 2nd IEEE International Conference on Engineering of Complex Computer Systems (held jointly with 6th CSESAW and 4th IEEE RTAW)*, pages 493–496.

Farrow, M. and King, P. J. B. (2008). Experiences with online programming examinations. *IEEE Transactions on Education*, 51(2):251–255.

Fonte, D., Boas, I., Oliveira, N., Da Cruz, D., Gançarski, A., and Henriques, P. (2014). Partial correctness and continuous integration in computer supported education. In *CSEDU 2014 - Proceedings of the 6th International Conference on Computer Supported Education*, volume 2, pages 205–212. cited By 2.

Fonte, D., Da Cruz, D., Gançarski, A., and Henriques, P. (2013). A flexible dynamic system for automatic grading of programming exercises. In *OpenAccess Series in Informatics*, volume 29, pages 129–144. cited By 10.

Förster, E., Förster, K., and Löwe, T. (2018). Teaching programming skills in primary school mathematics classes: An evaluation using game programming. In *2018 IEEE Global Engineering Education Conference (EDUCON)*, pages 1504–1513.

Gerdes, A., Jeuring, J. T., and Heeren, B. J. (2010). Using strategies for assessment of programming exercises. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 441–445, New York, NY, USA. ACM.

Gerdt, P. and Sajaniemi, J. (2006). A web-based service for the automatic detection of roles of variables. *SIGCSE Bull.*, 38(3):178–182.

Haldeman, G., Tjang, A., Babeş-Vroman, M., Bartos, S., Shah, J., Yucht, D., and Nguyen, T. D. (2018). Providing meaningful feedback for autograding of programming assignments. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 278–283, New York, NY, USA. ACM.

Hamouda, S., Edwards, S. H., Elmongui, H. G., Ernst, J. V., and Shaffer, C. A. (2018). Recurtutor: An interactive tutorial for learning recursion. *ACM Trans. Comput. Educ.*, 19(1):1:1–1:25.

Head, A., Glassman, E., Soares, G., Suzuki, R., Figueredo, L., D'Antoni, L., and Hartmann, B. (2017). Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, L@S '17, pages 89–98, New York, NY, USA. ACM.

Helmick, M. T. (2007). Integrated online courseware for computer science courses. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '07, pages 146–150, New York, NY, USA. ACM.

Helminen, J. and Malmi, L. (2010). Jype - a program visualization and programming exercise tool for python. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 153–162, New York, NY, USA. ACM.

Herout, P. and Brada, P. (2015). Duck testing enhancements for automated validation of student programmes: How to automatically test the quality of implementation of students' programmes. In *CSEDU 2015 - 7th International Conference on Computer Supported Education, Proceedings*, volume 1, pages 228–234. cited By 2.

Hettiarachchi, E., Huertas, M., and Mor, E. (2013). Skill and knowledge e-assessment: A review of the state of the art. *IN3 Working Paper Series*.

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

Hung, S.-L., Kwok, I.-F., and Chan, R. (1993). Automatic programming assessment. *Computers and Education*, 20(2):183–190. cited By 24.

Ihantola, P., Ahoniemi, T., Karavirta, V., and 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*, Koli Calling '10, pages 86–93, New York, NY, USA. ACM.

Insa, D. and Silva, J. (2018). Automatic assessment of java code. *Computer Languages, Systems and Structures*, 53:59–72. cited By 1.

Jackson, D. and Usher, M. (1997). Grading student programs using assyst. In *Proceedings of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '97, pages 335–339, New York, NY, USA. ACM.

Jadud, M. C. and Dorn, B. (2015). Aggregate compilation behavior: Findings and implications from 27,698 users. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, pages 131–139, New York, NY, USA. ACM.

Jamil, H. M. (2017). Automated personalized assessment of computational thinking MOOC assignments. In *2017 IEEE 17th International Conference on Advanced Learning Technologies (ICALT)*, pages 261–263.

Jelemenska, K., Cicak, P., and Gazik, M. (2016). VHDL models e-assessment in Moodle environment. In *2016 International Conference on Emerging eLearning Technologies and Applications (ICETA)*, pages 141–146.

Kamada, H., Nishikawa, K., and Okui, Y. (2016). The visual interactive programing learning system using image processing. In *2016 Third International Conference on Computing Measurement Control and Sensor Network (CMCSN)*, pages 158–161.

Kiesmueller, U., Sossalla, S., Brinda, T., and Riedhammer, K. (2010). Online identification of learner problem solving strategies using pattern recognition methods. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, pages 274–278, New York, NY, USA. ACM.

Kim, D., Kwon, Y., Liu, P., Kim, I. L., Perry, D. M., Zhang, X., and Rodriguez-Rivera, G. (2016). Apex: Automatic programming assignment error explanation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 311–327, New York, NY, USA. ACM.

Koh, K. H., Basawapatna, A., Bennett, V., and Repenning, A. (2010). Towards the automatic recognition of computational thinking for adaptive visual language learning. In *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 59–66.

Koh, K. H., Nickerson, H., Basawapatna, A., and Repenning, A. (2014). Early validation of computational thinking pattern analysis. In *Proceedings of the 2014 Conference on Innovation &; Technology in Computer Science Education*, ITiCSE '14, pages 213–218, New York, NY, USA. ACM.

Kohn, T. (2019). The error behind the message: Finding the cause of error messages in python. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 524–530, New York, NY, USA. ACM.

Kumar, A. N. (2005). Generation of problems, answers, grade, and feedback—case study of a fully automated tutor. *J. Educ. Resour. Comput.*, 5(3).

Kutzke, A. R. and Direne, A. I. (2015). *FARMA-ALG: An Application for Error Mediation in Computer Programming Skill Acquisition*, pages 690–693. Springer International Publishing, Cham.

Le Ru, Y., Aron, M., Gerval, J.-P., and Napoleon, T. (2015). Tests generation oriented web-based automatic assessment of programming assignments. *Smart Innovation, Systems and Technologies*, 41:117–127. cited By 1.

Lee, J., Song, D., So, S., and Oh, H. (2018). Automatic diagnosis and correction of logical errors for functional programming assignments. *Proc. ACM Program. Lang.*, 2(OOPSLA):158:1–158:30.

Liang, Y., Liu, Q., Xu, J., and Wang, D. (2009). The recent development of automated programming assessment. In *2009 International Conference on Computational Intelligence and Software Engineering*, pages 1–5.

Maguire, P., Maguire, R., and Kelly, R. (2017). Using automatic machine assessment to teach computer programming. *Computer Science Education*, 27(3-4):197–214. cited By 0.

Malmi, L., Karavirta, V., Korhonen, A., and Nikander, J. (2005). Experiences on automatically assessed algorithm simulation exercises with different resubmission policies. *J. Educ. Resour. Comput.*, 5(3).

Marin, V. J., Pereira, T., Sridharan, S., and Rivero, C. R. (2017). Automated personalized feedback in introductory java programming MOOCs. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1259–1270.

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).

Maschio, E. and Direne, A. (2015). Multiple external representations to support knowledge acquisition in computer programming. *Brazilian Journal of Computers in Education*, 23(03):81.

Moreno-León, J., Román-González, M., Harteveld, C., and Robles, G. (2017). On the automatic assessment of computational thinking skills: A comparison with human experts. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '17, pages 2788–2795, New York, NY, USA. ACM.

Morris, D. S. (2003). Automatic grading of student's programming assignments: an interactive process and suite of programs. In *33rd Annual Frontiers in Education, 2003. FIE 2003.*, volume 3, pages S3F–1.

Možina, M., Lazar, T., and Bratko, I. (2018). Identifying typical approaches and errors in prolog programming with argument-based machine learning. *Expert Systems with Applications*, 112:110–124. cited By 0.

Murray, W. (1987). Automatic program debugging for intelligent tutoring systems. *Computational Intelligence*, 3(1):1–16. cited By 12.

Muñoz De La Peña, D., Gómez-Estern, F., and Dormido, S. (2012). A new internet tool for automatic evaluation in control systems and programming. *Computers and Education*, 59(2):535–550. cited By 12.

Neapolitan, R. E. (2003). *Learning Bayesian Networks*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Novais, D., Pereira, M., and Henriques, P. (2016). Profile detection through source code static analysis. In *OpenAccess Series in Informatics*, volume 51, pages 91–913. cited By 0.

Oechsle, R. and Barzen, K. (2007). Checking automatically the output of concurrent threads. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '07, pages 43–47, New York, NY, USA. ACM.

Ota, G., Morimoto, Y., and Kato, H. (2016). Ninja code village for scratch: Function samples/-function analyser and automatic assessment of computational thinking concepts. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 238–239.

Patel, A., Panchal, D., and Shah, M. (2015). Towards improving automated evaluation of java program. *Advances in Intelligent Systems and Computing*, 337:489–496. cited By 1.

Petersen, K., Feldt, R., Mujtaba, S., and Mattsson, M. (2008). Systematic mapping studies in software engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, EASE'08, pages 68–77, Swindon, UK. BCS Learning & Development Ltd.

Pimentel, A. R. and Direne, A. I. (1998). Medidas cognitivas no ensino de programaçao de computadores com sistemas tutores inteligentes. *Revista Brasileira de Informática na Educaçao (IE)*, 3:17–24.

Pinto, M. A. (2013). Web-based system for automatic evaluation of java algorithms. In *Eurocon 2013*, pages 2123–2128.

Qian, K., Sztipanovits, M., and Fu, X. (2008). Automated testing and smart tutoring system for web application. In *2008 International Workshop on Education Technology and Training 2008 International Workshop on Geoscience and Remote Sensing*, volume 2, pages 582–585.

Quan, T., Nguyen, P., Bui, T., Huynh, L., and Do, A. (2009). A framework for automatic verification of programing exercises. pages 41–45. cited By 1.

Rahman, K. A. and 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*.

Rajala, T., Kaila, E., Lindén, R., Kurvinen, E., Lokkila, E., Laakso, M.-J., and Salakoski, T. (2016). Automatically assessed electronic exams in programming courses. In *Proceedings of the Australasian Computer Science Week Multiconference*, ACSW '16, pages 11:1–11:8, New York, NY, USA. ACM.

Ramadhan, H. A. (1997). Improving the engineering of model tracing based intelligent program diagnosis. *IEE Proceedings - Software Engineering*, 144(3):149–161.

Rashid, N., Lim, L., Eng, O., Ping, T., Zainol, Z., and Majid, O. (2016). A framework of an automatic assessment system for learning programming. *Lecture Notes in Electrical Engineering*, 362:967–977. cited By 3.

Rojas, S. A. (2014). Towards automatic recognition of irregular, short-open answers in Fill-in-the-blank tests. *Tecnura*, 18:47–61.

Rosenthal, T., Suppes, P., and Ben-Zvi, N. (2002). Automated evaluation methods with attention to individual differences-a study of a computer-based course in C. In *32nd Annual Frontiers in Education*, volume 1, pages T1B–T1B.

Siegfried, R., Klinger, S., Gross, M., Sumner, R. W., Mondada, F., and Magnenat, S. (2017). Improved mobile robot programming performance through real-time program assessment. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, pages 341–346, New York, NY, USA. ACM.

Singh, R., Gulwani, S., and Solar-Lezama, A. (2013). Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 15–26, New York, NY, USA. ACM.

Skupas, B. and Dagiene, V. (2010). Observations from semi-automatic testing of program codes in the high school student maturity exam. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 31–36, New York, NY, USA. ACM.

Souza, D. M., Felizardo, K. R., and Barbosa, E. F. (2016). A systematic literature review of assessment tools for programming assignments. In *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, pages 147–156.

Spandana, J. S. N., Srividhyasaradha, K., Subasri, G., and Vasuki, P. (2018). Automatic code evaluation system. In *2018 International Conference on Computer, Communication, and Signal Processing (ICCCSP)*, pages 1–5.

Staubitz, T., Renz, J., Willems, C., Jasper, J., and Meinel, C. (2014). Lightweight ad hoc assessment of practical programming skills at scale. In *2014 IEEE Global Engineering Education Conference (EDUCON)*, pages 475–483.

Striewe, M. and Goedicke, M. (2014). A review of static analysis approaches for programming exercises. In *International Computer Assisted Assessment Conference*, pages 100–113. Springer.

Suarez, M. and Sison, R. (2008). Automatic construction of a bug library for object-oriented novice java programmer errors. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5091 LNCS:184–193. cited By 11.

Sztipanovits, M., Qian, K., and Fu, X. (2008). The automated web application testing (AWAT) system. In *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ACM-SE 46, pages 88–93, New York, NY, USA. ACM.

Tiantian, W., Xiaohong, S., Peijun, M., Yuying, W., and Kuanquan, W. (2009). Autolep: An automated learning and examination system for programming and its application in

programming course. In *2009 First International Workshop on Education Technology and Computer Science*, volume 1, pages 43–46.

Traynor, D. and Gibson, J. P. (2005). Synthesis and analysis of automatic assessment methods in cs1: Generating intelligent mcqs. *SIGCSE Bull.*, 37(1):495–499.

Truong, N., Roe, P., and Bancroft, P. (2005). Automated feedback for "fill in the gap" programming exercises. In *Proceedings of the 7th Australasian Conference on Computing Education - Volume 42*, ACE '05, pages 117–126, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

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

Turner, S. A. (2015). Looking glass: A C++ library for testing student programs through reflection. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 528–533, New York, NY, USA. ACM.

Ullah, Z., Lajis, A., Jamjoom, M., Altalhi, A., Al-Ghamdi, A., and 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.

Ureel, L. C. and Wallace, C. (2015). Webta: Automated iterative critique of student programming assignments. In *2015 IEEE Frontiers in Education Conference (FIE)*, pages 1–9.

Ureel II, L. C. and Wallace, C. (2019). Automated critique of early programming antipatterns. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 738–744, New York, NY, USA. ACM.

VanPatten, B. and Williams, J. (2015). *Theories in second language acquisition: An introduction*. Routledge, second edition.

Vesin, B., Klašnja-Milićević, A., and Ivanović, M. (2013). Improving testing abilities of a programming tutoring system. In *2013 17th International Conference on System Theory, Control and Computing (ICSTCC)*, pages 669–673.

Wilcox, T. R., Davis, A. M., and Tindall, M. H. (1976). The design and implementation of a table driven, interactive diagnostic programming system. *Commun. ACM*, 19(11):609–616.

Wu, J. (2011). Improving the writing of research papers: Imrad and beyond.

Xiaohong Su, Jing Qiu, Tiantian Wang, and Lingling Zhao (2016). Optimization and improvements of a moodle-based online learning system for C programming. In *2016 IEEE Frontiers in Education Conference (FIE)*, pages 1–8.

Yaganteeswarudu, A. (2016). The speaking compiler-A compiler with audio for immediate error correction. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 980–983.

Yamashita, K., Sugiyama, T., Kogure, S., Noguchi, Y., Konishi, T., and 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*, pages 288–295. cited By 0.

Yang, C., Chien, L., Buehrer, D. J., and Chen, C. (2009). An evaluation on interaction between APAS, TDD and Learning Style. In *2009 International Conference on New Trends in Information and Service Science*, pages 350–355.

Ying, M. and Hong, Y. (2011). The development of an online sql learning system with automatic checking mechanism. In *The 7th International Conference on Networked Computing and Advanced Information Management*, pages 346–351.

APPENDIX A – SKILLS GRAPH

Figures A.1 e A.2 presents the skills graph elaborated by (Maschio, 2013). Where:

- **P** represents Prerequisite;
- **G** represents Generalization;
- **A** represents Analogy;
- **C** represents Correction;
- **R** represents Refinement.

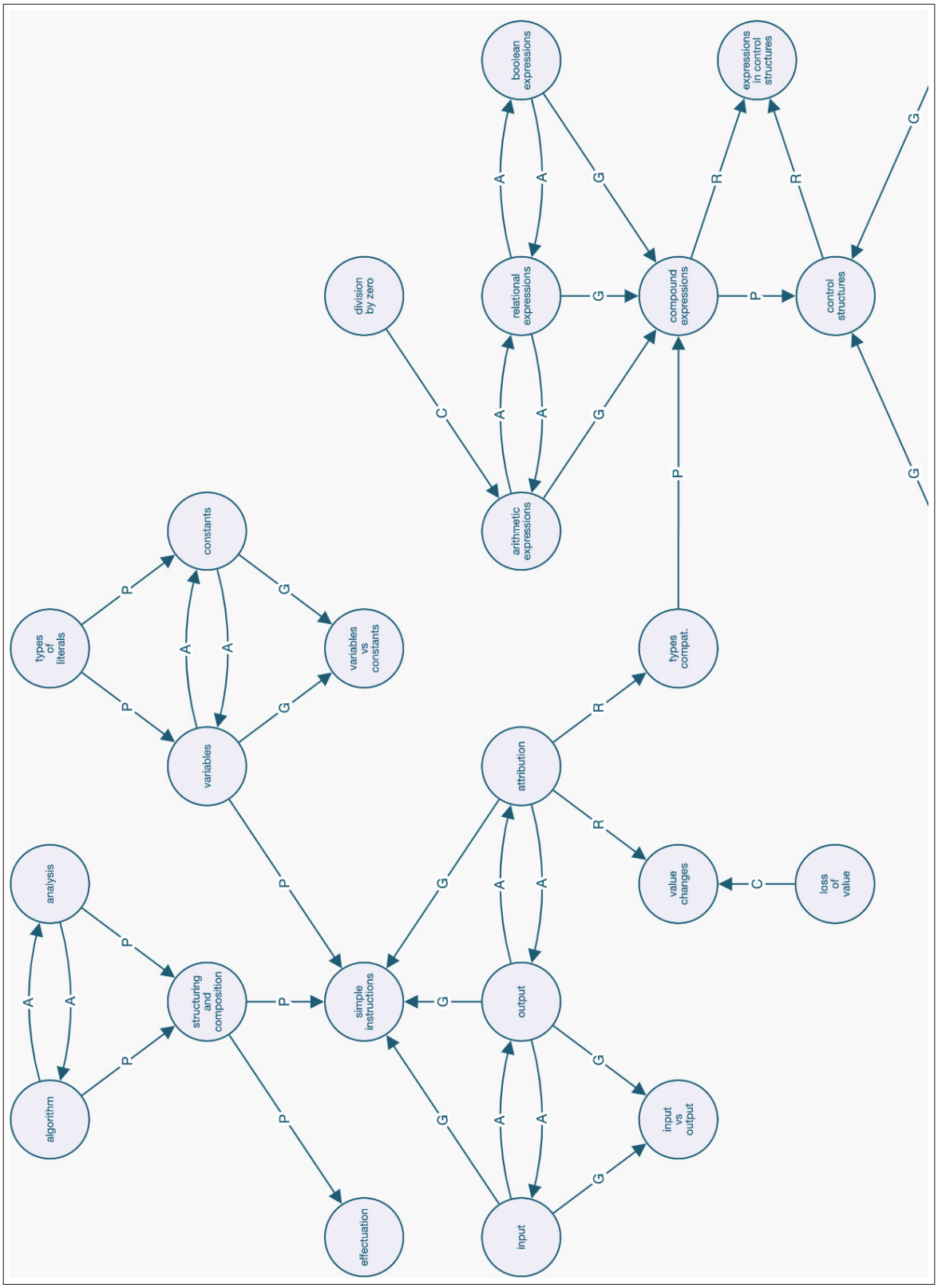


Figure A.1: Skills graph (superior segment). Translated from (Maschio, 2013).

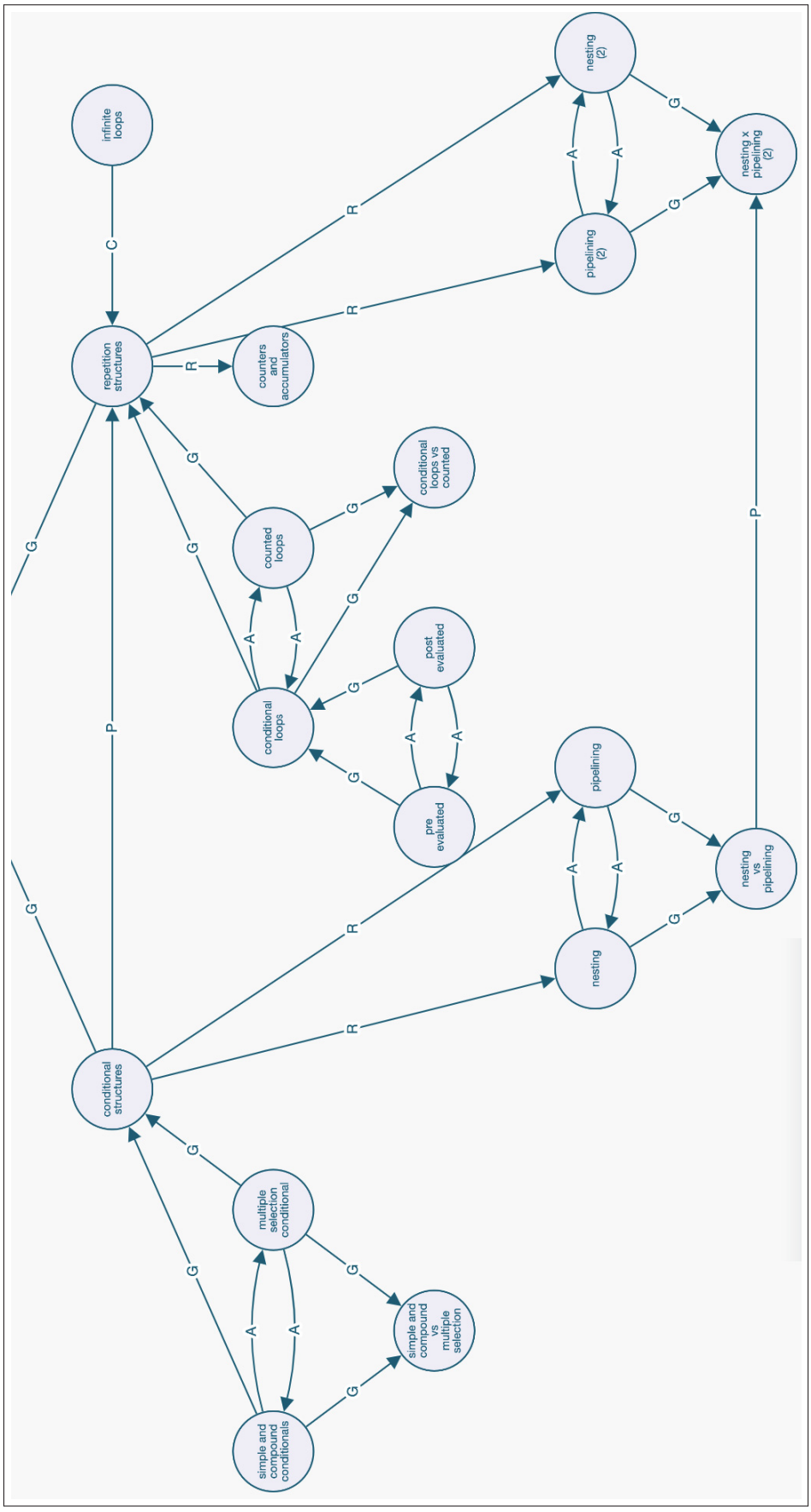


Figure A.2: Skills graph (inferior segment). Translated from (Maschio, 2013).

A.1 PROGRAMMING LANGUAGES ON SELECTED PAPERS

Table A.1 presents the data used to generate the most investigated programming languages visual representation.

Table A.1: Programming languages popularity.

Language	**1960/70**	**1970/80**	**1980/90**	**1990/00**	**2000/10**	**2010/20**
Ada	0	0	0	1	0	0
ALGOL 60	0	0	1	0	0	0
Assembly	0	0	1	0	1	0
C	0	0	1	0	8	13
C#	0	0	0	0	1	2
C++	0	0	0	1	4	7
Cobol	0	1	0	0	0	0
Fortran	0	1	0	0	0	0
Haskell	0	0	0	0	1	1
Java	0	0	0	0	14	37
Javascript	0	0	0	0	0	2
Lisp	0	0	2	1	0	0
Matlab	0	0	0	0	0	1
MODULA-2	0	0	1	0	0	0
Not specified	2	0	0	0	2	7
OCaml	0	0	0	0	0	1
OSSL	0	0	0	0	0	1
Pascal	0	0	1	1	2	1
Perl	0	0	0	0	1	0
PL/I	0	1	0	0	0	0
Visual Programming	0	0	0	0	0	9
Prolog	0	0	0	1	0	1
Python	0	0	0	0	0	7
Schema	0	0	0	0	0	1
Smalltalk	0	0	0	0	1	0
SQL	0	0	0	0	0	2
VHDL	0	0	0	0	0	2
Web	0	0	0	0	2	1

APPENDIX B – PROGRAMMING TOPICS: SYLLABUS ANALYSIS

Table B.1 shows all topics found on Section 3.1.2 syllabus analysis and their respective occurrences counter.

Table B.1: Programming Topics Syllabus Analysis

	Central-West		**Northeast**		**North**		**Southeast**		**South**		
Syllabus Topics	**UFG**	**UnB**	**UFCG**	**UFPE**	**UFAM**	**UFPA**	**UFMG**	**UFRJ**	**UFRGS**	**UFSC**	**Total**
Introduction to programming	x	x	x	x		x			x	x	7
Pseudocode		x									1
Abstraction		x									1
Data types	x	x	x		x	x	x		x	x	8
Variables	x	x			x	x	x		x	x	7
Input and output	x	x	x			x	x		x	x	7
Operators and expressions	x	x			x	x	x		x	x	7
Conditional structures	x	x	x	x	x	x	x		x	x	9
Repetition structures	x	x	x	x	x	x	x		x	x	9
Vectors	x	x	x	x	x				x	x	7
Matrices	x	x	x	x	x				x	x	7
Strings	x								x		2
Heterogeneous structures	x	x									2
Functions		x	x		x	x	x		x	x	7
Registers	x				x						2
Libraries	x										1
Testing and debug		x									1
Recursions		x			x	x			x		4
Code complexity calculation		x									1
Data structures (e.g. List, Stack)			x		x						2
Good practices, software quality			x	x							2
Object oriented programming				x						x	2
Imperative programming				x							1

APPENDIX C – IMPLEMENTED EVIDENCES

Skills and evidences located by automatic strategies and applied as input values for the Dynamic Bayesian network (learner model), with their respective strategies and valuations, are shown in figures C.1 to C.7. Green colored evidences represent strategies implemented as scripts. Orange colored evidences, in turn, represent strategies valued by inference.

Skills can have one or more evidences. Evidences can be identified by one or more automatic strategies. Each strategy has a valuation rule, which provides a percentage of success in using determined programming resources.

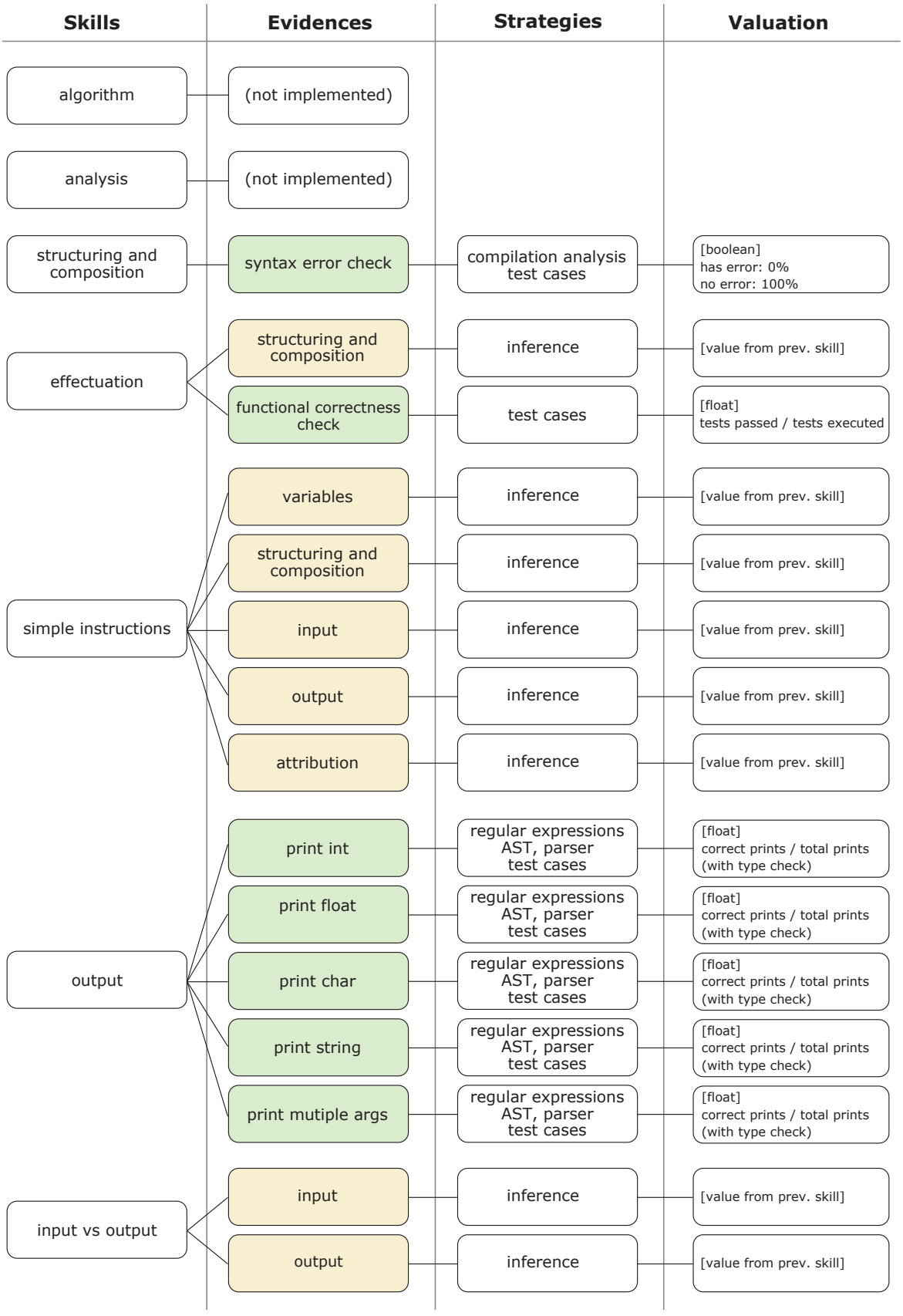


Figure C.1: Implemented evidences: part 1 of 7.

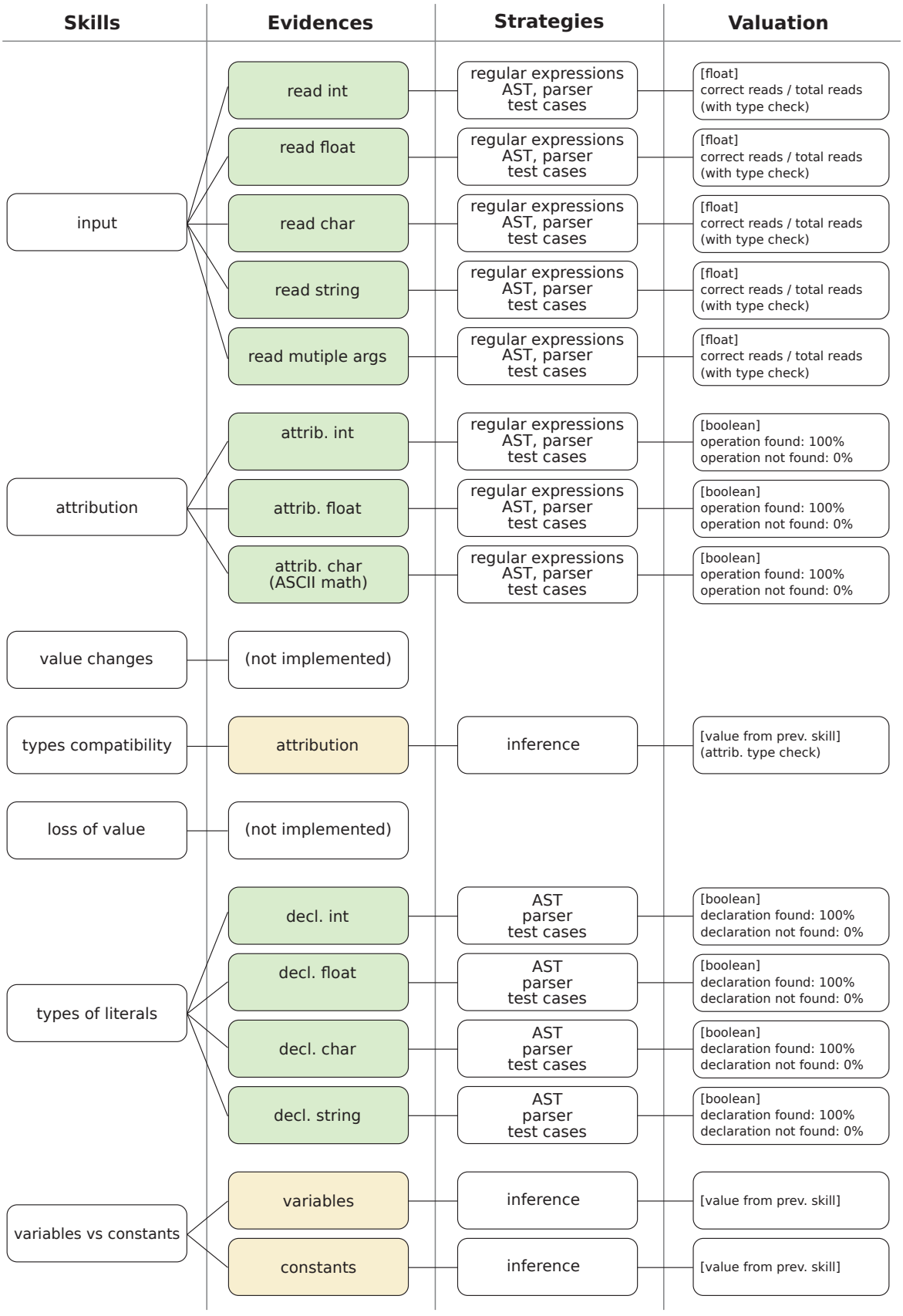


Figure C.2: Implemented evidences: part 2 of 7.

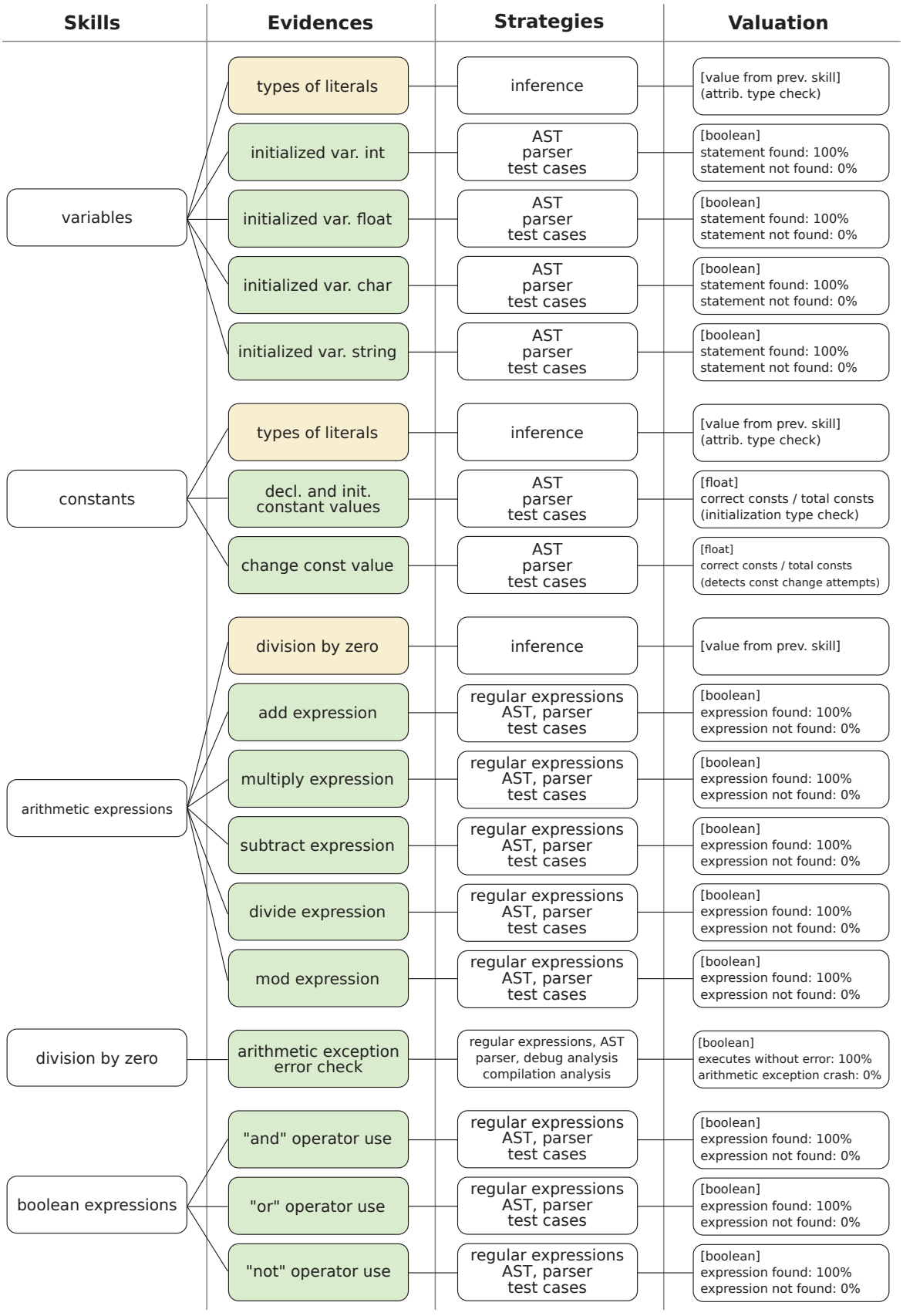


Figure C.3: Implemented evidences: part 3 of 7.

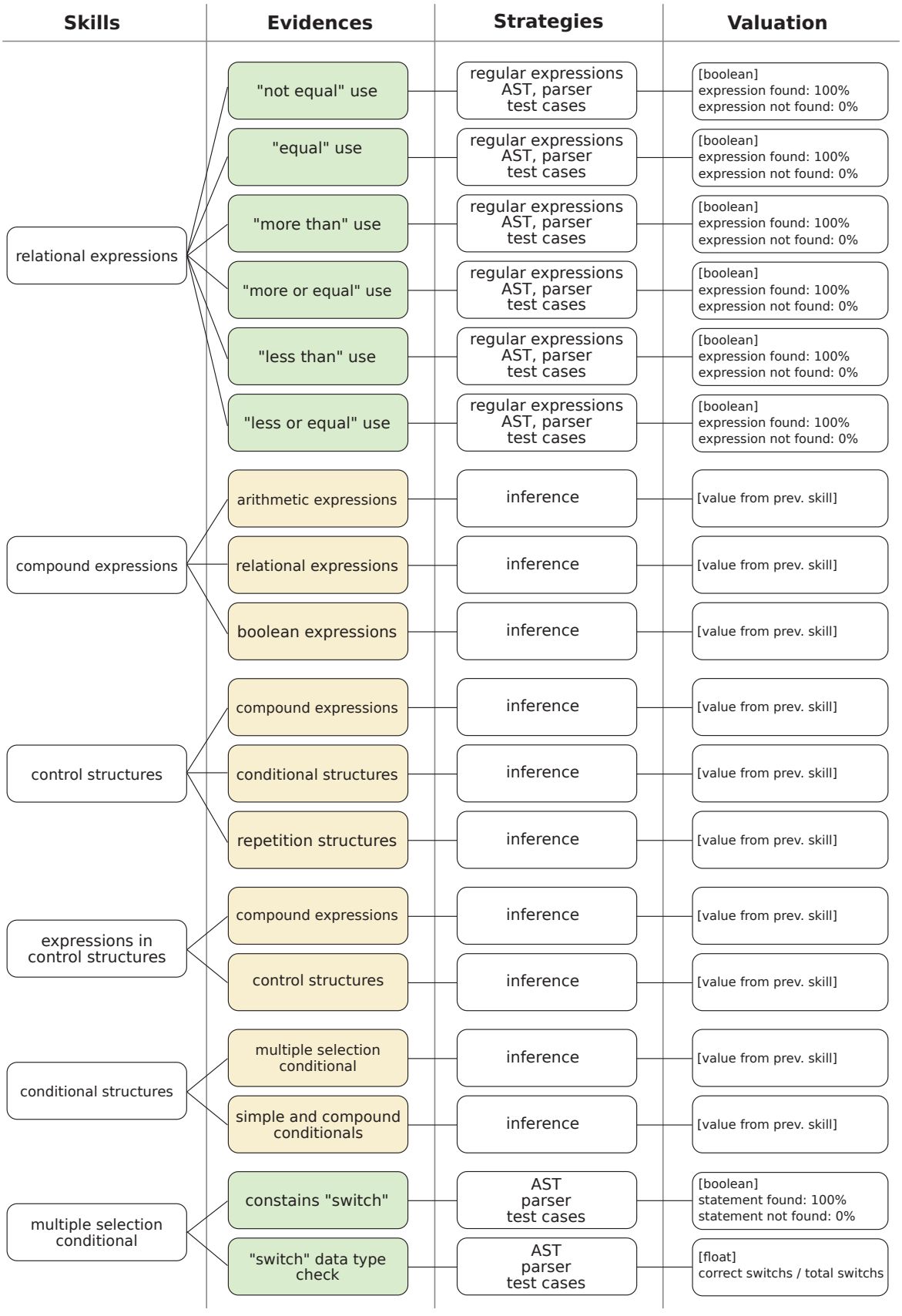


Figure C.4: Implemented evidences: part 4 of 7.

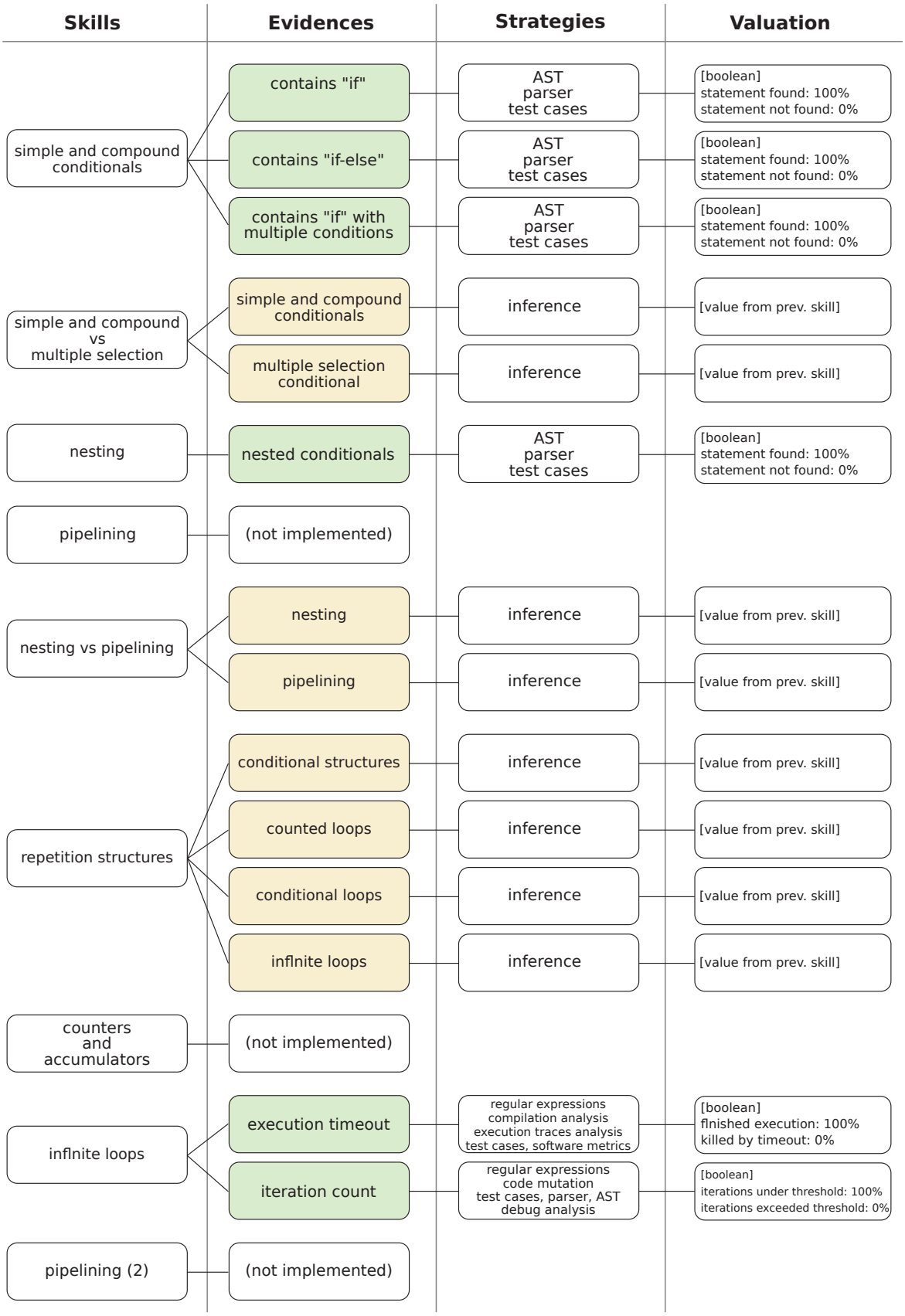


Figure C.5: Implemented evidences: part 5 of 7.

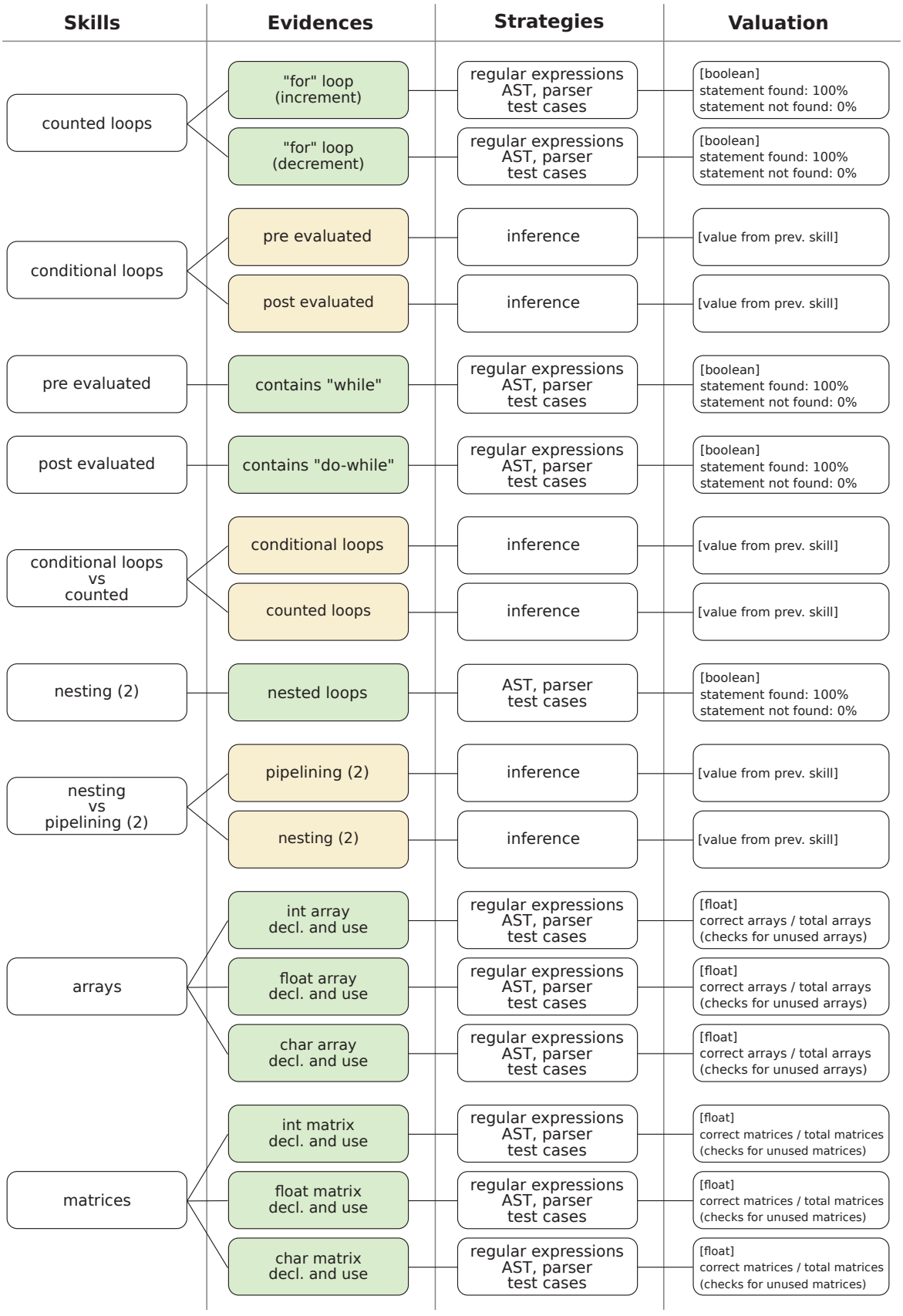


Figure C.6: Implemented evidences: part 6 of 7.

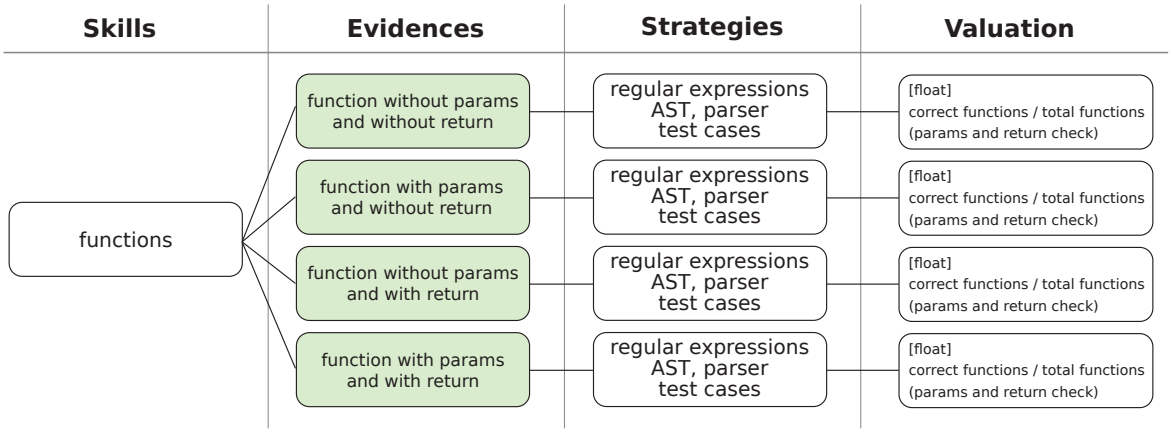


Figure C.7: Implemented evidences: part 7 of 7.

APPENDIX D – EVIDENCE MACHINE SETUP

D.1 SOURCE CODE AND EXECUTABLE

The Evidence Machine source code and executable (thesis snapshot) can be found online at http://bit.ly/doc_evidencemachine.

D.2 TESTED SYSTEM

- Linux Ubuntu 16.04.5 LTS (GNU/Linux 4.4.0-142-generic x86_64);
- Java 8 (1.8.0_181);
- Ruby 2.3.1.

D.3 INSTALLATION AND DEPENDENCIES

1. Install Java 8 (any method);
2. Install Ruby with cast gem:
   - 2.1. $ sudo apt-get install ruby2.3-dev
   - 2.2. Check ruby version (2.3.x): $ ruby -v
   - 2.3. Alterantive method to install ruby 2.3 on newer systems (run as root):
     - 2.3.1. # apt-add-repository
     - 2.3.2. # ppa:brightbox/ruby-ng
     - 2.3.3. # apt-get update
     - 2.3.4. # apt-get install ruby2.3 ruby2.3-dev
     - 2.3.5. Check ruby version (2.3.x): # ruby -v
     - 2.3.6. Install *cast* gem: # gem install cast
   - 2.4. $ gem install --user-install cast

D.4 PROPERTIES CONFIGURATION

Evidence Machine *app.properties* file contains configuration flags that can be changed before execution. Attributes configuration is detailed below:

- *port=4568* server port number (change if conflicts with some other service on server);
- *#log_file=stderr* output logs to standard console (uncomment by removing # to activate);
- *log_file=app.log* output logs to app.log file (comment with # if previous option is enabled);
- *log_requests=false* log http requests (debug API and web calls);

- *log_alg_executions=false* log evidence search algorithms execution (debug evidence search);
- *log_persistence=false* log local database storing (evidence search results storage);
- *student_model_require_login=true* login requirement can be disabled for (only) student model routes, set to false when integrating Event Machine to ITS through Database Adapter method (HTML <iframe>). Route *GET /student_model/:database/:team/:student* (and dependent API routes) will not require login if this attribute is false;
- *student_model_show_navbar=true* complementary change to previous attribute, disables student model navigation bar to avoid route changes when embedded to ITS.

D.5 EXECUTION COMMANDS

- `Command-Line Help: $ java -jar EvidenceMachine.jar --help`
- `Create Administrator User: $ java -jar EvidenceMachine.jar --create_admin <newusername> <password>`
- `Regular Execution: $ java -jar EvidenceMachine.jar`
- `Background Execution: $ java -jar EvidenceMachine.jar &`
- `Administrator login (localhost sample): http://localhost:4568/admin/`
- `Student login (localhost sample): http://localhost:4568/student_model`
- `User token generation (for use with REST API): http://localhost:4568/admin/user_manager`

APPENDIX E – EXERCISE SUBMITTER SETUP

E.1 SOURCE CODE AND EXECUTABLE

The Exercise Submitter source code and executable (thesis snapshot) can be found online at http://bit.ly/doc_exsub.

E.2 TESTED SYSTEM

- `Linux Ubuntu 16.04.5 LTS (GNU/Linux 4.4.0-142-generic x86_64)`
- `Java 8 (1.8.0_181)`
- `gcc version 5.4.0`

E.3 INSTALLATION AND DEPENDENCIES

1. `Install Java 8 (any method)`
2. `Install C-Language building tools:`
   `$ sudo apt-get install build-essential`

E.4 PROPERTIES CONFIGURATION

Exercise Submitter *app.properties* file contains configuration flags that can be changed before execution. Attributes configuration is detailed below:

- *port=7000* server port number (change if conflicts with some other service on server);
- *#log_file=stderr* output logs to standard console (uncomment by removing # to activate);
- *log_file=app.log* output logs to app.log file (add # if previous option is enabled).

E.5 EXECUTION COMMANDS

- `Command-Line Help: $ java -jar ExerciseSubmitterV2.jar --help`
- `Create Administrator User: $ java -jar ExerciseSubmitterV2.jar --create_admin <newusername> <password>`
- `Regular Execution: $ java -jar ExerciseSubmitterV2.jar`
- `Background Execution: $ java -jar ExerciseSubmitterV2.jar &`
- `Student logins creation: available at admin's panel`
- `Administrator login (localhost sample):`
  `http://localhost:7000/admin`
- `Student login (localhost sample):`
  `http://localhost:7000/student`

APPENDIX F – EVIDENCE MACHINE API DOCUMENTATION

This documentation can also be accessed directly by Evidence Machine server:
`http://[server_ip or domain]:port/api_docs.html`
e.g. `http://localhost:4568/api_docs.html`

All calls, except the API documentation, must be authenticated with user token, that means at least one administrator user must be created (check Evidence Machine execution commands on Appendix D).

F.1 REST DOCUMENTATION

GET api/databases.json
Requires Header:

```
Authorization -> Token {your token}
```

Returns:

```
{
   "DATABSE1_JavaClassName": "DATABASE1 String Description",
   "DATABSE2_JavaClassName": "DATABASE2 String Description",
   ...
   "DATABSEn_JavaClassName": "DATABASEn String Description"
}
```

GET api/:database/teams.json
Requires Header:

```
Authorization -> Token {your token}
```

Returns:

```
{
   "TEAM_ID": "TEAM NAME",
   ...
}
```

GET api/:database/:team/students.json
Requires Header:

```
Authorization -> Token {your token}
```

Returns:

```
[
   {
      "_id": "STUDENT ID",
      "username": "STUDENT NAME",
      "correct_answers": 0
   },
   {
      "_id": "STUDENT ID (2)",
      "username": "STUDENT NAME (2)",
```

```
            "correct_answers": 10
         }
      ]
```

GET api/:database/:team/:student/answers.json

Requires Header:

```
      Authorization -> Token {your token}
```

Requires Query/URL Param:

```
      sumarized -> true/false
```

Returns (sumarized=true):

```
      [
         {
            "answer_id": "answer1_id",
            "correct": true,
            "date": "2018-10-19"
         },
         ...
      ]
```

Returns (sumarized=false):

```
      [
         "ANSWER1 FULL SOURCE CODE AS STRING",
         "ANSWER2 FULL SOURCE CODE AS STRING",
         ...
      ]
```

GET api/:database/:team/:student/:answer

Requires Header:

```
      Authorization -> Token {your token}
```

Returns:

```
      Answer Full Source Code (raw text)
```

GET api/alg_version.json

Requires Header:

```
      Authorization -> Token {your token}
```

Requires Query/URL Param:

```
      alg_id -> {id from data-evidences.json, e.g: "alg_n12_declChar"}
      alg_runner -> RubyAlgRunner (or any other)
```

Returns:

```
      {
         "alg_id": "alg_n12_declChar",
         "version": 1
      }
```

GET api/alg_runners.json
Requires Header:

```
Authorization -> Token {your token}
```

Returns: Available AlgRunners

```
[
   "RubyAlgRunner"
]
```

POST api/alg_runner/run.json
Requires Header:

```
Authorization -> Token {your token}
content-type: application/json
```

Requires Body (json):

```
{
   database_id: "DATABASE_ID",
   team_id: "TEAM_ID",
   student_id: "STUDENT_ID",
   node_id: "n12-tipos-e-lit",
   alg_id: "alg_n12_declInt",

   "alg_runner": "RubyAlgRunner",

   answer_sumarized: {
      answer_id: "ansID",
      correct: true/false,
      date: "yyyy-MM-dd"
   },

   answer_source: "FULL ANSWER SOURCE CODE AS STRING"
}
```

Returns (json):

```
{
   "ansID": {
      "node_id": "n12-tipos-e-lit",
      "ev": [
         {
            "alg": "alg_n12_declInt",
            "key": "Decl. (int)",
            "value": 100,
            "info": "COMPUTED 100% FROM 1 DECLARATIONS",
            "extras": "EXTRAS (HTML FORMATTED)",
            "alg_version": 1
         }
      ]
   }
}
```

POST api/alg_runner/schedule.json
Requires exactly the same as api/alg_runner/run.json
Returns (raw text):

```
        "scheduled!"

NOTE: Result will be automatically stored in default SMPersistence
Database when the scheduler finishes the job.

GET api/alg_runner/schedule_start.json
Requires Header:

        Authorization -> Token {your token}

Returns (raw text):

        text informing what happened.

GET api/alg_runner/schedule_stop.json
Requires Header:

        Authorization -> Token {your token}

Returns (raw text):

        text informing what happened.

GET api/alg_runner/schedule_progress.json
Requires Header:

        Authorization -> Token {your token}

Returns (raw text):

        "Remaining: {number of executions to finish the job}"
```