Identifying Evidences of Computer Programming Skills Through Automatic Source Code Evaluation

Andres J. Porfirio[1,2], **Roberto Pereira**[1] **(advisor), Eleandro Maschio**[2] **(coadvisor)**

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brasil

[2]Coordenação do Curso de Tecnologia em Sistemas para Internet
Universidade Tecnológica Federal do Paraná (UTFPR)
Guarapuava – PR – Brasil

{andresporfirio,eleandrom}@utfpr.edu.br, rpereira@inf.ufpr.br

Abstract. *This research is contextualized in the teaching of computer programming. Continuous assessment of source codes produced by students on time is a challenging task for teachers. The literature presents different methods for automatic evaluation of source code, mostly focusing on technical aspects. This research presents the A-Learn EvId method, having as the main differential the evaluation of high-level skills instead of technical aspects. The following results are highlighted: updating the state of the art through systematic mapping; a set of 37 skills identifiable through 9 automatic source code evaluation strategies; construction of datasets totaling 8651 source codes.*

Resumo. *Esta pesquisa é contextualizada no ensino de programação de computadores. Avaliar os códigos fonte produzidos pelos alunos de forma contínua e em tempo hábil é uma tarefa desafiadora para os professores. A literatura apresenta diferentes métodos para avaliação automática de código fonte, em sua maioria focando em aspectos técnicos. Esta pesquisa apresenta o método A-Learn EvId, tendo como principal diferencial a avaliação de habilidades de alto nível ao invés de aspectos técnicos. Destacam-se os seguintes resultados: mapeamento sistemático para atualização do estado da arte; um conjunto de 37 habilidades identificáveis por meio de 9 estratégias automáticas de avaliação de código fonte; construção de bases de dados totalizando 8651 códigos fonte.*

1. Introduction

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

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

Assessing large amounts of source codes developed by a large number of students is a complex and exhausting task for teachers [Ullah et al. 2018, Rahman and Nordin 2007]. Therefore, investigating methods and developing support tools is a way to support teachers' work, and with more resources, they can do a better job.

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

From a systematic literature mapping it was found many works dealing with automatic assessment, 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, Ahmed et al. 2018]. Initial programming courses, however, usually have their syllabi focused on concepts and desired skills, not on technical aspects, which are often conveyed through classes and evaluated in specific situations where students are supposed to succeed only if they have mastered certain programming skill.

For this thesis, 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. In the computer programming context, we consider that students acquire knowledge through learning concepts, and later manifest it by applying different programming resources in source codes. Therefore, we consider that behavioral changes are marked by using previously unreported programming resources, thus suggesting evidence of new skills development.

When it comes to automatic source code assessment, [Hettiarachchi et al. 2013] presents two types: knowledge-based and skill-based. Knowledge-based assessment is described by the authors as a simplified form of assessment, usually easy to apply, but with a limited scope that may lead to just a quiz of facts about the area of study. Skills-based assessment, in turn, is described as more authentic and capable to assess higher-order cognitive skills, however, hard to apply. Also, knowledge-based assessment is related to simple aspects and rarely give any insight into the thought process students used to elaborate their responses (analogous to technical aspects evaluation previously mentioned), while skills-based assessment can be applied to evaluate high-level cognitive skills [Hettiarachchi et al. 2015].

The **central problem** addressed in this research is focused on the teacher's point of view, being characterized as the challenge of providing assessment of the students' source codes in a continuous and timely manner and, with this, identify manifestation of new programming skills. Therefore, the **main objective** of this research 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:

- To identify the state of the art and elaborate a literature mapping;
- To identify programming skills candidate to automatic evaluation;

- To identify a programming skill-set able to be automatically evaluated;
- To investigate strategies to automatically evaluate the identified skills;
- To implement the strategies as algorithms that receive student source codes as input and returns the identified skills as output;
- To implement a learner model to represent student knowledge based on a predefined skill-set;
- To apply strategies results as input data to feed the learner model;
- To provide resources to track student progress through the learner model; and
- To evaluate the proposed method regarding its automatic evaluation capacity.

2. State of Art

A systematic literature mapping[1] was conducted to identify what aspects of source code have been evaluated automatically, and what strategies have been adopted in literature. Based on 126 papers selected from databases ACM, IEEE, Scopus, Scielo and CEIE, our analysis revealed 43 different aspects of source code identified automatically via 25 different strategies.

The literature mapping revealed different attempts to use automatic strategies for source code evaluation. Data revealed that the most popular aspects addressed in literature are the ones dealing with automatic evaluation of source code as a whole, pointing to generic results such as *Functional Correctness* assessment and detection of *Semantic and Compilation Errors*. Regarding the strategies employed in the automatic assessment process, we identified *Test Cases* and *Unit Tests* as popular approaches. The majority of studies focused on applying one (or a few) strategy, providing specialized solutions to evaluate specific source code aspects. Using hybrid approaches to mix different strategies and evaluate multiple code aspects simultaneously was identified as a gap in the literature.

The systematic mapping suggested that the more aspects are evaluated in a source code the richer feedback possibilities could be and, consequently, more clues about students' programming skills could be provided. Results suggested that, although specialized solutions are adequate for specific contexts, methods combining multiple strategies to evaluate multiple aspects and to provide detailed and holistic feedback is a research gap. The A-Learn EvId method conceived in this thesis contributes to address this gap, improving our capacity to identify evidences of students' programming skills by automatically analyzing the source code they produce.

3. A-Learn EvId: Automatic Learning Evidences Identification Method

The Automatic Learning Evidences Identification (A-Learn EvId) method adopts 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, strategies employed, as well the results and learned lessons about limitations and possibilities.

Figure 1 presents a scheme for the method. Source code serve as input to evaluation, where different strategies are applied to identify evidences used for feeding a

[1]Details and results from the systematic mapping are under review in an international journal paper.

skill-based learner model (output). Evidences can be identified from single or multiple strategies, as well as from inference through combinations of previously evaluated skills.

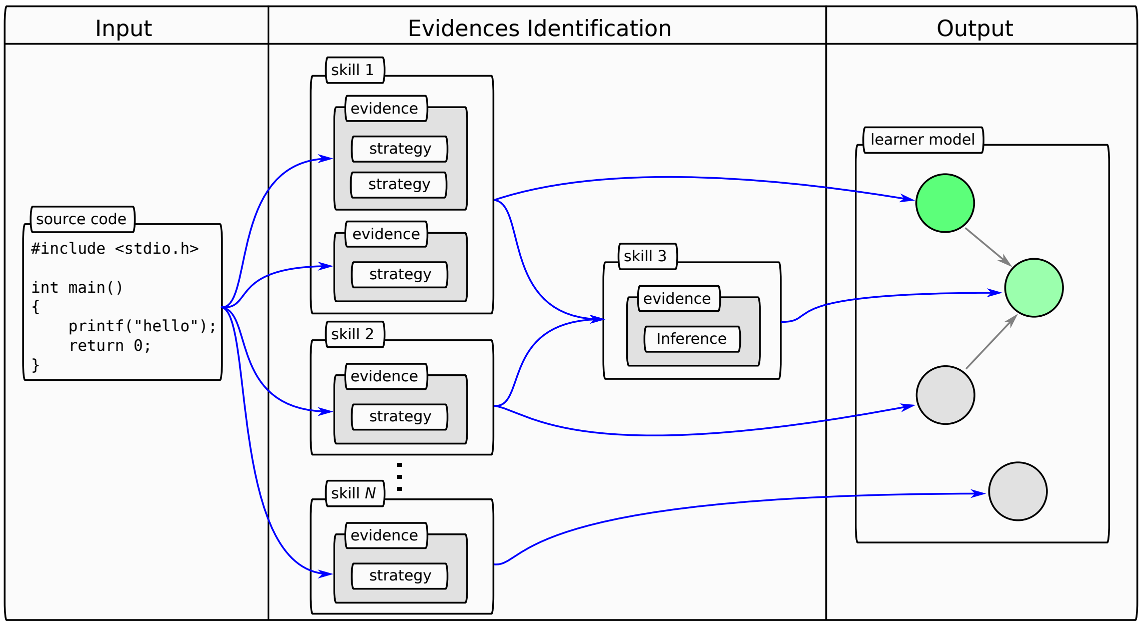


Figure 1. A-Learn EvId Method overview.

Figure 2 presents examples of skills and their respective valuation sources (when automatically identifiable). Skills are valuated by one or more 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 a specific programming resource).

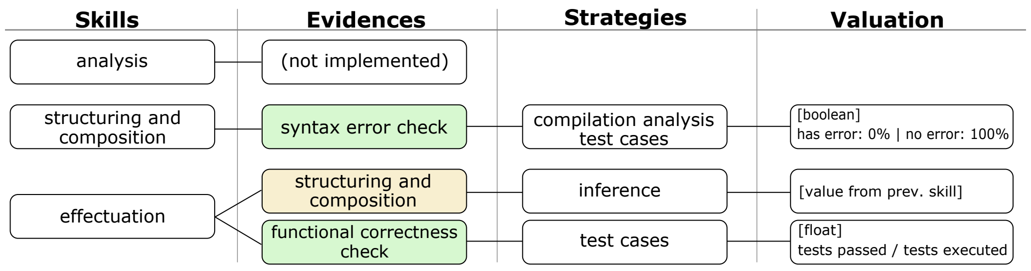

Skills
Evidences
Strategies
Valuation
analysis
(not implemented)
structuring and composition
syntax error check
compilation analysis test cases
[boolean] has error: 0% | no error: 100%
effectuation
structuring and composition
inference
[value from prev. skill]
functional correctness check
test cases
[float] tests passed / tests executed

Figure 2. Skills valuation sample.

Previous literature mentions the existence of desired skill sets related to training students in programming. Research from [Pimentel and Direne 1998] and [Maschio 2013], as well as real world programming courses syllabus analysis, were used to define a Full Skill Set containing aspects candidates to be implemented via strategies for automatic skills identification. Those aspects were then classified regarding their potential and priority for identification through automatic source code analysis, results are shown in Figure 3.

For each aspect representing a skill, automatic strategies were defined and implemented. Detecting evidences for different skills requires different strategies, some with

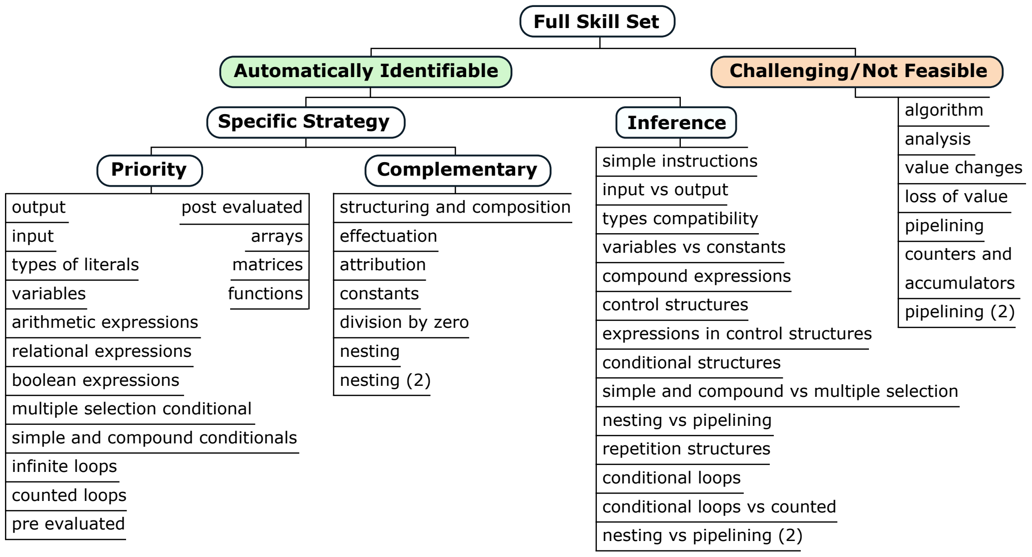


Figure 3. Skills categorized by their potential and priority for automatic identification.

easier automatic detection and others impractical. Considering the *automatically identifiable* skills previously defined, 9 of the 25 strategies from the systematic mapping were selected to compose the method: *AST* (Abstract Syntax Tree), *Code Mutation*, *Compilation Analysis*, *Debug Analysis*, *Execution Traces Analysis*, *Parser*, *Regular Expressions*, *Software Metrics* and *Test Cases*. Hybrid systems employing two or more of the selected strategies were also applied. Strategies were selected according to: (1) source code aspect it was capable to analyze; (2) type of analysis (static or dynamic); (3) availability of implementation documentation; and (4) adaptation possibility.

Finally, identified evidences are used to feed the learner model. For this research, the learner model is established to organize and present characteristics representing skills, developed or not by students, necessary for proper execution of programming tasks. Our learner model is based on the Overlay Graph presented by [Maschio 2013], oferring additional features:

- *Skill visualization detailed per source code*: ability to present details, in a source code or even in a code fragment, about where a student started showing a specific skill;
- *Skill valuation on multiple source codes*: ability to identify skills in multiple source codes (e.g., multiple exercises submissions) and apply a decision criterion to globally valuate students' skills;
- *Timeline based exercises selection*: considering multiple source codes evaluation, it allows to select specific source codes for analysing skills in specific programming topics/exercises and, with this, monitor students' progress;
- *Uncertainty treatment*: evolving from an Overlay Graph [Maschio 2013] to a Dynamic Bayesian Network to deal with unexpected situations possibly found in students' source codes, e.g., importing incompatible libraries, using unsupported/legacy syntax, submitting source code with incompatible encoding, and using operating system dependent resources.

4. Experiments and Results

Seven experiments were conducted to investigate the capabilities of the A-Learn EvId method. Experiments were carried out both in controlled scenarios (source codes specifically designed for testing purposes) and in real scenarios (dataset built from real-world exercise solutions, formally specified and collected). Table 1 summarizes our experiments according to the IMRaD format [Wu 2011], 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). Dataset size (number of source codes) is shown in the first column.

Table 1: Experiments summary (IMRaD structure).

Experiment	**Why?**	**How?**	**What?**	**So What?**
First (Pilot Test) (1 source)	To investigate preliminary automatic source code analysis strategies.	*AST* and *parser* strategies were applied to a reference source code, results were then compared to human 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 (29 sources)	To 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 Test) (29 sources)	To 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 (113 sources)	To 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 (142 sources mixing data from Third and Fourth experiments)	To 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 (3860 sources + 101 reference solutions)	To 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 (4434 sources + 84 reference solutions)	To demonstrate skill-based assessment using automatic strategies as means of identifying functionally correct but conceptually incorrect solutions.	Desired skills were defined for each programming exercise and then compared to students automatically detected skills.	Skill-sets comparison indicated source codes that deviated from reference solutions.	Skill-based assessment proved to be a valuable resource for locating conceptually incorrect solutions built with subterfuges.

In addition, two highlights are presented regarding the *Sixth* and *Seventh* experiments. The *Sixth* experiment specifically focused on analyzing student progress. Representing a more realistic environment, the experiment used a skill set based on topics commonly covered in real programming courses, showing that student progress can be monitored (e.g., between one exercise list and another), and that it is possible to identify when each skill start to be detectable. Figure 4 exemplifies a progress analysis where the vertical axis represents programming topics coverage and the horizontal axis represents ten exercise lists. The absence of learning evidence means the student did not submit a specific list of exercises.

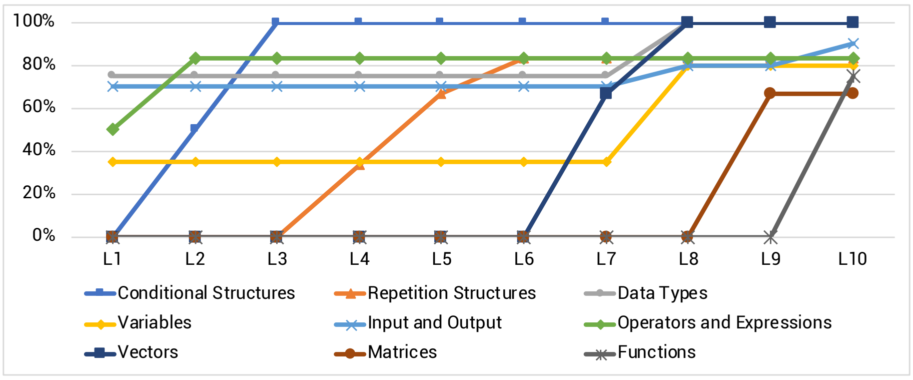
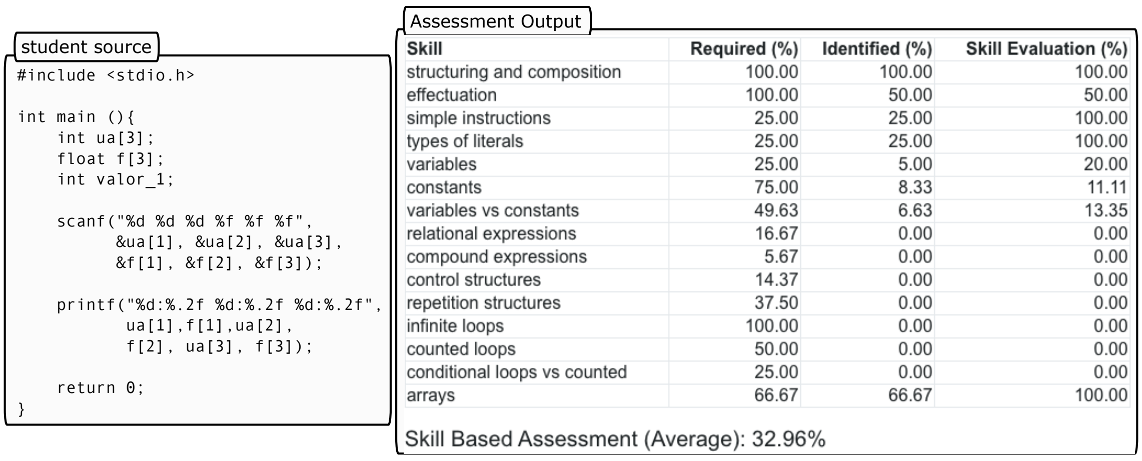


Figure 4. Example of a single student progress across the ten lists.

Finally, the *Seventh* experiment investigated the capabilities of skill-based assessment. Situations where subterfuges were used as means to achieve source code's functional correctness have been identified. Automatic search for learning evidence considering different programming skills has proved to be an interesting and effective method, providing indicative of potentially incorrect solutions where manual assessment is required. Figure 5 exemplifies the experiment with a sample solution to the following problem: *read* *an integer vector and a floating-point vector, each with three positions. Subsequently, traverse the vectors with a single repeating loop and print the sets in parallel (Int1:Float1, Int2:Float2, Int3:Float3).* The assessment output points out the student did not employ any loop-related resource, solving the exercise in a *forced way* (with hard-coded vector indices). This type of solution would be accepted by simpler strategies such as *test cases*, but specifications of a desired skills-set (e.g., higher valuation of loop-related concepts) prevents the solution from achieving a good score and indicates it for manual inspection.



student source

```
#include <stdio.h>

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

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

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

    return 0;
}
```

Assessment Output

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

Skill Based Assessment (Average): 32.96%

Figure 5. Source code skill-based assessment inspection.

5. Contributions

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

- The Systematic Literature Mapping provides an updated and rigorous panorama of automatic programming source code evaluation, presenting categorization schemes and important concepts, therefore informing this thesis and future research;
- A set of 37 standardized computer programming skills relevant to automatic identification;
- A set of 9 strategies, as well as their implementation and evaluation, responsible for automatic identifying skill evidences from students' source codes;
- A learner model capable of representing students' knowledge (skills acquired) identified by the automatic strategies and monitoring knowledge evolution;
- The A-Learn EvId: Automatic Learning Evidences Identification method;
- Seven source code datasets totalling 8651 C-Language programs (8436 from real-world exercise solutions, formally specified and collected; 215 specifically designed for testing purposes), as well as method and tools sharing[2];
- Publications: [Porfirio et al. 2016]; [Porfirio et al. 2017]; [Porfirio et al. 2018]. Papers presenting details and results from the systematic mapping and the seven experiments with the method are under review in international journals.

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

6. Conclusion

Considering the central problem, the challenge to provide assessment of the students' source codes in a continuous and timely manner and identify manifestation of new skills, the A-Learn EvId method was proposed and evaluated. 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, which provided resources for monitoring students' progress. Therefore, 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 improve method's accuracy and reliability.

This research resulted in updating and expanding the state of the art through systematic literature mapping. 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 students skills and detect potential concept flaws. Lastly, the contributions extend to the general context of Computer Science, where the acquisition of programming skills is a crucial activity for the vast majority of professionals. Thus, the development of resources that support the teaching of this activity tends to bring benefits and improve this process.

7. Acknowledgment

To professor Alexandre Direne (*in memoriam*), the initial advisor of this research.

References

[Ahmed et al. 2018] 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.

[Hettiarachchi et al. 2013] 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 et al. 2015] 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.

[Ihantola et al. 2010] 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.

[Jackson and Usher 1997] 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.

[Liang et al. 2009] 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.

[Maschio 2013] 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).

[Morris 2003] 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.

[Pimentel and Direne 1998] 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.

[Porfirio et al. 2016] Porfirio, A., Maschio, E., and Direne, A. (2016). Modelagem genérica de aprendizes com Ênfase em erros na aquisição de habilidades em programação de computadores. *Anais dos Workshops do Congresso Brasileiro de Informática na Educação*.

[Porfirio et al. 2017] Porfirio, A., Pereira, R., and Maschio, E. (2017). Atualização do modelo do aprendiz de programação de computadores com o uso de parser ast. *Anais dos Workshops do Congresso Brasileiro de Informática na Educação*, 6(1):1121.

[Porfirio et al. 2018] Porfirio, A., Pereira, R., and Maschio, E. (2018). Inferência de conhecimento a partir da detecção automática de evidências no domínio da programação de computadores. *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação - SBIE)*, 29(1):1553.

[Rahman and Nordin 2007] 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*.

[Souza et al. 2016] 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.

[Ullah et al. 2018] 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.

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

[Wilcox et al. 1976] 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 2011] Wu, J. (2011). Improving the writing of research papers: Imrad and beyond.