Source Code based Approaches to Automate Marking in Programming Assignments

Thilmi Kuruppu, Janani Tharmaseelan, Chamari Silva, Udara Srimath S. Samaratunge Arachchillage, Kalpani Manathunga, Shyam Reyal, Nuwan Kodagoda and Thilini Jayalath
Faculty of Computing, Sri Lanka Institute of Information Technology (SLIIT), Malabe, Sri Lanka

Keywords: Automated Assessments, Whitebox, Machine Learning, Syntax Graph, Static Tools, Literature Review, Programming.

Abstract: With the embarkment of this technological era, a significant demand over programming modules can be observed among university students in larger volume. When figures grow exponentially, manual assessments and evaluations would be a tedious and error-prone activity, thus marking automation has become fast growing necessity. To fulfil this objective, in this review paper, authors present literature on automated assessment of coding exercises, analyse the literature from four dimensions as Machine Learning approaches, Source Graph Generation, Domain Specific Languages, and Static Code Analysis. These approaches are reviewed on three main aspects: accuracy, efficiency, and user-experience. The paper finally describes a series of recommendations for standardizing the evaluation and benchmarking of marking automation tools for future researchers to obtain a strong empirical footing on the domain, thereby leading to further advancements in the field.

1 INTRODUCTION

1.1 Background and Motivation

Programming assignments are an essential element in computer programming modules taught at university. With the growth of class sizes, evaluating assignments become challenging (Higgins, Gray, Symeonidis, & Tsintsifas, 2005) and researchers explore novel methods to automate assessments. Marking automation has advantages like speed, consistency, reduced need for post-marking moderation, better utilisation of human-hours, and eliminate favouritism and bias from the marking (Ala-Mutka, 2005). As an illustrative example, the authors' university has 10 programming modules in the undergraduate programs, each with 2 assignments, with an average of 1000 students, resulting in 20,000 programming assignments per semester. An average time of 20 minutes was estimated to mark each programming assignment which results in 400,000 human hours spent on marking each academic semester.

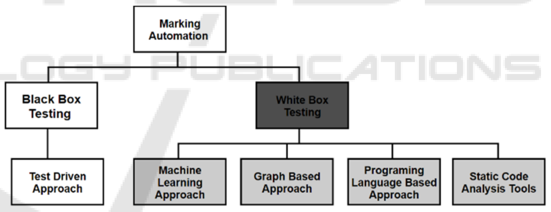

Figure 1: Hierarchy of marking automation approaches.

Automatic marking approaches can fall into two broad categories – Blackbox testing and Whitebox testing as shown in Figure 1.

Blackbox testing, as per its definition, focuses on the program producing the expected output for a given input. A number of Blackbox testing approaches like Unit Testing exists and is used in a number of commercial and non-commercial tools (Rahman, Paudel, & Sharker, 2019) like REPL.it, GradeScope, Moodle-Extension, Stacscheck by St Andrews, etc. The major drawback of Blackbox testing is that (a) programs should be developed with an interface or API to provide inputs and obtain outputs (via console, files, methods arguments and return values, etc.) and (b) to produce an output, the program should be syntactically correct, and run without errors – both of which cannot be guaranteed with student

Kuruppu, T., Tharmaseelan, J., Silva, C., Arachchillage, U., Manathunga, K., Reyal, S., Kodagoda, N. and Jayalath, T.
Source Code based Approaches to Automate Marking in Programming Assignments.
DOI: 10.5220/0010400502910298
In *Proceedings of the 13th International Conference on Computer Supported Education (CSEDU 2021) - Volume 1*, pages 291-298
ISBN: 978-989-758-502-9

submissions. On the other hand, Whitebox Testing assigns a mark by reading the submitted source code, whether Blackbox testing is possible or not. It is usual practice in universities to include a marking rubric that has both a Blackbox and Whitebox component e.g. 50 marks for the program producing the correct outputs for a given set of inputs, and 50 marks by reading the source code – looking for correctness, neatness and good coding practices, etc.

1.2 Contributions and Organisation

The main objective of carrying out this research work is to study the holistic domain of marking automation of programming assignments and provide a panoramic view of the existing research approaches in the domain. As illustrated in Fig.1, authors will study only Whitebox testing approaches from four different aspects, i.e., Machine Learning (ML) approaches, source graph (SG)-based approaches, programming languages (PLs) based approaches and static code analysis (SCA) approaches. As programming assignments, authors consider a piece of any computer program provided as students' answers. This paper does not address parts of the assignment which have a different component in the SDLC (e.g. design), where submissions may have flowcharts, UML diagrams, test reports, coverage reports, and integration reports etc.

The research paper is structured as follows: in section II, authors provide a review of four identified approaches i.e. ML, SG, PLs, and source code analysis. These are critically evaluated on three main benchmark criteria i.e. accuracy, efficiency, and user-experience (UE). The section III provides a 'meta-review' based on cross-literature-comparison to identify the strengths and weaknesses in the current state-of-the-art, thereby recognizing the suitability, scope, applicability, limitations, and research gaps. A discussion of the findings in sections II and III, providing the authors interpretation, insights, and arguments is available in the section IV. Finally, the section V provides recommendations based on the findings and discussion, for further advancement of the field.

2 LITERATURE REVIEW

2.1 Machine Learning (Ml) Approaches

ML is one of the main approaches used to evaluate the source code marking. Some existing ML approaches follow the holistic approach. In the study of (Srikant & Aggarwal, 2014), Linear Ridge Regression and Support Vector Machine (SVM) combined with different kernels based on rubric and hand-graded predictions have been used as the regression techniques. Also, (Srikant & Aggarwal 2014) uses random forests to determine the closeness of the logic. The studies used one-class modelling as the Prediction model. According to the results, ridge regression showed better cross-validation and error validation results than SVM regression. It showed that more than 80% of the predicted grades are within its corresponding expert rated grades. Moreover, regression against expert-grades can provide much better grading than the ubiquitous test-case-pass based grading and rivals the grading accuracy in marking.

Some studies provide personalized feedback for the student submissions using ML, based on factors like code quality and similarity (Zhou, et al., 2018) or either fix the code minimally and present feedback for a given solution against a marking rubric (Singh, Gulwani, & Solar-Lezama, 2013). In (Zhou, et al., 2018) similarity model distances were found by transforming the features derived from the assignment. (Singh, Gulwani, & Solar-Lezama, 2013) used a two-phase translation solution to find minimal corrections. The effectiveness of the tool was 64% of feedback for all incorrect attempts within 10 seconds on average. The results were more convincing in (Zhou, et al., 2018) approach that could be viable for marking automation with its efficiency and the accuracy rate.

A Rule-based system and linear regression models had been used to predict the position of compilation errors and assess uncompilable codes (Takhar & Aggarwal, 2019). This was achieved using n-gram based token prediction approach which is called as Make Compliable (MC), Rule Relaxation (RR). Then a ML model was developed by combining MC and RR as RRMC. Performance measurements were median for MC is 0.73, a higher value than for RR (which was 0.69). RRMC approach resulted with a mean of 0.71 correlations offering best results with reduced time and effort. Another technique was feature extraction using ML approaches to automate marking student codes (Russell, et al., 2018). Linear Support Vector Classification (Linear SVC), Gaussian Naive Bayes and Multinomial Naive Bayes have been used as three practical algorithms to classify exam submissions and a holistic approach to pick up the pattern from manually marked submissions. Convolutional Neural Network (CNN) and Recurrent Neural Network were used for feature extraction (Russell, et al., 2018). It uses Linear SVC and Naïve Bayes as scoring functions, and it was found that the Multinomial Naïve Bayes (MNB) over Gaussian Naïve Bayes (GNB) where the most accurate prediction gave a hit-rate of 73.39%. MNB with weighted scoring had a prediction accuracy of 79.03%. Hence, Naive Bayes algorithm shows promising results. In (Russell, et al., 2018) the best results achieved using features learned via CNN and classified with an ensemble tree algorithm.

2.2 Static Code Analysis (SCA) Tools

Striewe & Goedicke (2014) suggested the two approaches regarding the analysis of source code as Abstract Syntax Tree (AST) and Abstract Syntax Graph (ASG). Furthermore, recursive methods can identify using ASG, based on the inter-dependency arcs in between operation declaration and its invocation (Striewe & Goedicke, 2014), (Striewe M. , 2014). In (Blumenstein, Green, Nguyen, & Muthukkumarasamy, 2004), the famous code analysis algorithms known as Abstract Interpretation for execution path analysis has been introduced. It produces higher accuracy, but results indicated lower efficiency ratio. The authors have suggested how the previous approach can be combined with the classic analysis algorithm and elevate the level of performance. Automated Static Analysis Tool (ASAT) had been considered by couple of tools proposed by different studies. Gallier (2015) discussed regarding the logical inference introduced with the tool called SMT solver. As illustrated in Rautenberg (2010), horn clauses and logic programming tools would be precursor to achieve the most optimum solution by combining abstract interpretation and the logical inference techniques (Ala-Mutka, 2005), (Cousot, Cousot, & Mauborgne, 2013), (Vert, Krikun, & Glukhikh, 2013).

Another study (Vert, Krikun, & Glukhikh, 2013) enumerated the most dominating SCA tools such as Coverty SAVE Platform, Astree, PC-Lint/Flex Lint, and Aegies. In addition, study of (Digitek-labs, 2011) would be beneficial due to its unique feature analysis considering the dimensions of accuracy and performance. In the study of (Buyrukoglu, Batmaz, & Lock, 2016), authors performed a comparative analysis of SCA tools to check the coding conventions of Java which explicitly discussed the single versus multi files analysis and their prevailing strengthens and weaknesses. Moreover, (Vetr`o, 2014) suggested that the learning approach would be significant in detecting the raised false positives in the codebase and successfully removing them using the approach of Bayesian inference-based learning model with training a neural network.

2.3 Programming Language based Approaches

(Blumenstein, Green, Nguyen, & Muthukkumarasamy, 2004), (Souza, Felizardo, & Barbosa, 2016) provide an extensive list of assessment tools used to evaluate programming assignments and they had presented series of classification schemas like assessment types (manual, automatic or semi-automatic), approach (instructor-centred, student-centred or hybrid), specialization (tools for contests or for quizzes or for testing) and a comprehensive analysis of tools (considering the type of verification, language compatibility, interoperability with IDEs and LMS, etc.) that assist evaluation of programming assignments. Another segmentation is that dynamic analysis of codes assessing functionality, efficiency, and testing skills of student's vs. static checks to analyze and provide feedback for style, programming errors, software metrics, and even design (Ala-Mutka, 2005).

In (Blumenstein, Green, Nguyen, & Muthukkumarasamy, 2004), authors introduced a system to assess the student programming assignments which are written using Java and C. To extend the functionalities of the current system, authors have introduced a Java framework where new marker modules could be added manually. The system GAME caters a better UE by providing a simple GUI and the evaluation summery. Authors have increased the flexibility through java framework. It concluded that system could produce high accuracy results due to evaluation process of having specific rubric for assignments. A separate interpretation of the system will be accessible to the students to execute in future developments and collect the feedback.

CourseMarker (Higgins, Gray, Symeonidis, & Tsintsifas, 2005) can be used to mark command-line driven Java and C++ programming assignments with correct configurations. The evaluation happens using several stages and internal tools as typographical tool (to check layouts), dynamic tool (solutions compared against test data), feature tool (check specific syntaxes such as "if-then-else" blocks), flowchart tool, object-oriented tool and logic tool (to check logic circuits) marks at least as well as humans do, provides on-demand, impartial feedback, and as a bonus saves hundreds of marking hours for the academic staff. Another study (Buyrukoglu, Batmaz, & Lock, 2016) employs semi-automatic code assessing, considering the program structures like sequence and iteration. Novice's code is compared with the manually marked sample and using code similarity the rest of the scripts are assessed. Human markers may also provide feedback. The process includes a segmentation stage, codifying process, grouping, and marking.

Automated assessments using a Domain Specific Language (DSL) called Output semantic-similarity Language (OSSL) had been proposed in (Fonte & Cruz, 2013). A Flexible Dynamic Analyzer architecture with the components like OSSL and its grammar to specify output specification are extensively discussed. The approach supports for partial marking of code scripts and for interoperability with any automated grading system that support for Learning Objects. Immediate evaluation is possible by running the program over a set of predefined tests and comparing each result (the actual output produced by the submitted code) against an expected output specification. Yet, authors had not conducted a proper evaluation of the approach or the DSL, which may limit the paper as a conceptual model.

2.4 Source Graph based Approaches

Graph is a mathematical model that shows connections between elements where it describes rules over sets of nodes and edges of a graph. A program written in a PL can be transformed into a syntax tree by a parser. When additional information such as bindings are included in the representation, the syntax tree is extended into a syntax graph (Rensink & Zambon, 2009). One study gives a review of tools useful in automated grading and tutoring in the context of object-oriented programming with Java (Striewe & Goedicke, 2014). Authors emphasize on the necessity of tools being able to process multiple source files. According to authors, in pre-processing steps, extending syntax trees to syntax graphs with additional information helps achieving more flexible and exercise specific configurations. When such automated tools are more general, more effort is necessary to perform specialized tasks. Since learning scenarios may require very specialized and even exercise specific checks which are not among the standard checks offered by program analysis tools, these authors suggest that integration of several tools can be more productive.

In (Striewe M. , 2014), the suitability of three data structures: Strings, Trees and Graphs has been evaluated. As mentioned, trees are limited because they only allow one parent per node hence, they could not represent different kinds of relations like "is defined in" and "is called by" between elements. To check for recursive methods, it is necessary to have this information. Graph-based representations can be used to store this information, because they allow an arbitrary number of connections between nodes. Therefore, authors concluded that the attributed graphs are an appropriate representation of any kind of code analysis. Study used TGraphs to handle attribute graphs where it designed for efficient handling and analysing of large graphs. Queries on this graph format are expressed using a query language named GReQL. The GReQL for queries on TGraphs can be extended by user defined functions. In (Striewe M. , 2014) two solutions have been used to analyse the syntax graphs: a graph transformation tool and a graph query engine. As concluded, both techniques can create pre-defined generic sets of rules that are independent of specific exercises.

3 THE META-REVIEW

3.1 Machine Learning based Approaches

Grades for the programming assignments can be evaluated in many ways, among those, ML is outweighing the other approaches (Korkmaz & Correia, 2019). The reason behind this is the ability of "learning" of the model. It can analyse the new codes and learn, hence marking structure is up to date and more accurate. Here, the focus is on the accuracy, efficiency and configurability of different ML algorithms found in the literature. Linear ridge regression, random forests and kernel based SVM, Naive Bayes (Multinomial Naive Bayes and Gaussian Naive Bayes), SVM can be identified as most popular and effective algorithms used in the literature. When comparing, linear ridge regression largely shows better cross-validation and error validation results than random forests and kernel-based SVM (Srikant, & Aggarwal, 2014). Moreover, regression can provide much better grading than the test-case-pass based grading. When the uncompilable codes to be corrected up to some level where if the exams give more priority to algorithms and logic, not for the program syntaxes. In that case linear ridge regression models were best. However, results also shows that models built using both, semantic features and test-cases shows better results. Moreover, developing of problem independent grading techniques which may be facilitated by efficient one-class modelling techniques is shown as a necessity. On the other hand, the proposed algorithms need to train to use for more diverse problems which is a need in the future.

3.2 Static Code Analysis Tools

Number of SCA tools exponentially grow with the advancement of the technology and those support for different PLs in different scale, and this paper mainly focuses on their contributions over evaluating programming assignments. To analyse the source code there are two approaches that can be used as AST and ASG and the ASG is an enriched version of the tree, but it comprises additional arcs (Striewe & Goedicke, 2014). Compared to the AST, the ASG contains a couple of advantages as it can easily detect irrelevant pieces of code, for instance, unused methods can be identified using the method declaration nodes and ASG is capable of easily identifying recursive methods based on the cyclic dependency arcs in-between method declaration and method call nodes (Novak & Krajnc, 2010). Consequently, most SCA tools tend to use ASG as its main analytical technique. Some studies suggest generating ASG from AST to increase the accuracy as a valuable pre-processing step before evaluating the source code in programming assignments (Striewe & Goedicke, 2014), (Striewe M. , 2014), (Gallier, 2015), (Rautenberg, 2010), (Cousot, Cousot, & Mauborgne, 2013), (Vert, Krikun, & Glukhikh, 2013).

Furthermore, when analysing large volumes of programming assignments, efficiency and accuracy are the most significant factors to be considered. But when increasing the accuracy, the performance factor would compensate, hence achieving an optimum performance for the highest precision is the key concern here. To increase the accuracy, "Abstract interpretation" is a well-known code analysis algorithm (Cousot P. , 1996) which analyses every execution path without violating the reliability of the analysis. It analyses paths separately to achieve maximum accuracy, but it degrades the performance factor in a significant ratio. As a solution the joint path analysis could be used, which assumes joining variable values at flow conjunction point. However, that leads to lower accuracy, but it gains satisfactory performance level. Another possible approach to increase the precision while using the joint path analysis is combining the classic analysis algorithm (Glukhikh, Itsykson, & Tsesko, 2012). Nevertheless, this dependency analysis partially compensates on reducing the level of accuracy because of joining paths during the analysis. To further optimize the situation dependency analysis implemented using logical inference methods and tools such as first and higher-order logic (Gallier, 2015), SMT solvers, and Horn clauses and logic programming tools would be instrumental. Considering the literature Patric Cousot suggested a mathematical model of integrating abstract interpretation with a logical inference as the

Table 1: Emerged Static Code Analysis Tools.

Tool	Capability	Language support	Related work
Coverity SAVE	1. Provide accuracy 80% - 90% 2. Detects 7 errors per 1000 LOC 3.Understand Patterns and Programming idioms (Design Patterns intelligence) 4.Seamless integration with any build system	C/C++, Java and C#	(Vert, Krikun, & Glukhikh, 2013)
Astree	1. Capable of identifying run-time errors 2. capable of analyzing medium-scale industry-based projects. 3. Analyze 100,000 (LOC) 4. The tool can detect dead-code and uses abstract semantic domains 5. Perform their main analysis procedure from top to bottom	C language	(Cousot, Cousot, & Mauborgne, 2013)
FlexeLint/ PC Lint	1. This performs data flow/control flow analysis 2. It performs the interprocedurally analysis 3. Tool enhances user experience by user-defined semantic checking of functional arguments	C/C++ programs (FlexLint for UNIX, Mac OS, Solaris platforms) PC-Lint for Windows	(Vert, Krikun, & Glukhikh, 2013)
Aegis	1. Simple dependency analysis to increase the accuracy. 2. Can be used as for defect detection and for decision of other program engineering tasks 3. Used to analyze many open-source projects	C90 and C++ 98 source code	(Digitek-labs, 2011),
Checkstyle	1.Find naming convention errors 2.Identify whitespaces, line, length specific errors, wrong use of brackets	Java code	(Ashfaq, Khan, & Farooq, 2019)

most optimized solution (Ala-Mutka, 2005), (Cousot, Cousot, & Mauborgne, 2013), (Vert, Krikun, & Glukhikh, 2013). Currently there are lots of elegant tools available, depicted in Table 1 as a result of research and development activities which gains high accuracy and performance levels.

Problem in most code analysis tools is that those generate false warnings. It causes a significant drop in the precision level of the outcome. The literature attempted to fulfil the gap in removing static analysis warnings on software quality metrics. A neural network has been trained and the Bayesian inference-based learning model (Vetr`o, 2014) used on top of the extracted byte code, and AST used as the underlying analysis technique which improves the prediction of false positive in SCA.

3.3 Programming Language based Approaches

Literature can be found with classifying student-led or teacher-led tools considering efficiency and extensibility has been addressed by LMS or IDE extension whereas usability has achieved using automated feedback or verification techniques (Souza, Felizardo, & Barbosa, 2016). GAME (Blumenstein, Green, Nguyen, & Muthukkumarasamy, 2004) tool offers a good UE with a simple GUI and a summary of the evaluation. Authors have increased the flexibility of the system by introducing a Java framework to extend the functionalities. Since the student answers are evaluating over a marking schema, accuracy of the results is high. CourseMaker (Higgins, Gray, Symeonidis, & Tsintsifas, 2005) can mark at least as well as humans do, though they have not given any statistics on the fact. The tool is reliable as it uses Java's exception handling mechanisms and provide impartial feedback on demand while saving time. Extensibility has been considered by customizing the exercises or by customizing the marking server, enabling integration of additional features. In (Buyrukoglu, Batmaz, & Lock, 2016), humans need to mark only 32 from a sample of 153 (27%) which indicates a reduction of human effort. This system provides feedback to students. But the complete process of marking highly depends on the question preparation and teachers need to pay attention to detail at that stage. DSL based approach (Fonte & Cruz, 2013) had not carried out a proper evaluation study. Hence, accuracy of the tool or efficiency had not been addressed. Yet, authors claim that their approach is user-friendly and interoperable with any other system that support to Learning Objects.

3.4 Source Graph based Approaches

In (Striewe & Goedicke, 2014), a review on tools useful in automated grading and tutoring is conducted. In reviewing the tools, authors have given attention on graphs on how to increase the configurability of the tools when the syntax graphs are used. Authors have not given an attention specifically on syntax graphs on the context that how the syntax graphs affect on the accuracy or efficiency of a tool. But authors have made a general comment on how integration of different tools could increase efficiency. Syntax graphs are an appropriate representation for the code analysis (Striewe M. , 2014). It makes the solutions more accurate and flexible. Authors mention that the attributed graphs make it efficient to handle large graphs. Also, authors have used a query language on the graphs, which provides the configurability on for the user defined functions. In another study, graph transformation had been used for source code analysis (Rahman, Paudel, & Sharker, 2019) which concluded that the graph transformations improve the accuracy and flexibility of the solution.

4 DISCUSSION

The existing literature review discussed four major approaches: ML, SG-based, PL-based, and SCA. The outcomes were critically evaluated based on the dimensions: accuracy, efficiency, and user experience. SG-based approach highlighted that, syntax graphs provide significant improvement in accuracy and flexibility. It further emphasized usage of ASG over AST as a valuable pre-processing step.

PL based approaches suggested some tools that enhance the user experience, flexibility, and extensibility factors and few cases on how marking accuracy is more reliable when compared marking by a human-counterpart. Similarly, there are some tools discussed under SCA, how it reached the accuracy considering the metrices like LOC and types of errors. Moreover, SCA show strengths and weaknesses in algorithms to reach highest precision while keeping the optimum performance. It proposed a mathematical model that integrates abstract interpretation with a logical inference as the most optimized solution. Authors highlight the most popular algorithms that researchers used in the ML approaches. Linear ridge regression model could be predominant due to the unique capability of logic-based evaluation over syntactical evaluation in programming assignments. In the literature of ML, it is proposed that the algorithms should be trained for diverse problems to fulfil the future expectations.

Across these four identified approaches, it was observed that some critical aspects had not been considered in the evaluations. Most works (except a very few (Singh, Gulwani, & Solar-Lezama, 2013)) have not compared their novel approaches against the obvious baseline condition – what if the same sample was marked by a human? Also, many had not considered inconsistencies of accuracy and unbiasedness in comparison with human marking. Previous studies could not be cross compared since authors had measured different aspects, e.g. some report on the efficiency of the solution (Striewe & Goedicke, 2014), (Novak & Krajnc, 2010) some (rarely) on the user-friendliness and configurability of the tools proposed (Ala-Mutka, 2005), (Striewe & Goedicke, 2014), (Novak & Krajnc, 2010), (Ashfaq, Khan, & Farooq, 2019) and some on the accuracy (Srikant & Aggarwal, 2014), (Takhar & Aggarwal, 2019), (Novak & Krajnc, 2010). Therefore, it is not evident enough to conclude whether a solution is accurate yet inefficient, or accurate and efficient yet difficult to configure etc. Given the maturity of the domain and the plethora of marking automation techniques proposed, a strong empirical footing on each approach, measured using generic comparable metrics is overdue.

Lack of generic "test-samples" for evaluating the efficiency, accuracy and user-friendliness of each approach is identified a significant challenge. Currently, each method is evaluated with different marking samples made up by the authors, which therefore cannot be cross compared. Another major limitation in most work (except a few (Srikant & Aggarwal, 2014), (Takhar & Aggarwal, 2019)) is that they cannot be seamlessly integrated to existing educational platforms such as Moodle. Teachers need to install these tools, while configuring runtime environments and handling dependencies. Further, this process may include obtaining the submissions from the LMS platform, marking them using the tool, and uploading the marks back to the LMS. Hence, interoperability and portability of tools is a non-trivial consideration. Though some studies provide feedback (Zhou et al., 2018), (Singh, Gulwani, & Solar-Lezama, 2013) many have not addressed providing feedback for student improvement. This is essential and becomes more challenging for human markers when addressing larger classes.

5 RECOMMENDATIONS

After scrutinizing the literature, authors would like to contribute with some recommendations for marking automation. A set of generic "marking-samples" to evaluate each proposed marking automation mechanism seems critical. This will provide a strong empirical footing on the accuracy of each method. As a baseline condition, these could be marked by a human for comparison. For further accuracy, a crowdsourcing approach could be used to obtain the human-given mark for the "marking samples". Secondly, these marking samples need to be applicable across domains, such as assignments written in procedural, functional and object-oriented, as well as covering student-submissions that include meta-files (e.g. XML, properties, and other configuration files). Third, it is recommended to standardize a "recipe" of activities and measurements for evaluating the UE of each tool for both teacher and student. It is obvious that a tool which has utility has more tendency to be used if it is user friendly. Student feedback is key; hence novel marking automation tools need to provide feedback with marks within the same tool, or suite of tools which work together. Automatic tools emphasize the need for careful pedagogical design of assessment settings and these solutions need to be interoperable and portable.

6 CONCLUSION

This paper addresses how literature had studied marking automation domain. Strategies had been analysed on four specific approaches to propose a series of recommendations to be adhered for future research in this domain.

ACKNOWLEDGEMENTS

Authors would like to extend their gratitude to Sri Lanka Institute of Information Technology for extending support to carry out this research work.

REFERENCES

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

Ashfaq, Q., Khan, R., & Farooq, S. (2019). A Comparative Analysis of Static Code Analysis Tools that check Java Code Adherence to Java Coding Standards. *2nd International Conference on Communication, Computing and Digital systems (C-CODE)*. 98-103.

Blumenstein, M., Green, S., Nguyen, A., & Muthukkumarasamy, V. (2004). Game: A generic automated marking environment for programming assessment. *International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004*, 212-216.

Buyrukoglu, S., Batmaz, F., & Lock, R. (2016). Increasing the similarity of programming code structures to accelerate the marking process in a new semi-automated assessment approach. *11th International Conference on Computer Science & Education (ICCSE),*371-376.

Cousot, P. (1996). Abstract interpretation. *ACM Computing Surveys (CSUR)*, 324-328.

Cousot, P., Cousot, R., & Mauborgne, L. (2013). Theories, solvers and static analysis by abstract interpretation. *Journal of the ACM (JACM)*, 1-56.

Digitek-labs. (2011). *Static analysis framework*. Retrieved from digiteklabs: http://www.digiteklabs.ru/en/aegis/platform/

Fonte, D., & Cruz, D. D. (2013). A flexible dynamic system for automatic grading of programming exercises. *2nd Symposium on Languages, Applications and Technologies,* 129-144.

Gallier, J. H. (2015). *Logic for computer science: foundations of automatic theorem proving.* Courier Dover Publications.

Glukhikh, M. I., Itsykson, V. M., & Tsesko, V. A. (2012). Using dependencies to improve precision of code analysis. *Automatic Control and Computer Sciences*, 338-344.

Higgins, C. A., Gray, G., Symeonidis, P., & Tsintsifas, A. (2005). Automated assessment and experiences of teaching programming. *Journal on Educational Resources in Computing (JERIC)*.

Korkmaz, C., & Correia, A. P. (2019). A review of research on machine learning in educational technology. *Educational Media International*, 250-267.

Novak, J., & Krajnc, A. (2010). Taxonomy of static code analysis tools. *33rd International Convention MIPRO*, 418-422.

Rahman, M. M., Paudel, R., & Sharker, M. H. (2019). Effects of Infusing Interactive and Collaborative Learning to Teach an Introductory Programming Course. *Frontiers in Education Conference (FIE)*, IEEE.

Rautenberg, W. (2010). *A concise introduction to mathematical logic.* Springer.

Rensink, A., & Zambon, E. (2009). A type graph model for Java programs. *Formal Techniques for Distributed Systems,* 237-242.

Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., & McConley, M. (2018). Automated vulnerability detection in source code using deep representation learning. *17th IEEE International Conference on Machine Learning and Applications (ICMLA),* 757-762.

Singh, R., Gulwani, S., & Solar-Lezama, A. (2013). Automated feedback generation for introductory programming assignments. *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 15-26.

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

Srikant, S., & Aggarwal, V. (2014). A system to grade computer programming skills using machine learning. *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 1887-1896.

Striewe, M. (2014). *Automated analysis of software artefacts-a use case in e-assessment (Doctoral dissertation).*

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

Takhar, R., & Aggarwal, V. (2019). Grading uncompilable programs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 9389-9396.

Vert, T., Krikun, T., & Glukhikh, M. (2013). Detection of incorrect pointer dereferences for C/C++ programs using static code analysis and logical inference. *Tools & Methods of Program Analysis*, 78-82.

Vetr`o, A. (2014). *Empirical assessment of the impact of using automatic static analysis on code quality (Ph.D. dissertation).*

Wang, T., Su, X., Wang, Y., & Ma, P. (2007). Semantic similarity-based grading of student programs. *Information and Software Technology*, 99-107.

Zhou, W., Pan, Y., Zhou, Y., & Sun, G. (2018). The framework of a new online judge system for programming education. *Proceedings of ACM Turing Celebration Conference-China*, 9-14.