ISSN 2029-7564 (online)
SOCIALINĖS TECHNOLOGIJOS
SOCIAL TECHNOLOGIES
2011, 1(2), p. 399–414.

AUTOMATED EVALUATION OF COMPUTER PROGRAMS AT UNDERGRADUATE LEVEL: SUITABILITY STUDY FOR COMPETITIVE EVENTS

Jūratė Skūpienė

Mykolas Romeris University, Lithuania, jurate.skupiene@mruni.eu

Abstract

Purpose—*The paper is aimed towards the problem of automated evaluation of computer programs designed by students during competitive events where the participants have to design an algorithm and to implement it as a working program, which has to be evaluated. A similar evaluation has to be performed at universities by the lecturers teaching basics of programming. The purpose of this paper is to investigate the automated evaluation applied to computer programs designed by the university students, and to analyze the suitability of such methods for evaluation of computer programs designed by high school students in competitive events.*

Design/methodology/approach—*Comparative analysis is the main approach used in this paper.*

Findings—*The educational and technical goals that seek evaluation of computer programs at the university level are different from those in informatics contests. Therefore the majority of approaches applied at universities have led to development of automated evaluation systems with functionalities different than the ones required by informatics contests, and in the opinion of the author cannot be easily transferred to informatics contests. We identified one area where such transfer might be possible: automated evaluation of programming style.*

Research limitations/implications—*The experience of automated evaluation of*

ISSN 2029-7564 (online)
https://www.mruni.eu/lt/mokslo_darbai/st/apie_leidini/
https://www.mruni.eu/en/mokslo_darbai/st/apie_leidini/index.php

programming style applied at the universities has a potential to be transferred to informatics contests; however the universities may ask their students to apply specific programming style, while informatics contests should accept a very broad range of reasonable programming styles. Additional investigation is required to answer the question in which way and to which extent the experience gained at the universities can be applied in informatics contests and the maturity exam.

***Practical implications**—This paper is a step towards developing a fair and motivated evaluation scheme in informatics contests. The paper would be useful for the researchers as it gives guidelines for future research.*

***Originality/Value**—It is very important to apply fair and motivated evaluation in informatics contests as they involve the majority of high-school students interested in computer science in Lithuania and some other countries as well and may motivate them to study computer science at the university. To the knowledge of the author, this is the first paper analyzing the suitability of automated evaluation methods developed at the university level, to be applied for the evaluation at high school level in particular in informatics contests. On the other hand, the evaluation in informatics contests is an issue that attracts the interest of international community of informatics contests.*

Keywords: *automated evaluation, programming assignments, programming courses, programming style, automated evaluation systems, informatics contests.*

***Research type*:** *literature review.*

1. Introduction

There are many co-curricular activities in which the motivated high school students are involved in competitive learning. Informatics contests are introduced as the fastest expanding co-curricular activity related to computer science which is seen as a good model of competitive learning (Revilla et al, 2008). They are the contests of algorithmic problem solving. The contestants are given an algorithmic problem and have to design an algorithm, to implement it, and submit as a working program. The proof of the algorithm correctness and efficiency is not required. One of the reasons is that presenting a correctness proof to such problems is too difficult for the contestants who are still at high school.

Thus we arrive at the concept of an algorithm-code complex. The term stands for a program which contains an implementation of an unknown algorithm designed to solve the given task. An algorithm-code complex combines the outcome of both an algorithm design and program development. As a result, it becomes hard to evaluate characteristics of the algorithm, and of its implementation in the algorithm-code complex, because it is hard to separate them and identify whether a feature of an algorithm-code complex belongs to an algorithm or to its implementation. Thus, evaluation of qualities of an

algorithm-code complex in informatics contests becomes an interesting scientific problem.

The current practice of many such contests is that the dominant part of evaluation (i.e. deciding about properties and qualities of implemented algorithm and the implementation in a form of scores) is based on empirical black-box testing of algorithm-code complex. Task designers have certain expectations about the relationship of the quality and characteristics of solutions to the measure of those qualities expressed in points.

However, the essence of black-box testing is that no knowledge of key ideas of algorithm, internal logic and code structure is revealed. Therefore, the conclusions about the qualities of the algorithm-code complexes made in the form of assigned scores raise various educational and scientific questions.

Similar educational situation is often encountered at the universities providing introductory programming courses. Originally automated evaluation was developed at universities for evaluating submissions to programming assignments given in the programming courses, and a lot of research is designated to that. We use the term *programming assignment* when we refer to the tasks given in the programming courses to make distinction from the tasks given in informatics contests. Typically, static and/or dynamic evaluation is performed to evaluate programs which are given as an assignment or an exam task in undergraduate courses of computing education.

Technically both in informatics contests and in the programming courses we deal with an algorithm presented in the form of implementation. Therefore analyzing the experience of the universities might assist in developing a fair grading in informatics contests.

There have been published many papers about different aspects of evaluation in informatics contests (Cormack et al., 2006; Pohl, 2006; Skienna and Revilla, 2003; Trotman and Handley, 2006; Vasiga et al., 2008; Verhoeff, 2009). In this paper we will overview the finding and trends in the automated evaluation of programming assignments at undergraduate level with special emphasis on the suitability of the applied practices for informatics contests.

2. Development of Automated Evaluation of Programming Assignments

The roots of necessity to the automated evaluation are similar both in informatics contests and in programming courses. Programming problems and assignments are considered essential elements of software engineering and computer science education courses (Douce et al., 2005). Academic institutions face the challenge of providing their students with a better teaching quality. Simultaneously they need to decrease the amount of additional work for the staff. As a consequence of that, huge numbers of programming assignments (programs) have to be evaluated and provided with feedback in a short period of time (Joy and Luck, 1995).

The schedule is even tighter in informatics contests. It can be estimated that over a thousand submissions have to be evaluated in a few hours in IOI. Therefore at present it is considered that there is no alternative to automated evaluation neither in informatics contests, nor in programming courses (Kemkes et al., 2006).

Automated evaluation tools of programming assignments share common features like speed, consistency, and availability of evaluation (Ala-Mutka, 2005). A comprehensive overview of automated evaluation systems was presented in (Colton et al., 2006; Douce et al., 2005).

The earliest examples of the automated evaluation system can be found (Hollingsworth, 1960). The next step was a system, where automated evaluation was applied in testing beginner's student programs written in Algol. Routines had to be written for each task. The tests were randomly generated and the programs were executed with those tests. The output was checked for correctness (Forsythe and Wirth, 1965). New ideas were introduced in (Hext and Winings, 1969). This system could already compile and run programs without a human intervention and it was testing each program with two tests. It had implemented the scoring policy, i.e., assigned points for a successful compilation, a short running time, etc. Those earliest first generation automated evaluation systems already had the most important features and demonstrated the power of automated evaluation. Using automated evaluation tools of the first generation required a certain qualification and experience.

Automated evaluation systems of the second generation were tool-oriented systems (Douce et al., 2005). They were developed using existing tools. The focus of such systems was the same as in the earlier systems, i.e., functional correctness of submitted programs. Some second generation systems already had already implemented a remote submission and use of a network (Benford et al., 1995; von Matt, 1994). The automated evaluation systems started to support grading with generation of grading reports that allow the tutors to assign the weights to the tests.

The third generation automated evaluation systems can be called web-oriented tools. They use web-technology, adopt more sophisticated testing approaches (e.g. "diagram" evaluation in *CourseMarker* (Cou, 2010)), support many programming languages, automatically evaluate the program design, provide a rich feedback for the student, introduce plagiarism detection, etc. (Douce et al., 2005).

Note that there were many informal grading systems developed, where it is difficult to transfer the results among institutions and even among course instructors (Edwards, 2003).

Development of automated evaluation in informatics contests has some parallels. We have not found a published overview of the development of evaluation systems in informatics contests. However we observed the appearance of tool-oriented evaluation systems and their development into web-based CMS (Contest Management System) in LitIO and IOI (Skupiene, 2004). Earlier systems were more limited technically. For example, they did not provide real-time feedback during the contest. The contestants had no aid in detecting errors related with format specification (e.g., the wrong file

name or extra space at the end of the line). As a result, those errors had more weight in informatics contests than they were supposed to.

Modern web-based contest management systems (IOI, 2002; Mareš, 2007) for managing informatics contests are supplied with many features like real-time feedback during the contest, contest management features, analysis mode after the contest, etc. They have improved the quality of contests in many aspects. However, the main concern of applying black-box testing to the evaluation in informatics contests (i.e., detecting all incorrect submissions and validity of assigning partial scores to such submissions) remains.

3. New Role of Automated Evaluation Tools in Programming Courses

In recent years the role of automated evaluation tools in computer science and programming courses has changed significantly.

Looking at the history of automated evaluation in programming courses in universities, we observe a shift of emphasis. The ability to automatically compile, run, and test a student's program and provide the score was most important in the early systems. These remain important issues both for the contests and for the programming courses. However, the emphasis was shifted in different directions in the informatics contests and in the evaluation of programming courses.

A course on a subject (programming) is a lengthy process which involves many assignments, submissions and resubmissions, deadlines, observation of student's performance, progress, and feedback from the evaluator. The grading tool components that support the course management became important (Benson, 1985). Even though the main role remains measuring student knowledge and skills, the role of a grading tool as a learning device became very significant. The students need supporting learning (and evaluation) environments, because the learning environments help achieve better learning outcomes (Roberts and Verbyla, 2002). Designing a course and comfortable monitoring of the learning and evaluation process became important features of learning environments and tools.

Automated evaluation systems try to solve a number of other issues that are outside the scope of informatics contests, for example, plagiarism detection, evaluating programs with graphical interfaces, performing the formative assessment of programming assignments, evaluating the automated programming assessment with respect to the stated objectives, student's knowledge of language constructions, analysing the program structure in order to identify whether the program followed the given skeleton, supporting different types of assignments, parameterising programming problems, and peer-assisted automated evaluation. These other issues were discussed in (Amelung et al., 2006; Benson, 1985; Carter et al., 2003; Douce et al., 2005; Lewis and Davies, 2004; Malmi et al., 2002; Pardo, 2002; Saikkonen et al., 2001; Woit and Mason, 1998).

The new roles require additional features of the environments and they have become an active topic of research in the area of computing science education. On the contrary, that did not become an active topic in informatics contests and they are not relevant in the context of this research.

The CMS does not have to perform the role of a learning tool in informatics contests. Some other features (e.g., evaluating programs with graphics) might become relevant if the format of the contest were different. The attitude towards the support and feedback for the contestants is different. The contestants are provided support from the CMS on the issues that might distract them from concentrating on the algorithm. For example, the CMS detects output formatting errors or provides run-time information. Providing too much feedback might conflict with the nature of the contest as an event. With the growing speed of processors, precise measurement of program execution time becomes highly important in informatics contests as this is directly related to the ability of the system to distinguish between different efficiency classes of solution. Advertising the contest (e.g. the ability to demonstrate the scores of the contestants on a live score board for the spectators during the contest) is another new expected feature of CMS.

Here we have shown that the research of automated evaluation in computing education is relevant and active. However, the direction of the research is different from that in informatics contests (Fig. 1). The number of recent publications directly related to the evaluation of programming assignments is very limited. That was also mentioned (Ala-Mutka, 2005). Next we will look over the automated evaluation experience in programming courses.

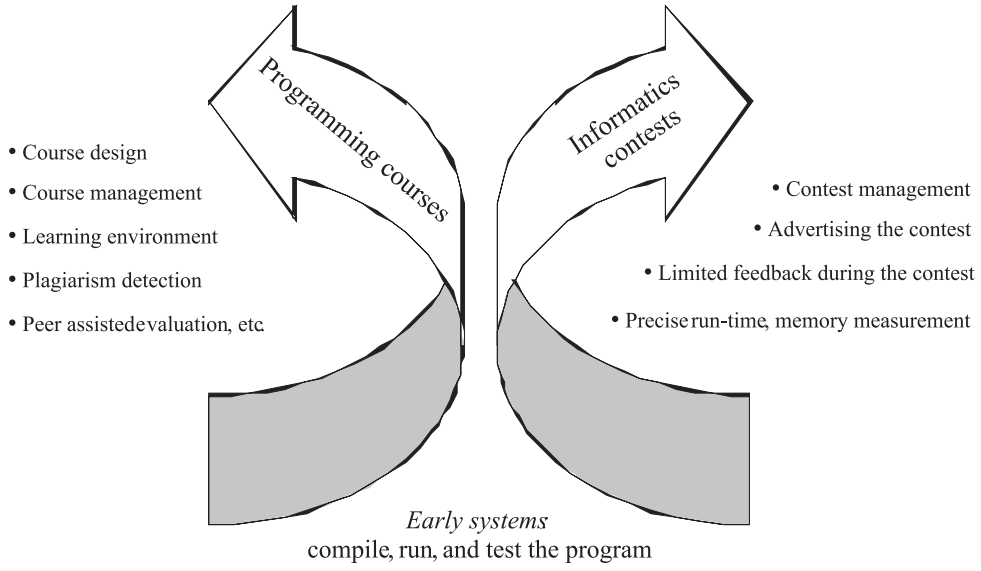

Figure. 1. The directions of development of automated evaluation in programming courses and in informatics contests

4. Evaluating Programming Assignments by Testing

In this chapter we will look through some aspects of applying black-box testing in the evaluation of programming assignments. We located just a few sources and the most extensive reference is (Ala-Mutka, 2005).

We did not find any extensive discussions in the published papers based on the fact that testing cannot be used to prove the program correctness (in our case the algorithm-code complex correctness) (Dijkstra, 1972).

We suggest that one of the reasons is the difference in the difficulty of tasks. The tasks at high level informatics contests might be a real challenge even for graduates of computer science studies. Therefore heuristic approaches are common among the submissions of contestants. They are incorrect, and it is rather difficult to detect all of them by black-box testing (Verhoeff, 2006).

The situation is different with the course assignments. Much research was devoted to the automated evaluation of introductory programming assignments (Califf and Goodwin, 2002), which are much easier if compared to the contest tasks. The assignments have to reflect the syllabus and should be solvable after taking the course.

Despite the inability to prove program correctness, testing still can show the absence of known errors (Leal and Moreira, 1998). For simple assignments (e.g. sorting an array) that are routinely given to the students, it is much easier to decide on the known errors and make tests against them. This might be the reason why we did not find discussions about the ability of black-box testing to detect errors in programming assignments.

In the informatics contests each task is (expected to be) original, requires problem solving skills and more complicated techniques. Therefore the concept of known errors remains rather vague in the informatics contests.

Assigning the score to incorrect solutions is a questionable issue both in evaluating the programming assignments and submissions (Verhoeff, 2006). The concept how close the algorithm-code complex is to the correct solution is a subjective judgement. The subjectivity arises either from the human grader or from the nature of black-box testing. In general, the black-box testing does not expose neither the nature, nor the scope of error). The practice of applying all-or-nothing scoring with a possibility of resubmission is acceptable for regular programming assignments (Colton et al., 2006). In the case of failure, the students might be given the failed test and have to fix their solutions. This is the essential difference from the practice of informatics contests. In the contests, the test set is fixed before the contest and the same set is applied to every submission to ensure the same testing conditions. While in the evaluation of programming courses, it is usual to apply randomly generated tests for evaluation and different students might be tested by different sets of tests (Colton et al., 2006). Such a practice is not directly applicable in the informatics contests.

We have found one more interesting approach that correlates with informatics contests. It deals with measuring the solution complexity (efficiency). This is the experience of assessing individual procedures rather than programs. The automated evaluation system *Scheme-robo* was developed and the experience of assessing simple

assignments in introductory programming courses was presented (Saikkonen et al.,2001). (Hansen and Ruuska, 2003) have implemented it by giving the students an input/output module for the assignments that concentrate on efficient data processing algorithms. Calculating the number of times, certain structures inside the program were executed, and comparing the results to model the solution was implemented on *CourseMaster* and *Assyst* systems (*Foxley et al., 2004; Jackson and Usher, 1997*).

A similar suggestion to use such a metric in the informatics contests was presented by (Ribeiro and Guerreiro, 2009). They address the difficulties related to measuring the efficiency. As the computer power increases, the size of input has to increase in order to separate the solutions of different complexity. Data increase causes other problems. Measuring behaviour of some structures within the program might be a solution in this case. The paper suggests asking to submit functions (procedures) rather than programs and repeating the same function call several times to increase clock precision. Thus input size, which nowadays has become too large and started causing problems, is decreased. Curve fitting analysis is proposed to be used to estimate program complexity rather than referring to the number of passed test cases. However experiments and the corresponding software are required before the proposal can be included into the evaluation scheme.

We presented a few examples of similar issues which occur in both contexts. On the one hand, this shows that concerns about black-box testing are not so active and severe in the evaluation of programming assignments. We did not discover the experience that could be directly transferred to informatics contests. The suggested different measurement of the algorithm-code complex efficiency is interesting and potentially applicable in informatics contests.

5. Automated Evaluation of Programming Style

From the observations in the previous chapters we have concluded that much of research in the area of automated evaluation in the programming courses is outside the interest of evaluation in the informatics contests. However, we have found an area where the experience of automated evaluation in the mass programming courses might be transferred to the informatics contests.

It is the automated evaluation of the quality of program design. This involves performing a static analysis and checking the program source against a set of characteristics. Many grading tools were designed that perform a static analysis and use software metrics to check readability, maintainability, and complexity of the source code (Ala-Mutka et al., 2004; Hirch and Heines, 2005; Jackson, 2000; Leal and Moreira, 1998; Spacco et al., 2005). Such tools were applied in evaluating programming assignments in universities. However, we found no evidence of such automation being applied in informatics contests.

The ability to write nice and elegant programs is already a skill and a very important skill which is not usually the focus of computer science and programming courses (Kernighan and Pike, 1999). It is easy to make a small program working despite a bad

style. Students often treat the programming style as secondary, not part of the program development process (Schorsch, 1995). That was also noticed both in the context of programming courses and in the context of informatics contests (Douce et al., 2005; Grigas, 1995; Struble, 1991).

In order to measure the programming style, we need a common understanding of programming style. "*A programming style is understood as an individual's interpretation of a set of rules and their application to the writing of source code in order to achieve the aim*" (Mohan and Gold, 2004) that the source code is readable and understandable. It can be said that everything that is related to program clarity, simplicity and generality, is understood as programming style. These types of definitions together with some guidelines can be applied for holistic approach to evaluate the programming style by human evaluators.

However, in order to introduce the automated evaluation, the elements of the style should be identified and concrete metrics for each element must be defined (Fig. 2) and associated with ranges of the expected values (Hirch and Heines, 2005).

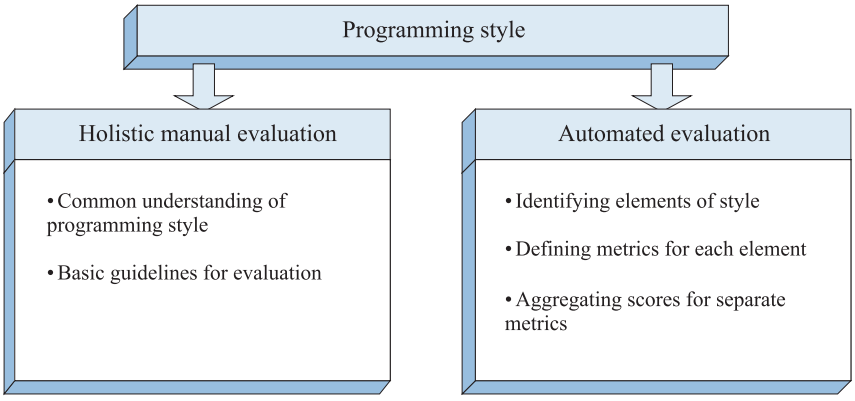

Figure 2. Holistic versus automated evaluation of programming style

The first tools for the automated evaluation of programming style were created in the eighties. Then the basic guidelines for the programming style were created. One of the early systems was a *Style* system (Rees, 1982) for automated evaluation of the programming style of Pascal programs. The system had ten style measures that could be easily calculated. They were: average line length, percentage of comment lines, numbers of *goto*'s, average length of identifiers, use of blank lines as separators, etc. The scoring scheme included five parameters for each metric which defined the conversion curve presented in Fig. 3. The parameter max specifies the maximum score to be awarded for each measure. If the value of the measure lies in the range defined by *lotol* and *hitol* then the maximum score for that measure is given. If the value of the measure is lower than lo or higher than hi, then the score is zero. For example, it the percentage of comment lines in the source is either very low (basically no comments) or very high (nearly every line

is commented) then the score for this measure would be very low. To get a maximum score the amount of comments should be within some reasonable range.

The total score was an aggregate of the scores of separate metrics.

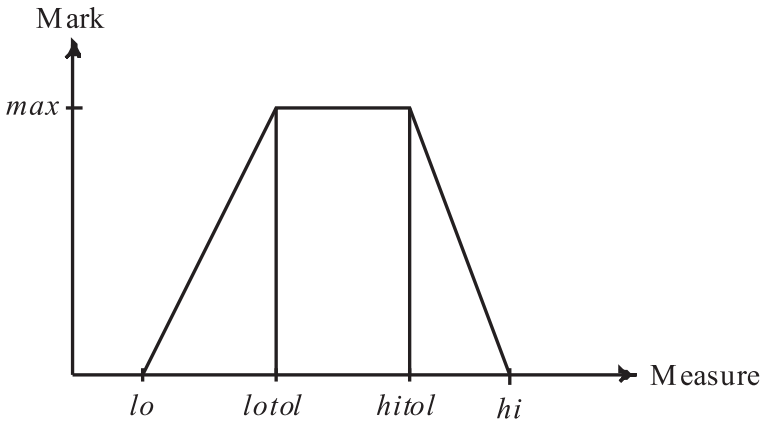

Figure 3. Programming style marking scheme suggested by (Rees, 1982)

max—maximum score for the programming style. If the value of a concrete style measure is lower that *lo* or higher than *hi* then the score is zero

Later on, a C analyser was developed for evaluating the programming style. It served as a basis for developing other automated evaluation tools (Benford et al., 1995; Berry and Meekings, 1985). The feature of those systems was that the course designers could configure the parameter values for the metrics. The new tools foresee other programming languages (Jackson and Usher, 1997; Leal and Moreira, 1998; Redish and Smyth, 1986) and more metrics. For example, (Dromey, 1995) incorporated 99 metrics for automated evaluation of C programs. A variety of tools and the increasing number of metrics resulted in the classification of measurements and developing taxonomy for the programming style (Oman and Cook, 1990). The taxonomy proposed four stylistic factors. These were: general programming practices, typographic style, control structure style, and information structure style.

Among later systems we could mention *Checkstyle* for automated evaluation of Java programs (Burn, 2003). This is an open source tool that provides an extensive analysis of the source code programming style. The feature of this tool is its modularity. The *Checkstyle* consists of a variety of checks and additional checks can be written to include new metrics.

Earlier available systems either did not cover important features of object-oriented programming or used some obsolete checking which is currently performed by compilers. Therefore a STYLE++ tool has been created for automated evaluation of the programming style of C++ programs (Ala-Mutka et al., 2004). The tool covers 64 different measures. Metrics were developed that meet the software quality requirements.

They also included non-functional quality requirements, such as reliability and efficiency. Four programming style categories have been introduced: transportability, understandability, modifiability, and readability. They were decomposed into nine smaller categories until measurable features of a concrete level have been reached. Scoring is based on the ideas of (Rees, 1982), however, since different courses may require emphasis of different style aspects, the system allows much tailoring, irrelevant measures may be switched off and different weights might be associated with different measures.

Program documentation (which includes proper commenting) can be considered as a separate part of programming style. We discovered efforts to create an automated evaluation tool for evaluating the quality of program code documentation. Even though currently there are no guidelines (and no measurable standards) how to perform such an evaluation, there exist tools that help creating such a documentation. Given those tools students, should be required to produce a qualitative documentation (Hirch and Heines, 2005).

The interest in the automated programming style has lowered if compared to the eighties. The efforts to find modern program style development tools or programming style evaluation guidelines for C++ evaluation for educational purposes were unsuccessful (Ala-Mutka et al., 2004). The accessible guidelines are industrial high-level recommendations for object-oriented program design.

There is one significant difference between the evaluation of programming style in the programming courses and that in the informatics contests. Universities sometimes develop their own standards, they might ask the students to follow some specific programming style standards, while the informatics contests should be open to a variety of programming styles. The contestants do want precision when they deal with getting or loosing points (Grigas, 1995). To ensure equal conditions for the contestants, the evaluation of programming style should be language independent.

Note that evaluation of the quality of program design is supported not by all educators. Design is important if the program works. There is an opinion that, once the students have learned to program, it is easy to teach them good design, but not vice versa (Daly and Waldron, 2004).

It can be concluded that much research has been done in the area of evaluating programming style of programming assignments in the programming courses, and we found no evidence of any of that being applied in the informatics contests. The main idea of the automated evaluation of programming style is performing the static analysis, calculating different metrics, and associating the expected ranges. We feel that this experience can be transferred to the informatics contests. In order to apply the experience in the informatics contests, the research should be conducted in two areas. Metrics should be chosen and tailored so that they would not favour some programming styles and disfavour the others. Another direction of research is to ensure the evaluation compatibility between different programming languages.

6. Conclusions

The automated evaluation of programming assignments is not a new area in computing education research. Much research on the evaluation of programming assignments during the programming courses has been conducted and published. However, the research and development of tools of automated evaluation of programming assignments are moving into a direction different than that of the informatics contests. We suggest two main reasons. One reason is that informatics contests are not part of any successive learning process, therefore many aspects relevant in the learning process are not valid in the informatics contests. The second reason is nature and difficulty of tasks. Programming assignments should be coherent with course curricula while tasks in informatics contests are problem solving tasks covering broad range of topics. Therefore some approaches (like concept of *known errors*) that can be applied for evaluation of completed programming assignments cannot be applied in informatics contests.

An exception is automated evaluation of programming style. Many tools were created for this purpose and much research has been pursued on applying such tools to the programming courses. We see the potential that this experience might be transferred to informatics contests especially at a national level. However, it requires an extensive separate study to ensure that automated evaluation of programming styles does not favour some programming styles and disfavour the others.

Literature

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

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*, 3:245–262.

Amelung, M., Piotrowski, M., and Rosner, D. 2006. EduComponents: experiences in e-assessment in computer science education. *ACM SIGCSE Bulletin*, 38(3):1–5.

Benford, S. D., Burke, E. K., Foxley, E., and Higgins, C. A. 1995. The Ceilidh system for the automatic grading of students on programming courses. *In Proceedings of the 33rd annual on Southeast regional conference*, p. 176–182.

Benson, M. 1985. Machine assisted marking of programming assignments. *ACM SIGCSE Bulletin*, 17(3):24–25.

Berry, R. E. and Meekings, B. A. E. 1985. A style analysis of C programs. *Communications of the ACM*, 28(1):80–88.

Burn, O. 2003. CheckStyle. SourceForge.net. [interactive] [accessed 27-12-2011] <http://checkstyle.sourceforge.net/>.

Califf, M. E. and Goodwin, M. 2002. Testing skills and knowledge: introducing a laboratory exam in CS1. *ACM SIGCSE Bulletin*, 34(1):217–221.

Carter, J., Ala-Mutka, K., Fuller, U., Dick, M., English, J., Fone, W., and Sheard, J. 2003. How shall we assess this? *ACM SIGCSE Bulletin*, 35(4):107–123.

Colton, D., Fife, L., and Thompson, A. (2006). A web-based automatic program grader. *In*

Proceedings of ISECON, the Conference for Information Systems Educators, v. 23, p. 1–9, Dallas, USA.

Cormack, G., Kemkes, G., Munro, I., and Vasiga, T. 2006. Structure, scoring and purpose of computing competitions. *Informatics in Education*, 5(1):15–36.

CourseMarker. Automatic Marking and Feedback for Students and Teachers. [interactive] School of Computer Science and IT, The University of Nottingham, UK. 2010 [accessed 27-12-2011] <http://www.cs.nott. ac. uk/~cmp/cm_com/index.html>.

Daly, C. and Waldron, J. 2004. Assessing the assessment of programming ability. *In Proceedings of the 35th SIGCSE technical symposium on Computer science education*, p. 210– 213.

Dijkstra, E. W. 1972. The humble programmer. *Communications of the ACM, ACM Turing Lecture*, 15(10):859–866.

Douce, C., Livingstone, D., and Orwell, J. 2005. Automatic test-based assessment of programming: A review. *ACM Journal of Educational Resources in Computing*, 5(3):1–13.

Dromey, R. G. 1995. A model for software product quality. *IEEE Transactions on Software Engineering*, 21:146–162.

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*, p. 148–155, Anaheim, CA, USA.

Forsythe, G. E. and Wirth, N. 1965. Automatic grading programs. *Communications of the ACM*, 8(5):275–278.

Grigas, G. 1995. Investigation of the relationship between program correctness and programming style. *Informatica*, 6(3):265–276.

Hansen, H. and Ruuska, M. 2003. Assessing time efficiency in a course on data structures and algorithms. *In Koli Calling 2003, 3rd Annual Finnish/Baltic Sea Conference on Computer Science Education*, p. 86–93.

Hext, J. B. and Winings, J. W. 1969. An automatic grading scheme for simple programming exercises. *Communications of the ACM*, 12(5):272–275.

Hirch, B. and Heines, J. M. 2005. Automated evaluation of source code documentation: Interim report. *In Proceedings of the 36th SIGCSE technical symposium on Computer science education*, p. 1–5, S. Louis, Missouri, USA.

IOI'2002 manual. [interactive] South Korea 2002, [accessed 27-12-2011] <http://www. ioi2002. or.kr/eng/PracticeCompetitionMaterial/ ContestSystemManual.pdf>.

Hollingsworth, J. 1960. Automatic graders for programming classes. *Communications of the ACM*, 3(10):528–529.

Jackson, D. 2000. A semi-automated approach to online assessment. *In Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, p. 164–167, Helsinki, Finland.

Jackson, D. and Usher, M. (1997). Grading student programs using ASSYST. *ACM SIGCSE Bulletin*, 29(1):335–339.

Joy, M. and Luck, M. 1995. On-line submission and testing of programming assignments. *In Innovations in Computing Teaching*, SEDA, London.

Kemkes, G., Vasiga, T., and Cormack, G. 2006. Objective scoring for computing competition tasks. *In Proceedings of International Conference in Informatics in Secondary Schools –Evolution and Perspectives, Lecture Notes in Computer Science*, p. 230–241. Springer-Verlag.

Kernighan, B. W. and Pike, R. 1999. *The Practice of Programming*. Addison-Wesley.

Leal, P. J. and Moreira, N. 1998. Automatic grading of programming exercises. [interactive] Technical report series: Dcc-98-4, University of Porto, Portugal, Department of Computer Science. [accessed 27-12-2011] <www.ncc.up.pt/~nam/ publica/dcc-98-4. ps.gz>.

Lewis, S. and Davies, P. 2004. The automated peer-assisted assessment of programming skills. *In Proceedings of the 8th JAVA & The Internet in the Computing Curriculum Conference JICC8*, p. 1–10.

Malmi, L., Korhonen, A., and Saikkonen., R. 2002. Experiences in automatic assessment on mass courses and issues for designing virtual courses. *ACM SIGCSE Bulletin*, 34(3):55–59.

Mareš, M. 2007. Perspectives on grading systems. *Olympiads in Informatics*, 1:124–130.

Mohan, A. and Gold, N. (2004). Programming style changes in evolving source code. *In IEEE Proceedings of the 12th International Workshop on Program Comprehension*, p. 236– 240, Italy.

Oman, P. and Cook, C. 1990. A taxonomy for programming style. *In 18th ACM Computer Science Conference Proceedings*, p. 244–250.

Pardo, A. 2002. A multi-agent platform for automatic assignment management. *ACM SIGCSE Bulletin*, 34(3):60–64.

Pohl, W. 2006. Computer science contests for secondary school students: Approaches for classification. *Informatics in Education*, 5(1):125–132.

Redish, K. A. and Smyth, W. F. 1986. Program style analysis: a natural by-product of program compilation. *Communications of the ACM*, 29(2):126–133.

Rees, M. J. 1982. Automatic assessment aids for Pascal programs. *ACM SIGPLAN Notices*, 17(10):33–42.

Revilla, M. A., Manzoor, S., and Liu, R. 2008. Competitive learning in informatics: The Uva Online Judge experience. *Olympiads in Informatics*, 2:131–148.

Ribeiro, P. and Guerreiro, P. 2009. Improving the automatic evaluation of problem solutions in programming contests. *Olympiads in Informatics*, 3:132–143.

Roberts, G. B. and Verbyla, J. L. M. 2002. An online programming assessment tool. *In Proceedings of Australasian Computing Education Conference (ACE2003), volume 20 of Conferences in Research and Practice in Information Technology*, p. 69–75, Adelaide, Australia.

Saikkonen, R., Malmi, L., and Korhonen, A. 2001. Fully automatic assessment of programming exercises. *In Proceedings of the 6th annual conference on Innovation and technology in computer science education*, p. 133–136, Canterbury, United Kingdom.

Schorsch, T. 1995. CAP: An automatic selfassessment tool to check pascal programs for syntax, logic and style errors. *In Proceedings of the 26th SIGCSE technical symposium on Computer science education*, p. 168–172, USA.

Skienna, S. and Revilla, M. 2003. *Programming Challenges—the Programming Contest Training Manual*. Springer-Verlag, New York.

Skupiene, J. (2004). Testing in informatics olympiads (in Lithuanian). *In Information Technologies Conference Proceedings*, p. 37–41, Kaunas. Technologija.

Spacco, J., Strecker, J., Hovemeyer, D., and Pugh, W. 2005. Software repository mining with Marmoset: An automated programming project snapshot and testing system. *In Proceedings of the 2005 international workshop on Mining software repositories*, p. 1–5.

Struble, G. 1991. Experience hosting a high school level programming contest. *ACM SIGCSE Bulletin*, 23(2):36–38.

Trotman, A. and Handley, C. 2006. Programming contest strategy. *Computers & Education*, 50(6):821–837.

Vasiga, T., Cormack, G., and Kemkes, G. 2008. What do olympiads tasks measure? *Olympiads in Infomatics*, 2:181–191.

Verhoeff, T. 2009. 20 years of IOI competition tasks. *Olympiads in Informatics*, 3:149–166.

Verhoeff, T. 2006. The IOI is (not) a science olympiad. *Informatics in Education*, 5(1):147–158.

von Matt, U. 1994. Kassandra: the automatic grading system. *ACM SIGCUE Outlook*, 22(1):26–40.

Woit, D. M. and Mason, D. V. 1998. Lessons from on-line programming examinations. *In Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education: Changing the delivery of computer science education*, p. 257–259, Ireland. Dublin City University.

PROGRAMAVIMO UŽDAVINIŲ AUTOMATINIO VERTINIMO GALIMYBĖS IR TAIKYMAS PROGRAMAVIMO VARŽYBOSE

Jūratė Skūpienė

Mykolo Romerio universitetas, Lietuva, jurate.skupiene@mruni.eu

Santrauka. *Automatinis studentų sukurtų programų vertinimas yra išsamiai analizuojamas moksliniuose straipsniuose jau daugelį metų. Dėstytojams pateikus pradinių kursų studentams programavimo užduotis tenka įvertinti gautas programas. Dažnai vertinamų programų skaičius didelis, todėl taikomas automatizuotas vertinimas. Su analogiška situacija susiduriama programavimo varžybose. Programavimo varžybos yra algoritminių uždavinių problemų sprendimo varžybos, kuriose dalyviai turi sukurti algoritmą, sprendžiantį duotąjį uždavinį, ir jį realizuoti veikiančia programa bet kuria iš varžybose numatytų programavimo kalbų bei pateikti savo darbą įvertinti.*

Šio straipsnio tikslas – išanalizuoti automatinio programavimo užduočių vertinimo patirtį aukštosiose mokyklose ir įvertinti šios patirties tinkamumą ir perkeliamumą į programavimo varžybas.

Edukaciniai bei techniniai (suponuoti edukacinių tikslų) studentų sukurtų programų vertinimo tikslai labai skiriasi nuo vertinimo tikslų programavimo varžybose. Todėl universitetuose taikomi vertinimo būdai bei automatizuoto vertinimo poreikiai sąlygojo atitinkamų automatinių vertinimo sistemų sukūrimą. Tačiau šių sistemų funkcionalumas labai skiriasi nuo funkcionalumo, tinkamo informatikos varžyboms, todėl, autorės nuomone, automatizuoto studentų sukurtų programų vertinimo patirtį nėra tikslinga perkelti į programavimo varžybas. Straipsnyje išskirta viena sritis (automatizuotas programavimo stiliaus vertinimas), kurioje universitetų patirtis gali būti potencialiai pritaikyta varžybose.

Automatizuotas programavimo stiliaus vertinimas, taikomas kai kuriose aukštosiose mokyklose, gali numatyti konkretų programavimo stilių, kurio studentai turėtų laikytis. Reikalavimas naudoti vienodą stilių palengvina automatizuotą vertinimą. Tuo tarpu programavimo varžybose skirtingi programavimo stiliai turėtų būti laikomi vienodai priimtinais ir automatizuotas vertinimas neturėtų suteikti pranašumo jokiam konkrečiam stiliui ar jų grupei. Tad, norint perkelti automatizuoto programavimo stiliaus vertinimo patirtį į programavimo varžybas, reikalingas atskiras tyrimas, kuris atsakytų į šiuos klausimus.

Šis straipsnis – tai tolesnis žingsnis siekiant sukurti pagrįstą ir motyvuotą programavimo varžybų vertinimo schemą. Straipsnyje pateikiamos konkrečios tolesnių tyrimų gairės, tad jis naudingas mokslininkams, atliekantiems šios tematikos tyrimus.

Programavimo varžybose dalyvauja daug informatika (kompiuterių mokslu) besidominčių vyresniųjų klasių mokinių, ir dalyvavimas varžybose gali turėti įtakos jų pasirinkimui studijuoti informatiką. Yra populiaru organizuoti varžybas siekiant pritraukti potencialius studentus. Todėl itin svarbu, kad varžybose programų vertinimas būtų motyvuotas, pagrįstas ir suprantamas dalyviams, o kartu skatinantis tobulėti. Autorės žiniomis, šis straipsnis yra pirmasis, nagrinėjantis automatizuoto atliktų programavimo užduočių vertinimo, taikomo aukštosiose mokyklose, tinkamumą programavimo varžybose.

Raktažodžiai: *automatinis vertinimas, programavimo užduotys, programavimo mokymas, programavimo stilius, automatinės vertinimo sistemos, programavimo varžybos.*