Hayashi, Y., et al. (Eds.) (2017). Workshop Proceedings of the 25th International Conference on Computers in Education. New Zealand: Asia-Pacific Society for Computers in Education

Adoption of Computer Programming Exercises for Automatic Assessment — Issues and Caution

Yuen Tak YU[a*], Chung Man TANG[b], Chung Keung POON[b] & Jacky Wai KEUNG[a]

[a]*Department of Computer Science, City University of Hong Kong, Hong Kong*
[b]*School of Computing and Information Sciences, Caritas Institute of Higher Education, Hong Kong*
*csytyu@cityu.edu.hk

Abstract: Computational thinking is an interdisciplinary core skill to be acquired in STEM education, while computer program coding is a concrete manifestation of such a skill. In response to the increasing size of computer programming classes and rapidly growing number of learners, particularly in massive open online courses (MOOCs), many instructors nowadays heavily rely on the use of automated systems to assess the programming work of students. However, these automated assessment systems typically perform black box testing to determine the correctness of student programs, which limits the type of programming exercises that can be automatically assessed. This paper reports a case study on the adoption of programming exercises from textbook and online resources, and categorises some difficulties and issues of caution due to the technical limitation of typical automated assessment systems. The identified issues are mainly related to the input/output and non-deterministic nature of the programs or the intended learning outcomes of some of the exercises. The paper concludes with a brief outline of recent research directions to alleviate these problems for improvement of learning.

Keywords: Assessment of learning, automated assessment, black box testing, computational thinking, computer programming exercises, technology-enhanced learning and assessment

1. Introduction

Computational thinking is an interdisciplinary core skill to be acquired in STEM education with relevance across all four disciplines of science, technology, engineering and mathematics, while computer programming (or program coding) is a concrete manifestation of such a skill. Computer programming is now taught not only at tertiary and senior secondary levels of education, but increasingly at junior high schools or even primary schools. It is also a core subject of almost all STEM-related majors spanning across different faculties and schools in most universities, and is now a popular subject in general education for students from non-STEM majors. It is not unusual to have hundreds of students attending a computer programming course at the same time (Wang & Wong, 2007), particularly in massive open online courses (MOOCs) (Staubitz et al., 2015; Thiébaut, 2015).

Although there are many other ways to assess students' knowledge of computer programming such as the use of multiple choice or conventional questions (Siddiqi, Harrison, & Siddiqi, 2010), exercises that require students to write programs are much more common and relevant. Hence, to assess students' learning, it is necessary for instructors to perform the following tasks (Chong & Choy, 2004):

(T1) design and select many programming exercises as practice and for assessment,

(T2) administer the dissemination of exercises and collection of students' program submissions, and

(T3) assess the submitted programs and provide feedback to students.

As class sizes grow, it is increasingly impracticable for the instructor to manually administer the exercises, collect and assess every student's solution, and provide prompt and informative feedback to students. Many instructors nowadays heavily rely on automated program assessment systems (APASs) which not only alleviate their workload, but also significantly raise the motivation of students (Law, Lee, & Yu, 2010) and enhance their educational experience in a technology-enhanced hybrid learning and assessment environment (Chong & Choy, 2004; Ala-Mutka, 2005; Wang & Wong, 2007). Thus, APASs have been effective in assisting instructors in the tasks (T2) and (T3) listed above. This paper focuses on the effect of using APASs on task (T1).

Some examples of earlier APASs documented in the literature include BOSS (Joy, Griffiths, & Royatt, 2005), CourseMarker (Higgins et al., 2003) and PASS (Yu, Choy, & Poon, 2006), while more APASs have been reported since (Ihantola et al., 2010). An APAS typically performs black box testing to determine the correctness of the program outputs. Past anecdotal observations have suggested that such APASs may constrain the type of programming exercises amenable to adoption (Jackson, 1991; English, 2004; Yu & Tang, 2012). To better understand the extent of these effects, this paper reports a case study on the adoption of computer programming exercises from textbook and online resources, categorises some difficulties and issues of caution due to the technical limitation of typical APASs, and concludes with a brief outline of recent research directions to alleviate these problems.

2. Considerations in the Adoption of Programming Exercises

Designing good exercises for the assessment of learning is far from trivial. The instructor has to consider many such pedagogical factors as (1) the level of difficulty, (2) the nature of the programming constructs involved, (3) the intended learning outcomes being assessed, (4) the specific skills which the instructor wants the students to drill and practice, (5) the extent that the exercise can generate interests for students to work on and hence stimulate intrinsic motivation for their learning, and so on.

Since students need to do a lot of practice, there is a heavy demand for a large number of good programming exercises. While many instructors do compile custom-designed programming problems with their own ideas, they may still have to seek additional resources when the number of exercises in demand is large. Textbooks and online courses usually have a variety of practical exercises that are systematically categorised into different topics of a course, levels of complexity or difficulty, or according to the specific skill to be drilled and practiced. However, these exercises may not be precisely aligned with the teaching of the instructor. In any case, the instructor will have to carefully evaluate the suitability of the exercises and, where necessary, adapt them for the specific needs of the course.

When an APAS is to be used, the instructor has to consider additionally whether the exercise is amenable to automatic assessment. A programming task that is perfectly fit for student practice when manually assessed may be entirely impractical to be assessed automatically. For example, consider *Exercise 1* in Figure 1 which is sampled from a textbook on computer programming (Bentley, 1986).

Exercise 1. Write a "banner" procedure that is given a capital letter as input and produces as output an array of characters that graphically depicts that letter.

Figure 1. A programming exercise with vaguely specified requirements (Bentley, 1986).

Here the program requirements are vague and lack the necessary details for determining correctness in an objective manner. Given a capital letter, there are a myriad of ways it can be "depicted graphically" by an array of characters. While a human being can easily visually distinguish a correct output from an incorrect one, it is close to impossible for automatic assessment of the correctness of the output, given the numerous fonts, sizes, styles, and other attributes that a "graphically-displayed" character may possess. Thus, *Exercise 1* may be manually assessed but not easily by an APAS.

A simple remedy is to supplement *Exercise 1* with a sample output as in *Exercise 1a* (Figure 2). But *Exercise 1a* is still inadequate as it neither specifies the output for other inputs (such as 'B') nor states unambiguously whether other character arrays (such as a larger array or an array with a different length-height ratio) are acceptable as correct outputs for the input 'A'. These issues may be minor if the program is visually assessed by a human being, but become problematic with automated assessment.

With a view to systematically identifying the extent of these and other problematic issues, we performed a case study by examining the programming exercises in four commonly used textbooks (Bentley, 1986; Dale & Weems, 2005; Deitel & Deitel, 2005; Etter, 2005) and a publicly accessible online programming course (MIT, 2010). One of us judged whether the exercises could be adopted for automated assessment and, if yes, how much effort would be needed. If not, the causes, obstacles or concerns were identified and recorded. Another one of us re-examined these exercises, categorised the issues and illustrated each issue with one or more samples. Finally, the other authors reviewed the overall results with comments. It turned out that we all agreed with the results without amendment.

Exercise 1a. Write a "banner" procedure that is given a capital letter as input and produces as output an array of characters that graphically depicts that letter. *For example, if the input is* 'A', *the output is:*

```
    A
   A A
  AAAAA
 A     A
A       A
```

Figure 2. A programming exercise from (Bentley, 1986) supplemented with a sample output.

3. Issues in the Adoption of Programming Exercise for Automated Assessment

In this section, we categorise a list of problems and issues found in the adoption of programming exercises for automatic assessment. Limited by the scope of this case study, the list is not meant to be exhaustive, but it does illustrate the common difficulties that instructors typically encounter and their possible pedagogical concerns in adopting these programming exercises for assessment by an APAS.

3.1. Issues Related to Program Input/Output

Some exercises are not directly suitable for automatic assessment due to the program inputs or outputs.

Programs with no input: Interestingly, some textbook programming exercises (such as *Exercise 2* in Figure 3) only require the program to produce an output without the need to accept any input. For such an exercise, an APAS can still automatically determine the output correctness of the program, but it may not be very useful in assessing students' ability to write the program using the required method. *Exercise 2*, for example, can be completed by "hardcoding" the program to print the multiplication table using a series of output statements instead of a "nested for loop" as required. This exercise is more suited for manual assessment by code inspection. Moreover, testing such a program is trivial as the output is the same every time it is executed. Pedagogically, to effectively assess student's learning, it is advisable to adapt the exercise to ensure the program behaves differently with varying inputs, such as changing the ending integer 10 to a variable n whose value is obtained from user input.

Exercise 2. Write a nested `for` loop that prints out a multiplication table for integers `1` through `10`.

Figure 3. A programming exercise that accepts no input (Dale & Weems, 2005).

Programs with no output: Some exercises (such as *Exercise 3* in Figure 4) require students to write programs that only manipulate their internal states (such as memory contents) but produce no output. The lack of observable external behaviour of the reorder function in *Exercise 3* makes it unsuitable to be directly assessed automatically using black box testing. One remedy is to "wrap" the function with a custom-coded driver program that accepts three integer input values for initializing the variables *a, *b and *c, invokes the reorder function, and then outputs the new values of the three variables. Then the driver program together with the reorder function can be assessed by an APAS.

Exercise 3. Write a function that reorders the values in three integer variables such that the values are in ascending order. Assume that the corresponding function prototype statement is

```
void reorder(int *a, int *b, int *c);
```

where `a`, `b` and `c` are pointers to the three variables.

Figure 4. A programming exercise that produces no output (Etter, 2005).

Unspecified or unclear input/output requirements: Some exercises describe clearly the program processing task, but the input/output requirements are not specified (such as *Exercise 1* in Figure 1 and *Exercise 4* in Figure 5) or unclear (such as *Exercise 1a* in Figure 2 and *Exercise 5* in Figure 6).

Exercise 4. Write a program to convert miles to kilometers. (Recall that 1 mi = 1.6093440 km.)

Figure 6. A programming exercise with unspecified output format requirements (Etter, 2005).

Exercise 5. Write a code segment that prints the days of a month in calendar format. The day of the week on which the month begins is represented by an `int` variable `startDay`. When `startDay` is zero, the month begins on a `Sunday`. The `int` variable `days` contains the number of days in the month. Print a heading with the days of the week as the first line of output. The day numbers should neatly align under these column headings.

Figure 5. A programming exercise with unclear output format requirements (Etter, 2005).

For *Exercise 4*, a common remedy by many instructors is to supplement it with a sample input/output, such as in *Exercise 4a* (Figure 7).

Exercise 4a. Write a program to convert miles to kilometers. (Recall that 1 mi = 1.6093440 km.) *A sample run is shown as follows, with the first line being the input and the second line the output.*

```
10
10 mi converts to 16.09 km
```

Figure 7. A programming exercise from (Etter, 2005) supplemented with a sample input/output.

While *Exercise 4a* is a lot better than *Exercise 4* in terms of having specified the expected output requirements, the former is still inadequate if the programming solutions are to be assessed by APASs built with a naïve implementation of string comparison for determining output correctness. This is because such an APAS would only treat the program output as correct when it *exactly* matches the expected output string. In practice, many students produce programs whose outputs are *admissible variants*, that is, outputs that differ “slightly” or “insignificantly” from the specified outputs but are still accepted as correct by a reasonable human assessor (Jackson, 1991; Tang, Yu, & Poon, 2010). For example, given the input value 10 for *Exercise 4a*, the following output strings (denoted by *s*1 and *s*2) are accepted by many instructors as admissible variants:

*s*1 = “`10 mi converts to   16.09 km`”

*s*2 = “`10 Miles convert to 16.09 Kilometers.`”

Here *s*1 differs from the expected output by having extra blanks in front of the kilometer value, while *s*2 uses the full names of the units with the first letter in uppercase (`Miles` and `Kilometers`) instead of their abbreviations, and ends with an extra full stop. Most instructors would agree that the outputs are correct and the students have demonstrated their knowledge of how to write a program to compute the correct kilometer values, which is the primary assessed learning outcome, even though the students chose to output in a format slightly different from the given one. Unfortunately, an APAS built with a naïve implementation of correctness determination algorithm would treat both *s*1 and *s*2 as incorrect.

Of course, the instructor can insist that both *s*1 and *s*2 are indeed incorrect because they do not conform *exactly* to the given sample output format, but experience has been documented that such an approach often caused student frustration and confusion (Jackson, 1991; Joy, Griffiths, & Royatt, 2005) that are counter-productive in the learning process (which the instructor would not like to see) due largely to the technical issue of automated assessment (Tang, Yu, & Poon, 2010).

As another example, *Exercise 5* in Figure 6 requires the student to write a program that prints the days of a month in a “neatly aligned” calendar format, but the format details are not specified adequately, such as the column widths, type of alignment (say, left or right aligned) within a column, or whether the days of the week are in long (such as “`Sunday`”) or short (such as “`Sun`”) form. As a result of such ambiguity, there are many admissible variants that differ in, say, the number of blank spaces between two columns. Moreover, such ambiguity cannot be eliminated by giving a few sample outputs. Many programs which are manually assessed to be correct may produce outputs that deviate slightly from the outputs of the “instructor-conceived model program”. Unfortunately, these “correct” student programs will not be tolerated by APASs built with a naïve method of determining output correctness.

To avoid confusion and post-mortem debates, some instructors chose to ensure “uniqueness” of correct outputs by writing lengthy and overly detailed specifications of the output format requirements, sometimes spanning more than a page for a simple programming task (Tang, Yu, & Poon, 2010). For instance, an instructor gave several specific examples of “incorrect outputs” (in addition to the task description and a sample run with the expected output “`Volume = 360`”), as follows.

“The following outputs will all be graded as incorrect for the above example:

* `volume = 360` (*reason:* “`Volume`” *misspelt as* “`volume`”)

* `Volume=360` (*reason: spaces around* `=` *sign missing*)

* `Volume =  360` (*reason: too many spaces around* `=` *sign*)

* `Volume = 360.` (*reason: additional dot at end of line*)”

However, any list of incorrect output samples can never be exhaustive, and the list may not be convincing to students or others. Moreover, compiling such a list together with a detailed specification is not only time-consuming, but also counter-educational as the exercise becomes overly restrictive, inhibits creativity and distracts students from the main programming task and intended learning goal.

Some instructors further chose to enforce the output requirements strictly and pre-warn students that any small deviation of the output format will be treated as incorrect. For example, an instructor using an APAS explicitly states that the APAS “*awards mark for correctness ONLY if your output adheres to the given format. Hence, do not add any other characters (such as blanks) that are not asked for in your output, or change the spelling in your output.*” Another APAS documents in its student guide that “*If you do have extra lines that are not blank, or missing lines, then the Curator may compare the wrong lines, in which case you will probably receive a very low score.*” (Curator, 2010) Such a “be-warned” strategy might reduce students’ complaints, but not necessarily their frustration, as evidenced by typical students’ remarks like “*too fussy*” or “*too picky with spaces*” (Joy, Griffiths, & Royatt, 2005), or “*Sometimes it is right to you but wrong to the automark*” (Suleman, 2008).

Some APASs implemented simple pre-processing algorithms such as filtering out extra spaces and ignoring the case of letters (so that uppercase and lowercase letters are not distinguished) before matching the output strings. For *Exercise 4a* above, for example, such an APAS would judge *s*1 as correct but still treat *s*2 as incorrect. Recently, a token pattern approach for determining output correctness has been proposed to provide more flexibility in comparing output strings in APASs (Tang, Yu, & Poon, 2009). Such an APAS would then treat both *s*1 and *s*2 as correct, which is closer to the ways that human assessors would judge while obviating the need for ultra-detailed output format specifications. A more detailed review of other strategies for dealing with the output correctness determination problem in APASs can be found in the article by Tang, Yu, & Poon (2010).

Programs with non-textual output: Programming exercises that involve graphical user interfaces (GUI) cannot be directly assessed automatically by common APASs which are designed for assessing textual outputs only. English (2004) proposed to adapt the wrapper/stub strategies (in which the instructor provides custom-designed drivers/stubs to students) for automatic assessment of GUI programs by converting the program input/outputs into text streams. Such strategies, however, have not been widely adopted by APASs in practice (Thornton et al., 2008; Tang, Yu, & Poon, 2010).

3.2. Issues due to Non-determinism

Some exercises cannot be easily adopted for automated assessment due to the non-deterministic nature of the function calls or operators involved whose resulting values are dynamically generated and cannot be predicted even when the inputs are known. An instructor who relies on an APAS for assessment may tend to avoid these kinds of exercises that require manual judgment, thus inducing the pedagogical risk of missing assessment of these topics related to these functions or operations in the course. Some examples are random number generator function and pointer or memory manipulation operations.

Random number generation: *Exercise 6* in Figure 8 cannot be directly adopted for APASs because of the non-deterministic values returned by the random number generator function rand.

Exercise 6. (*Computers in Education*) Computers are playing an increasing role in education. Write a program that helps an elementary school student learn multiplication. Use `rand` to produce two positive one-digit integers. It should then type a question such as:

```
How much is 6 times 7?
```

The student then types the answer. Your program checks the student's answer. If it is correct, print "`Very good!`", then ask another multiplication question. If the answer is wrong, print "`No. Please try again.`", then let the student try the same question repeatedly until the student finally gets it right.

Figure 8. A programming exercise involving random numbers (Deitel & Deitel, 2005).

Such exercises can even be difficult to assess manually if only black box testing is performed. If a program is supposed to output three random integers within 1 and 100, how can a human assessor know whether an output of the three integers `99`, `12` and `31`, say, is correctly generated by the program using the `rand` function or not? (The output numbers may as well be produced by some special formula unrelated to `rand`, or perhaps extracted from the year, month and day of a certain date!) The best way to assess such programs is to inspect the program codes, which cannot be easily automated.

Pointer or memory manipulation: A pointer holds the address of a memory location for direct manipulation of data in the location. It is a fundamental element of many other data structures such as linked lists. Since memory addresses are non-deterministic internal states generated dynamically at run-time, exercises (such as *Exercise 7* in Figure 9) that require pointer or memory manipulation operations generally cannot be automatically assessed by black box testing. To assess these exercises, the codes are either manually inspected or instrumented (say, with the aid of a debugger) to trace their execution.

Exercise 7. Write a code segment that checks whether the pointer `oldValue` actually points to a valid memory location. If it does, then its contents are assigned to `newValue`. If not, then `newValue` is assigned a new `int` variable from the heap.

Figure 9. A programming exercise involving pointers (Dale & Weems, 2005).

3.3. Issues of Assessing Some Course Learning Outcomes

Some intended learning outcomes of a programming course are to develop students' ability to use some specific methods or styles of coding. Exercises specifically requiring the use of a coding method or style are usually not suitable for automated assessment by black box testing, which can only evaluate a program's external behaviour but cannot detect non-conformance to the requirements due to the use of a different method or style of coding. For example, *Exercise 8* in Figure 10 requires the student to write a program to swap two integer values with specific mandatory use of the *call-by-reference* method. But an APAS cannot distinguish from the program outputs whether *call-by-reference* is used or not.

Exercise 8. Write a function that swaps two integer values using call-by-reference.

Figure 10. A programming exercise specifically requiring the use of call-by-reference (MIT, 2010).

Learning outcomes involving the following are generally hard to be assessed by APASs. As such, instructors relying heavily on APASs may also tend to avoid assessing these learning outcomes.

Coding style: Coding style (Ala-Mutka, Uimonen, & Jarvinen, 2004) refers to a set of rules (related to variable naming, use of indentation and comments, etc.) that guide the writing of the source code to ease the effort required to understand and maintain the code. Again these rules only affect human reading of source code but not its execution, and so they are better assessed manually or by static analysis.

Specified control structure: Some exercises are designed for students' practice of certain control structures, such as recursion. Many students may find recursive function calls difficult to code and use iterations instead. Since automated assessment by black box testing normally cannot distinguish the use of recursion or iteration, such exercises are usually manually assessed by inspecting the source code.

Behaviourally-equivalent algorithms: A programming task can typically be coded by using different algorithms that exhibit the same functional behaviour. If an exercise requires students to practice a specific algorithm (say, binary search), whether the student's submitted program really implements the required algorithm cannot be easily assessed automatically by means of black box testing.

Specified data structure: Data structures, such as arrays and linked lists, are ways for storing and organizing groups of data so that they can be managed neatly and manipulated efficiently. Some students may find the handling of one type of data structure (say, array) easier than another (say, linked list) and keep using the "easier" one despite the explicit requirement that specifies the use of another. The resulting program that uses an "easier" data structure may perform the same function despite not using the required data structure, which cannot be distinguished by means of black box testing.

4. Recent and Future Work

Our case study has identified and categorised a list of issues when adopting programming exercises for assessment by APASs. A common category of issues is related to program input/output, such as non-existent or underspecified input/output or the use of GUIs. The recently developed token pattern approach for output correctness determination has partly resolved the difficulties of correctly assessing admissible variants without the need of writing overly detailed and rigid specifications (Tang, Yu, & Poon, 2009), while the wrapper strategies proposed by English (2004) have partly addressed the assessment of GUI outputs, but further research to advance these technologies and broaden their applicability is still needed in these directions (Thornton et al., 2008; Tang, Yu, & Poon, 2010).

Another category of issues is due to the non-deterministic functions or operators being practised. Some kinds of non-determinism can be eliminated, such as by fixing the seed of the random number generator. For other types of non-determinism, further research may study the applicability of instructor-defined stubs or drivers (Jackson, 1991; Tremblay et al., 2008; Tang, Yu, & Poon, 2010) to convert these programs into deterministic ones for assessment by APASs.

Finally, there are issues of assessing learning outcomes that require the use of a specific method or style of coding. If not addressed, instructors who rely on APASs may tend to avoid these kinds of exercises, causing some learning outcomes to be not assessed in the programming course. By nature, whether a program uses a specific coding style or method is hard to assess by testing and more feasibly by static code analysis. Some recent research has attempted to incorporate static analysis in APASs to perform such tasks as program style assessment (Ala-Mutka, Uimonen, & Jarvinen, 2004) and algorithm recognition (Taherkhani, Malmi, & Korhonen, 2008). Further work in this direction appears promising, but the applicability of most of the existing research results to real APASs is still limited (Striewe & Goedicke, 2014). Our case study has contributed to a categorisation of the main technical issues that hinder the adoption of some programming exercises for automated assessment, providing a basis for further and more focused concerted efforts in systematically addressing these issues.

Acknowledgements

The work described in this paper is supported by the grants from Research Grants Council of the HKSAR (project numbers UGC/FDS11/E02/15 and 11208017).

References

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., & Jarvinen, H.-M. (2004). Supporting students in C++ programming courses with automatic program style assessment. *Journal of Information Technology Education*, *3*, 245–262.

Bentley, J. (1986). *Programming Pearls*. Addison-Wesley Publishing Company.

Chong, S. L., & Choy, M. (2004). Towards a progressive learning environment for programming courses. *Proceedings of the 3rd International Conference on Web-based Learning* (*ICWL 2004*), 200–205.

Curator (2010). *Curator System Student Guide* (September 2010). http://courses.cs.vt.edu/curator/StudentInfo/CuratorStudentGuide.doc.

Dale, N., & Weems, C. (2005). *Programming and Problem Solving with C++*. Jones and Bartlett Publishers.

Deitel, H. M., & Deitel, P. J. (2005). *C++ How to Program*. Pearson Education International.

English, J. (2004). Automatic assessment of GUI programs using JEWL. *Proceedings of Annual Conference on Innovation and Technology in Computer Science Education* (*ITiCSE 2004*), 137–141.

Etter, D. M. (2005). *Engineering Problem Solving with C*. Pearson Education, Inc.

Higgins, C., Hergazy, T., Symeonidis, P., & Tsinsifas, A. (2003). The CourseMarker CBA system: Improvements over Ceilidh. *Education and Information Technologies*, *8*(3), 287–304.

Ihantola, P., Ahoniemi, T., Karavirta V., & Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. *Proceedings of Koli Calling International Conference on Computing Education Research*, 86–93.

Jackson, D. (1991). Using software tools to automate the assessment of student programs. *Computers and Education*, *17*(2), 133–143.

Joy, M., Griffiths, N., & Royatt, R. (2005). The BOSS online submission and assessment system. *ACM Journal on Educational Resources in Computing*, *5*(3), Article 2.

Law, K. M. Y., Lee, V. C. S., & Yu, Y. T. (2010). Learning motivation in e-learning facilitated computer programming courses. *Computers and Education*, *55*(1), 218–228.

MIT (2009). Course 6.096: Introduction to C++. *MIT OpenCourseWare*. http://hdl.handle.net/1721.1/74125.

Siddiqi, R., Harrison, C. J., & Siddiqi, R. (2010). Improving teaching and learning through automated short-answer marking. *IEEE Transactions on Learning Technologies*, *3*(3), 237–249.

Staubitz, T., Klement, H., Renz, J., Teusner, R., & Meinel, C. (2015). Towards practical programming exercises and automated assessment in massive open online courses. *Proceedings of IEEE International Conference on Teaching, Assessment, and Learning for Engineering* (*TALE 2015*), 23–30.

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

Suleman, H. (2008). Automatic marking with Sakai. *Proceedings of Annual Conference of the South African Institue of Computer Scientists and Information Technologists 2008* (*SAICSIT 2008*), 229–236.

Taherkhani, A., Malmi, L., & Korhonen, A. (2008). Algorithm recognition by static analysis and its application in students' submission assessment. *Proceedings of Koli Calling International Conference on Computing Education Research*, 88–91.

Tang, C. M., Yu, Y. T., & Poon, C. K. (2009). An approach towards automatic testing of student programs using token patterns. *Proceedings of International Conference on Computers in Education* (*ICCE 2009*), 188–190.

Tang, C. M., Yu, Y. T., & Poon, C. K. (2010). A review of the strategies for output correctness determination in automated assessment of student programs. *Proceedings of Global Chinese Conference on Computers in Education* (*GCCCE 2010*), 551–558.

Thiébaut, D. (2015). Automatic evaluation of computer programs using Moodle's Virtual Programming Lab (VPL) plug-in. *Journal of Computing Sciences in Colleges*, *30*(6), 145–151.

Thornton, M., Edwards, S. H., Tan, R. P., & Pérez-Quiñones, M. A. (2008). Supporting student-written tests of GUI programs. *ACM SIGCSE Bulletin*, *40* (1), 537–541.

Tremblay, G., Guérin, F., Pons, A., & Salah, A. (2008). Oto, a generic and extensible tool for marking programming assignments. *Software — Practice & Experience*, *38*(3), 307–333.

Wang, F. L., & Wong, T. L. (2007). Effective teaching and learning of computer programming with large class size. *Proceedings of Symposium on Hybrid Learning* (*SHL 2007*), 55–65.

Yu, Y. T., Choy, M. Y., & Poon, C. K. (2006). Experiences with PASS: Developing and using a programming assignment assessment system. *Proceedings of International Conference on Quality Software* (*QSIC 2006*), 360–365.

Yu, Y. T., & Tang, C. M. (2012). On the characteristics of programming exercises that affect their suitability for automated assessment. *Proceedings of Global Chinese Conference on Computers in Education* (*GCCCE 2012*), 661–662.