PERGAMON

Computers & Education 41 (2003) 121–131

COMPUTERS & EDUCATION

www.elsevier.com/locate/compedu

On automated grading of programming assignments in an academic institution

Brenda Cheang[a,*], Andy Kurnia[a], Andrew Lim[b], Wee-Chong Oon[c]

[a]*IOPT, SoC Incubation Centre, National University of Singapore, 3 Science Drive 2, Singapore 117543, Singapore*
[b]*Department of IEEM, Hong Kong University of Science and Technology Clear Water Bay, Kowloon, Hong Kong*
[c]*Department of Computer Science, National University of Singapore 3 Science Drive 2, Singapore 117543, Singapore*

Received 21 June 2002; accepted 28 January 2003

Abstract

Practise is one of the most important steps in learning the art of computer programming. Unfortunately, human grading of programming assignments is a tedious and error-prone task, a problem compounded by the large enrolments of many programming courses. As a result, students in such courses tend to be given fewer programming assignments than should be ideally given. One solution to this problem is to automate the grading process such that students can electronically submit their programming assignments and receive instant feedback. This paper studies the implementation of one such automated grading system, called the Online Judge, in the School of Computing of the National University of Singapore for a compulsory first-year course that teaches basic programming techniques with over 700 students, describing the student reactions and behavior as well as the difficulties encountered. The Online Judge was also successfully employed for an advanced undergraduate course and an introductory high school course.
© 2003 Elsevier Ltd. All rights reserved.

Keywords: Automated grading; Online judge; Computer science; Education

1. Introduction

One of the primary functions of an academic institution that teaches computer science must be to imbue all its students with the ability to program. After all, in order for anyone to claim to have knowledge of computer science, the ability to write a working program is a definite prerequisite. Furthermore, one of the most important steps in learning the art of programming is practise, i.e. getting one's hands dirty by actually sitting down to write working programs. Indeed, when it comes to learning computer programming, practise makes perfect.

* Corresponding author.

0360-1315/03/$ - see front matter © 2003 Elsevier Ltd. All rights reserved.
doi:10.1016/S0360-1315(03)00030-7

The School of Computing in the National University of Singapore offers a number of courses that teach the art of computer programming to its students. These include both basic courses that teach the fundamentals of programming (for first year students) as well as advanced courses that teach more complicated concepts. Almost all of these courses involve programming assignments that require the students to write working programs based on the concepts that are taught. However, programming assignments are notoriously difficult to manually grade in a fair and timely manner. Programs that appear to work on certain test cases may fail on others, and programs that are not entirely correct must still be graded on a relative scale based on several factors. Therefore, human graders generally have no choice but to strenuously examine each line of code in order to give a grade. Other problems run the gamut from human subjectivity and favoritism to the dependency of graders on "model" answers. In fact, the labor-intensiveness of grading programming assignments is the main reason why few such assignments are given to the students each semester, when ideally they should be given many more.

It is obvious that great potential benefit can be reaped if the grading of programming assignments can be automated. There have been several studies into the feasibility of using computer networks and other technologies in programming courses (Arnow, 1995; Kay, 1998; Mason & Woit, 1998). However, considering the potential benefits of automated grading, few academic institutions have implemented such a system. Examples include the TRY system (Reek, 1989); a "Supporting Technologies" course at the University of Brighton that is automatically assessed (English & Siviter, 2000); and the *Scheme-Robo* system implemented in the Helsinki University of Technology for automated assessment of exercises in the Scheme functional language (Saikkonen, Malmi, & Korhonen, 2001).

There are several issues that need to be addressed before such a system can be used. These include the danger of malicious programs, plagiarism and various psychological aspects of automated grading. An online grading system that checks for program correctness, called the Online Judge, was used in the third year course *CS3233 Competitive Programming* in July 1999 in the School of Computing in the National University of Singapore. While this implementation was relatively problem-free, the acid tests occurred when it was used in *CS1102 Data Structures & Algorithms* in January 2000, a first year course that teaches fundamental data structures with an enrolment of 712, and in January 2001 with an enrolment of *795*. With its successful implementation for *CS1102,* the Online Judge was then used in a high school introductory programming module *Object-Oriented Programming Using C++* in Victoria Junior College.

This paper examines the impact of implementing an automated programming assignment grader, and how its various issues are addressed. We trace the problems with human grading of programming assignments, and propose a model that can be used to implement an automated grader, supplementing the sparse literature currently available (Preston & Shackelford, 1999). This model was successfully employed in the implementation of the Online Judge system. Section 2 gives a background on the reasons for the implementation, while Section 3 highlights the problems inherent in human grading of programming assignments. We then describe the Online Judge in Section 4, explaining how it works and listing the verdicts that the Online Judge can deliver. In Section 5, the details of implementing the Online Judge for the basic programming course *CS1102* are given. Section 6 then goes on to give the observations made over the course of the semester and the problems encountered. Section 7 touches on the Online Judge's latest application in an introductory high school course in Victoria Junior College. Finally, we conclude

the paper in Section 8 by reviewing the advantages and pitfalls present in automating the grading of programming assignments, and possible future work.

2. Background

Since the formation of the School of Computing in 1998, there has been a consistent increase in enrolment. While only about 500 students enrolled in July 1998, about 700 enrolled in July 1999 and about 800 in July 2000. This increase in student intake has not been matched by a corresponding increase in teaching staff strength. Therefore, the ratio of students to teaching staff has been increasing, exacerbating the already heavy workload of the lecturers and tutors. Consequently, the number of teaching staff available to evaluate and grade the assignments limits the number of assignments given to the students.

This problem is especially apparent in the case of programming assignments, where the student is asked to write a program such that, given any legal input, the correct output (as specified by the question) is produced. The nature of programming assignments makes them perhaps the most difficult type of assignment to grade. There are three main components involved in the grading of programming assignments:

1. Correctness
   Given any legal input, a program is considered *correct* if it produces the desired output. The ability of the program to handle illegal input (by displaying the appropriate error message), or *robustness,* is comparatively less important than its ability to perform the basic functions of the program.
2. Efficiency
   A program is *efficient* if it performs its tasks without consuming too much processing time and memory space. How much is ''too much'' depends on the programming question.
3. Maintainability
   A program is *maintainable* if the code is easily understandable, employing devices such as descriptive variable names, comments, indentation and modular programming.

Obviously, the most important component when grading a programming assignment must be its correctness, since a program must first be able to perform its basic functions before it can claim to have answered the question. Efficiency is a close second, as a program that may be theoretically correct but is too slow or overflows the system memory is effectively useless. When viewed in this way, maintainability can be regarded as the least important component: although it is vital to the success of large business application programs, it should not be overly focused upon when learning the art of programming.

3. Human grading

Prior to the 1999/2000 academic year, all programming assignments were manually graded. Due to the difficulty in marking such assignments, only about three or four were given to the students per semester. As a result, each assignment tends to cover several topics and involves creating a moderately sized program complete with the peripheral tasks like user interface, error

messages and lengthy documentation. Students were given 2–3 weeks to complete each assignment. They may approach their assigned Teaching Assistants (TA), usually Honors year students, for help. On the day of the deadline, each student e-mails a softcopy of their program to their TA, and also hands in a hardcopy of their source code along with whatever other documentation.

There are several problems to this arrangement. Human graders have great difficulty in judging the correctness and efficiency of programs. As a result, many human graders tend to overemphasize maintainability. Some of the problems cited by both students (during the online feedback survey conducted at the end of each semester) and teaching staff are:

1. Judging correctness and efficiency
It is difficult for the human grader to judge the correctness and efficiency of a program simply by running it. Firstly, a program that does not run correctly must be examined line by line in order to discover the reason for its failure, so that a relative measure of correctness can be assigned. Secondly, a program that runs correctly and efficiently on some test data may fail to do so on other test data with different characteristics. Therefore, the test data devised must be comprehensive, which is difficult to do when the program scope is large. The efficiency of a program is also often best determined by examining the program code to discover the data structures and algorithms used. As a result, human graders often have to examine program submissions line by line even if they appear to work correctly.
2. Multiple approaches to the same problem
Given a problem, there may be many distinct solutions, all of which produce the desired result. Human graders are usually given a "model" answer to compare the submissions to, but this model answer necessarily only shows one possible approach. If the human grader were unfamiliar with the approach, he would be hard put to evaluate its correctness, let alone its efficiency. As a result, human graders tend to give lower grades to programs that are perfectly correct but different from the "model" answer.
3. Emphasis on aesthetics
Since it is so difficult to gauge a program's correctness and efficiency, human graders tend to place a disproportionate amount of emphasis on the maintainability of programs, which is reflected by their marking schemes for programming assignments. For example, an actual marking scheme conferred 10% of the grade to each of "appropriate identifiers", "appropriate comments", "indentation" and "modularity", making up 40% of the total assignment grade. By doing so, it is hoped that the submissions would be well-indented and commented, and therefore easier to mark. This has the unfortunate side effect of forcing students to spend extra time and effort on these aspects of the program to the detriment of its correctness and efficiency. Furthermore, human graders tend to give partial marks to programs that are "almost correct" as a reward for effort put in, even if the program does not even compile.
4. Human factors
The most common grievance that students have against human grading is inconsistency. The same program when graded by different human graders will almost always result in different grades. In fact, the same person may give a different grade to the same program depending on his mood, alertness or other factors. Of course, there is always the possibility of human error. It is all too easy to overlook an extra semicolon in a 500-line program that prevents the program from running correctly.

5. Time
As previously stated, human grading of programming assignments is a slow and tedious process involving the examination of every single line of code. Each human grader typically evaluates dozens of programs per assignment, which can take days to complete. Not only is this a huge burden on the marker's time, this slowness also makes it difficult for the students to obtain timely feedback on the mistakes that they might have made. By the time the programs have been evaluated and graded, the assignment is no longer fresh in the students' minds and feedback to them will be less effective.

Several other undesirable trends have been observed. Few students take advantage of their TA for consultation. This may be due to the lack of feedback that they get in the course of their programming: they may simply not know that they are doing anything wrong. It has also been observed that the students spend an unnecessary amount of effort on peripherals like user interface, ''user-friendly'' error messages and so on, since they are not sure whether they have done enough to ensure a satisfactory grade. This is both time-consuming and largely pointless to the task of learning the art of programming.

4. The Online Judge

The Online Judge was initially implemented in July 1999 for the third year course *CS3233 Competitive Programming,* which was used as preparation for the ACM Intercollegiate Programming Contest. This is a competition whereby teams of three are sent by each participating college through a set of regional trials before they can qualify for the finals. During each contest, the teams are given a set of six questions and a set amount of time to solve the questions. One point is given for each question solved, and penalty points are awarded for each submission of a wrong answer. The correctness of a submission depends on the program's ability to produce the desired result when run on the hidden input files set by the contest organizers within the set time limit. This submission is done online, and the result of the submission is returned instantaneously, in the form of an n-character string, where n is the number of test cases.

The Online Judge mirrors the ACM system. It is a simple application that basically runs the submitted programs with strict memory and time limits on a set of input files in a secure remote directory, and then compares the output to the pre-specified answers using simple string matching. The *correctness* of a submission is determined by whether the produced output matches the pre-specified answers. The *efficiency* of a submission is determined by whether the program is able to produce its output within the time limit and memory resources supplied. Giving erroneous or extreme test cases in the input files can test the *robustness* of a submission. However, the Online Judge cannot determine a submitted program's *maintainability*.

The dedicated Online Judge server used a 450 MHz processor running on the Linux operating system. The program was predominantly coded in C, with some parts in Perl and shell script. Currently, the Online Judge supports submissions made in C, C++ and Java, with provisions to include other languages should the need arise. The submission process made use of a wrapper program, which essentially opened up a port to the server to take in the submission with the user's ID and password. Submissions are processed on a first-come-first-served basis.

Several measures were taken to thwart the efforts of malicious or unethical users. These include using the Linux system function **chroot** to isolate the submitted programs within their own directory subtrees, running the programs as non-existent unprivileged users and setting strict resource limits (Kurnia, 2001).

The possible judgments for each test case include *Accepted* (short form: **.**); *Format error* (short form: **f**), whereby the output produced did not match the expected output exactly, but matches if whitespace and case sensitivity are ignored; *Wrong answer* (short form: **x**); and *Time limit exceeded* (short form: **t**), whereby the program failed to produce an output within the prescribed time limit. Therefore, a verdict of **[..x.t.]** describes the submitted program's performance on six test cases. It gave the correct answer for cases 1, 2, 4 and 6. It gave a wrong answer for case 3, and failed to produce an output within the prescribed time limit for case 5.

Fig. 1 shows a screen capture of the verdict log of the Online Judge. In this example, there are three test cases for question number 205. Among those whose submissions were accepted was a user with ID *zhangxil* (actual name Zhang Xiaoli, as shown on the hover tooltip), whose program produced the correct result for all three test cases in a total of 0.85 s, using 3592 KB of memory.

The Online Judge was invaluable in the conducting of the *CS3233* course. The students are given simulated programming contests every 2 weeks, using the same conditions as those found in the actual contest. Since this module is used to prepare for the ACM Programming Contest, the enrolment is low (eight in 1999) and generally comprises students who are very proficient programmers. The instantaneous response of the Online Judge system and its unforgiving requirement on accuracy and robustness of the program submissions helped the final NUS team achieve a creditable 22nd position in the 2000 competition.

5. Online Judge for a basic undergraduate course

In January 2000, the decision was made to modify the Online Judge system for use in *CS1102 Data Structures and Algorithms*. This is an essential first year course that all students must pass to graduate, teaching basic programming data structures and concepts like linked lists and recursion. The system was used again in January 2001. The Online Judge provided the correctness and efficiency grades for 11 programming assignments for all of the students (712 in 2000, 795 in 2001). Human graders are still required for maintainability marks. The 2001 version of the Online Judge can be found at http://andy.comp.nus.edu.sg/-cs11021.

When the Online Judge was used for *CS3233 Competitive Programming,* its low enrolment meant that cases of plagiarism would be easily detectable. However, for such a large course as *CS1102,* it is essential to have a facility to detect plagiarism cases. Plagiarism detection has been the subject of much study (Baker, 1995; Clough, 2000; Wise, 1992), and many plagiarism detection programs are available. The Online Judge package was modified to include a function that tries to detect cases of plagiarism, which we call the Cheater Checker. Currently, the Cheater Checker makes use of the **sim** program (Grune & Huntjens, 1989) with minor modifications. In brief, the program first uses the UNIX command **lex** to tokenize the programs into their lexical equivalents, which can be viewed as a long string. A forward-reference table of tokens is prepared, which is used to compare the two target programs for their longest common substrings of at least a minimum length. These substnngs are removed from the programs, and the operation is

repeated until no more such substrings can be found. In this way, the Cheater Checker computes the percentage of one program that is a subset of the other. We were thus able to effectively detect cases where a student plagiarized by adding redundant code, changed variable/method names or swapped the order of method statements. All cases of possible cheating are re-scrutinized by human evaluators. If the likelihood of cheating is high, the students involved are given a chance to plead their case in person. If it is not an obvious case of plagiarism, the students are generally given the benefit of the doubt. While we cannot claim that the Cheater Checker will be able to detect all instances of plagiarism, a significant number of such cases that would have slipped past human graders will be detected.

With the Online Judge system in place, the students were given a half-hour talk on the intricacies of the system in the first lecture. Two example questions were given to the students to help them familiarize themselves with the system. Over the course of the semester, the students were given a total of 11 programming assignments that were significantly shorter than those given in previous years. Each is designed to bolster the understanding on only one or two isolated aspects of programming. For each assignment, at least 10 secret test cases were constructed for the Online Judge to run. When students submit their program, the system reveals to the students the results on each test case. If the system reports that all the test cases have been negotiated correctly, the student is assured of all the correctness and efficiency marks. After the deadline, the final submitted codes are distributed to the TAs for the maintainability grades. We believe that this implementation model is suitable for any generic automated grader.

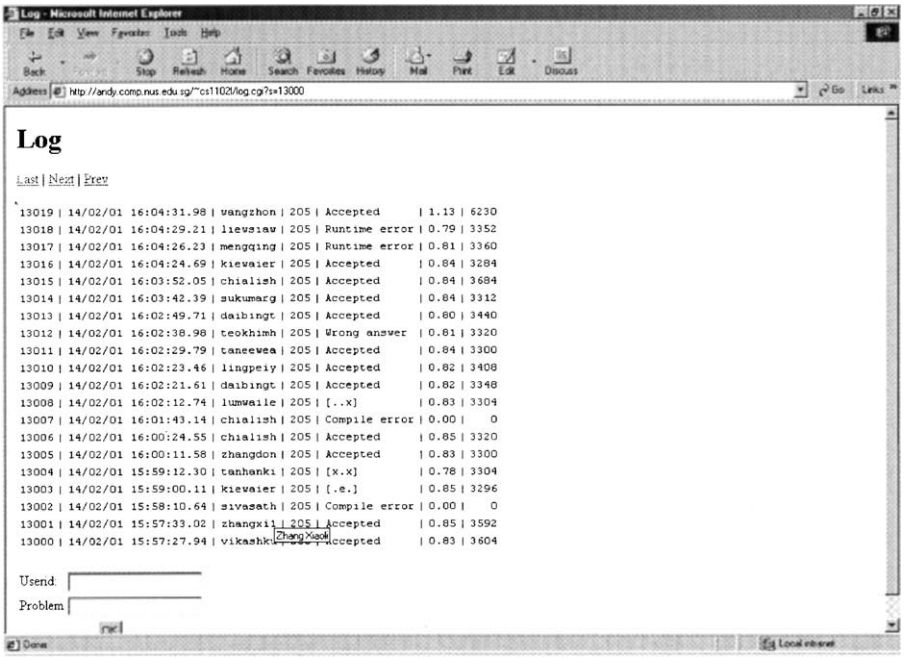

Fig. 1. Screen capture of the verdict log of the Online Judge.

It is difficult to assess the effects of the Online Judge on the programming ability of the students. Comparisons with the results of previous semesters, when the Online Judge was not used, would be misleading since the questions assigned would vary in difficulty and volume. However, one claim that can be confidently made is that the Online Judge allows for more assignments to be given, so the students have more practise.

6. Observations

There are two major factors to consider when porting the Online Judge program from *CS3233* to *CS1102*. Firstly, *CS1102* is a basic programming course, involving some students that have never previously studied computer science. Secondly, the enrolment of *CS1102* is over 700 for both semesters that the Online Judge was used. This section describes the issues to be addressed due to these factors.

6.1. Plagiarism

Plagiarism is considered a very serious offence in the School of Computing. The detection of plagiarism for programming assignments has always been a difficult task for human graders, especially if different people grade the offending parties. On this first testing of the Online Judge, we were deliberately vague about the inner workings of the Cheater Checker, only stating that plagiarism "can be detected". Further, we warned the students that cheating would result in the student failing the course, and perhaps lead to more severe penalties. Nonetheless, the number of plagiarism cases detected for the first assignment in 2000 was an astonishingly high 98 out of the total enrolment of 712, contrasting with only one or two cases detected in previous years before the Online Judge was used. Several of these cases were instances where one program was 100% substrings of another program, overwhelming evidence of plagiarism.

When we confronted the suspected students, reactions were mixed. Some of the students admitted outright to the offence. Others vehemently denied the charge, arguing on the statistical probability of two independently developed programs being misjudged as plagiarism cases. In response, we displayed the offending program and highlighted the obviously similar code portions. There were also cases where the final submission presented a marked change in programming style when compared with earlier submissions by the same student. In the end, all of the suspected students admitted to the offence.

We believe that the amount of plagiarism detected is not in fact abnormally high. The number of plagiarism cases can be attributed to several factors:

1. Students believed that an automated grader would be unable to detect plagiarism, since its primary function is to run test cases and provide verdicts.
2. The Online Judge was able to compare all programs to all other programs. Copying from someone in another tutorial group was no protection.
3. Previously, programming assignments were longer and more involved. This allowed the students more variation. Therefore, a student could copy the main workings of another program, and then mask it by adding peripherals.

We took a lenient view to first-time offenders (after extracting promises that they would not cheat again) by giving them zero marks for that particular assignment only. After all, the point of the course was to get the students to learn how to program. If they fail the course due to their single error in judgment, they would have no incentive to try for the rest of the semester. Happily, the number of plagiarism cases dropped drastically as the semester wore on. In fact, in the following year, only 11 cases of plagiarism out of *795* students were found.

6.2. Queue hogging

Since the Online Judge is a dedicated server that processes the submissions in a first-come-first-served basis, queue hogging is a major problem. One reason is that students tend to use the Online Judge as a second "compiler". They create a program and submit it to the system, then modify the program according to the verdicts returned. This tends to fill the queue with poorly checked programs that have little chance of being the final submission for the student. Ideally, the student should only submit a program once, having already thoroughly tested their program offline. There were also cases of malicious students that wrote programs to repeatedly submit programs to the judge, deliberately hogging the queue. Two measures were implemented to solve this problem. First, all students were limited to a total of 10 submissions per assignment. Second, each user may only have one program in the queue at any one time.

6.3. Assignments

One side effect of the Online Judge was to force the setters of the programming assignments to phrase the questions in a completely concise and clear manner. This is because the Online Judge would evaluate all output that was not *exactly* the same as the model answer as wrong. This meant that more time was required to proofread and double-check the questions to make sure that there was no ambiguity. In effect, even as the students become better at programming, the tutors have to become better at setting questions.

Because of the need to impose concise questions and the ability to impose more programming assignments, the assignments given using the Online Judge tended to be smaller and more focused than before. While previously the students would be given a few assignments, each of which involved several topics spanning across several weeks' lessons, we can now divide each topic into separate assignments as they are taught to the students. This added focus of each assignment can only help students to understand individual concepts better.

6.4. Test cases

The Online Judge was ported over from its initial task of *CS3233 Competitive Programming,* which focused on difficult problems involving very stringent test cases. Unfortunately, this mentality was transferred over to *CS1102,* which gave the first year students considerable difficulty. Initially, too many of the test cases focused on extreme data. For example, out of 10 test cases, perhaps as many as four or five would test extreme cases like very large data, null data and worst cases. As a result, a program that could handle basic cases may score poorly because it did not handle the extreme cases well.

While it is important for a program to be robust, it is equally important not to discourage the students by giving an overly poor grade to a program that performs correctly on normal cases but not on obscure extreme cases. This was the source of much negative student feedback, and is a problem that will be addressed as more experience on question setting is gained.

7. Online Judge for an introductory high school course

After its successful implementation for *CS1102,* the Online Judge was commissioned by Victoria Junior College (VJC) for an introductory level programming module *Object-Oriented Programming Using C++* (Gi, Lim, & Kurnia, 2002). The Online Judge (under the name *Judge Code)* handles the practical programming assignments for this module. At the time of writing, this 2-month long module is being taught to 40 Junior College second year Computing students, and nine practise questions and two assessment questions have been set. The students may attempt the nine practise questions at any time.

The implementation of the Online Judge for this introductory level module is essentially identical to the more advanced courses. The difference lies in the questions given in the progranuning assignments, which must be simple enough for high school students to solve, while at the same time challenging enough to help the students learn the programming concepts taught. At present, the implementation seems to have a positive effect on the students, with over 1200 submissions assessed. Student feedback thus far has also been generally positive. Judge Code and its current statistics can be found at http://www.judgecode.com.

8. Conclusion and future work

The Online Judge system has proved to be a useful tool for three significantly different programming courses: the advanced third-year undergraduate course *CS3233 Competitive Programming;* the basic first-year undergraduate course *CS11O2 Data Structures and Algorithms;* and the introductory high school course *Object-Orient Programming Using C++*. This shows that it is both powerful enough to be used in advanced programming courses with skilled programmers as well as simple enough for novices. An automated programming assignment grader such as the Online Judge confers several advantages to an academic institution that teaches programming:

- More programming assignments can be given to students.
- As the assignments are smaller but more numerous, each assignment can be more focused on a particular topic.
- Students are given instantaneous feedback on their submissions.
- Correctness and efficiency are accurately and fairly graded.
- Human markers need only grade a program on maintainability.
- Course-wide electronic submissions help plagiarism-detection software to detect cases of plagiarism across the entire course enrolment.

However, there are also some pitfalls to be avoided:

- Test cases should not be overly stringent, especially in introductory or basic courses.
- Problem specifications and expected output must be clear and concise.
- The type of feedback is limited (the Online Judge cannot identify the source of errors).
- Maintainability must still be graded by humans.

The Online Judge program is currently undergoing major re-development to address some problems that have been discovered, as well as to include some other features. When completed, the new Online Judge program will be able to handle several courses simultaneously. A repository of past questions will be maintained in a database structure (in contrast to the current text file system) such that particular questions can be extracted via appropriate queries. Students can then practice on as many past questions as they desire, with the benefit of the Online Judge's instant feedback mechanism. There will also be a facility for users to upload questions into the database. Finally, a style checker that checks for consistency in indentation and naming conventions is being investigated, although it is unlikely to completely replace the human grading element.

Automated programming assignment grading is not difficult to achieve. The core program simply runs a submission and compares its output to the pre-defined answers. As long as care is taken over the nature of the questions assigned and security issues, the potential benefits of automated programming assignment grading are plain to see. Students can be given more programming practice than was possible when only human graders are available, which is a crucial component in learning the art of programming.

References

Arnow, D. When you grade that: using e-mail and the network in programming courses. *Proceedings of the 1995 ACM Symposium on Applied Computing*, May 1995, pp. 10–13, ACM Press, NY, USA.

Baker, B. S. On finding duplication and near-duplication in large software systems, *Proceedings of the 2nd IEEE Working Conference on Reverse Engineering*, July 1995, pp. 86–95, IEEE Computer Society Press..

Clough, P. (2000, July). Plagiarism in natural and programming languages: an overview of current tools and technologies, (*CS-00-05 Internal Report*). Department of Computer Science, The University of Sheffield.

English, J., & Siviter, P. (2000, July). Experience with an automatically assessed course. In *Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education,* (pp. 168–171).

Gi, S. C., Lim, A., & Kurnia, A. (2002). *Judge Code—a practical approach for learning problem solving through computer programming*. (Internal Report), Victoria Junior College.

Grune, D., & Huntjens, M. (1989). Het detecteren van kopieen bij informatica-practica. *Informatie, 31, (11)*, 864–867.

Kay, D. Large introductory computer science classes: strategies for effective course management, *Proceedings of the 29th SIGCSE Technical Symposium*, February 1998, pp. 131–134, ACM Press, NY, USA.

Kurnia, A. (2001). *Online Judge*. Honors year thesis, School of Computing, National University of Singapore.

Mason, D., & Woit, D. Integrating technology into computer science examinations. *Proceedings of the 29th SIGCSE Technical Symposium* February 1998, pp. 140–144, ACM Press, NY, USA.

Preston J., & Shackelford, R. Improving on-line assessment: an integration of existing marking methodologies, *Proceedings of the 4th Annual SIGCSE/SIGCUE on Innovation and Technology in Computer Science Education*, June 1999, pp. 29–32, ACM Press, NY, USA.

Reek, K. A software infrastructure to support introductory computer science courses, *Proceedings of the 27th SIGCSE Technical Symposium*, February 1996, pp. 125–129, ACM Press, NY, USA.

Saikkonen, R., Malmi L., & Korhonen, A. (2001). Fully automatic assessment of programming exercises. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*. (pp. 133–136).

Wise, M. J. Detection of similarities in student programs: YAP'ing may be preferable to plague'ing, *Proceedings of the 23rd SIGCSE Technical Symposium* March 1992, pp. 268–271, ACM Press, NY, USA.