Computer Science Education
Vol. 15, No. 2, June 2005, pp. 83–102

Routledge
Taylor & Francis Group

A Survey of Automated Assessment Approaches for Programming Assignments

Kirsti M. Ala-Mutka*
Tampere University of Technology, Tampere, Finland

Practical programming is one of the basic skills pursued in computer science education. On programming courses, the coursework consists of programming assignments that need to be assessed from different points of view. Since the submitted assignments are executable programs with a formal structure, some features can be assessed automatically. The basic requirement for automated assessment is the numerical measurability of assessment targets, but semiautomatic approaches can overcome this restriction. Recognizing automatically assessable features can help teachers to create educational models, where automatic tools let teachers concentrate their work on the learning issues that need student-teacher interaction the most.

Several automatic tools for both static and dynamic assessment of computer programs have been reported in the literature. This article promotes these issues by surveying several automatic approaches for assessing programming assignments. Not all the existing tools will be covered, simply because of the vast number of them. The article concentrates on bringing forward different assessment techniques and approaches to give an interested reader starting points for finding further information in the area. Automatic assessment tools can be used to help teachers in grading tasks as well as to support students' working process with automatic feedback. Common advantages of automation are the speed, availability, consistency and objectivity of assessment. However, automatic tools emphasize the need for careful pedagogical design of the assignment and assessment settings. To effectively share the knowledge and good assessment solutions already developed, better interoperability and portability of the tools is needed.

1. INTRODUCTION

An international survey of computer science academics studied current assessment practices and perceptions of computer aided assessment (CAA) on computer science courses (Carter et al., 2003). Results showed that several different types of assessment are used, but as the most common option, 74% of respondents had used practical work as assessment. While all types of assessment were submitted both manually and electronically, practical work was the only one submitted more often electronically. However, the most common marking technique for practical work was manual. When

*Corresponding author. Tampere University of Technology, P.O. Box 553, 33101 Tampere, Finland. E-mail: kirsti.ala-mutka@tut.fi

ISSN 0899-3408 (print)/ISSN 1744-5175 (online)/05/020083-20 © 2005 Taylor & Francis Group Ltd
DOI: 10.1080/08993400500150747

comparing the experience of using CAA with the perceptions of its ability to measure higher-order learning, it was seen that those with more experience saw CAA as a more versatile tool than the respondents with less experience. This was interpreted to reflect the fact that many teachers still see the possibilities of CAA to be limited to simple assessment tasks, such as multiple choice questions. The other major use seems to be submission management.

Actually, CAA can offer versatile possibilities for computer science education, especially when considering programming courses. In many fields, CAA is used mainly for information delivery and management, or as general assessment tool, not sensitive to the contents of the assignment. Such tools perform, e.g., different types of multiple-choice tests. On programming courses, students create systems and computer programs that follow the formal semantics of programming languages. The contents of a programming assignment can be parsed automatically and executed for studying its behavior. Therefore, it is easy to create automatic tools to study the program. The difficulty is in designing measurements that are relevant for program quality and learning programming.

Programming belongs to the core competence in both computer science and software engineering curricula. Programming courses are often large in size and cause heavy workload for the teacher, since many programming assignments are required for exercising programming in practice. Assessing and providing feedback on computer programs is time-consuming, because there are many aspects relating to good programming that need to be considered. The often advertised promises of CAA (speed, objectivity, consistency etc.) would really be needed in order to guarantee a reasonable amount of practical exercise and feedback for all students in programming courses. Unfortunately, the knowledge and experience of using automatic approaches for assessing computer programs have not been spread widely for general discussion. This article promotes these issues by gathering together different automatic assessment approaches for programming assignments to give an interested reader starting points for finding further information in the area.

The contents of the paper are organized as follows. In Section 2, some general discussion topics relating to assessment on programming courses are presented. Section 3 reviews different aspects of programs that have been assessed automatically, either by static or dynamic program analysis. Section 4 gives the reader an overview of the reported approaches for using automated assessment on programming courses. Section 5 discusses issues relating to the usage of automated assessment for programming assignments, based on both literature and this author's own experiences. Finally, Section 6 concludes the paper.

2. ASSESSING PROGRAMMING ASSIGNMENTS

The objectives and the assessment on programming courses are often discussed and questioned. The goals set by the teachers do not always seem to be achieved by the students. McCracken et al. (2001) found in their study that the programming skills of first year CS students were much lower than expected. In fact, most students could

not complete the required programming task in the given time. Lister and Leaney (2003) criticized the assessment practices and goals for the first programming courses in general. They proposed a criterion-referenced scheme, where each grade is clearly connected to certain requirements, and students have a possibility to decide themselves which grade they want to pursue. The programming assignments are designed according to different cognitive levels and the assignments on higher levels entitle to better grades that the assignments on lower levels. With this approach, it can be recognized and admitted that all students do not possess higher-level programming skills at the end of the first course.

Lister and Leaney encourage teachers to design assessment according to the cognitive levels defined in the Taxonomy of Educational Objectives (Bloom, 1956). These levels are, from lowest to highest: recall, comprehension, application, analysis, synthesis, and evaluation. Following these ideas, teachers could design programming assignments that emphasize, e.g., application (simple programming tasks), analysis (debugging tasks), or synthesis (advanced programming assignments) skills. This work is interesting when considering the perceptions that CAA can only be used for lower cognitive levels. If the result of an assignment is a correctly functioning and well-constructed computer program, similar (possibly automated) assessment approaches could be used for all the programs and the cognitive level of the task is determined by the assignment design.

One must not, however, forget that understanding of programming concepts can also be assessed without writing program code. Cox and Clark (1998) presented examples of multiple-choice questions that assess all cognitive levels of learning in an introductory programming course. In addition to the traditional multiple-choice tests, there are also other automated approaches for assessing programming concepts. Automated assessment can, e.g., be used for design diagrams, as implemented in CourseMarker (Higgins, Symeonidis & Tsinsifas, 2002). Another approach is to simulate algorithm execution, e.g., Trakla (Korhonen & Malmi, 2000) can generate algorithm simulation exercises and assess the answers automatically. However, understanding concepts and principles does not guarantee the ability to generate computer programs. Novice programmers have common problems in expressing their program solutions as computer programs, i.e., in applying programming concepts to program construction (Robins, Rountree & Rountree, 2003). Thus, if the goal of the course is to teach practical programming skills, these should be exercised and assessed by practical programming tasks. Woit and Mason (2003) also obtained positive results by using weekly quizzes that were directly connected to the practical programming exercises on the course.

Typically, programming assignments are assessed by the resulting programs, and the assessment criteria vary between different teachers and universities. For example, some use holistic assessment approaches and some more or less detailed analytic assessment criteria. Olson (1988) studied the differences between these approaches and noticed that they emphasize different features in the assignment. Holistically a program could get a reasonable mark even if it failed in some analytical categories, e.g., compilation or basic functionality, and would have

received a failed mark in the analytical grading. As the grading work in large courses often needs to be distributed to several tutors, the analytic approach makes it possible to define detailed grading criteria for common use. For example, Becker (2003) proposed rubrics for defining the grading criteria. Detailed criteria are a necessity for the tutors and could also be published for students, to help their self-directed learning. This is a kind of by-product with CAA; automation requires formally specified grading principles that could be transformed to a general guide for students' work.

In addition to assessing skills of the students by their programs, their working habits should be considered. Howles (2003) raised the issue of learning software quality culture for discussion. She discovered from a local student survey that only 5% of the responding students always designed their work before coding and only 39% always tested their program code statically. Moreover, majority of the students executed unit tests only sometimes or never. The working process of the students is difficult to assess, but it can be guided by the assignment and assessment design. Howles proposes, among others, incremental grading and requiring students to find, fix and document all the defects found in the assessment. Automated assessment could be used also in this kind of a process for helping teachers and students to manage and compare different program submission versions.

Although many authors seem to appreciate the objectivity and efficiency of the automated assessment of programming assignments (e.g., Foxley, 1999; Chen, 2004; Morris, 2004), also opinions against it have been presented. Ruehr and Orr (2002) discussed different assessment criteria and considered interactive demonstration as the best assessment method for programming assignments. They see it as a rewarding situation for both students and instructors. A personal contact situation guarantees versatile and individual feedback on the program for the student. The approach is best suited for small student groups, but could also be used selectively in larger classes.

3. AUTOMATIC ASSESSMENT FOR DIFFERENT FEATURES

Several approaches to automated program assessment can be found from journal and conference articles as well as from other sources. The basic requirement for automated analysis is that some kind of measurement values can be extracted from the program and that the values can be compared to given requirements or to a model solution. For an educational use, the measurement values also need to be justified by the teaching goals of the course.

This section lists program features that have been automatically assessed by different assessment tools and systems reported in the literature. The focus is on approaches developed for programming assignments that are implemented with standard programming languages and tools. In this presentation, the features are organized according to whether they need execution of the program or can be statically evaluated from the program code.

3.1. Dynamic Assessment

Kay, Scott, Isaacson, and Reek (1994) already stated that it is not possible to consistently and thoroughly grade students' programs without automated assistance. This applies especially to dynamic features of the program, since even small programs typically have a large number of possible execution paths. Automation provides means to systematically cover a large number of different execution possibilities. However, running a program written by a student is a risky task. The program may have bugs — or even intended malicious features — that lead the program to try actions that could cause damage. For example, a program may try to delete or read files from the running environment, e.g., teacher's machine or assessment system database. Nor is it rare that student programs have bugs that cause programs to reserve huge amounts of memory or CPU time, hindering other processes running on the computer. These are concerns that always need to be taken into account when testing students' programs. Thus, an essential requirement for automated dynamic assessment is to provide a secured running environment, so called sandbox, for running students' programs without risks to the surrounding environment.

3.1.1. Functionality. The most common form of assessment for programming assignments is to check that the program functions according to the given requirements. The functionality is usually tested by running the program against several test data sets. The coverage of the assessment depends on the test case design. The results are typically compared either to a separate specification or by executing a model program for comparison. The correctness of the functionality is then compared either by the printed output or the return values.

Functionality assessment tools for programming assignments such as Try (Reek, 1989), were implemented already in the 1980's. This kind of assessment is nowadays typically included to all versatile assessment tools, such as Ceilidh (Foxley, 1999) that is nowadays CourseMarker (Higgins, Hergazy, Symeonidis, & Tsinsifas, 2003), Assyst (Jackson & Usher, 1997), HoGG (Morris, 2003), Online Judge (Cheang, Kurnia, Lim, & Oon, 2003) and BOSS (Luck & Joy, 1999). These assess the functionality of the program by comparing its output. Ceilidh/CourseMarker also checks for the return status of the program.

The basic implementation for output comparison is to compare the program output text to the model output text, possibly ignoring whitespace characters. Assyst uses pattern matching implemented with Lex and Yacc while Ceilidh/CourseMarker and HoGG use regular expressions for defining the assessment criteria for program output. These approaches offer teachers a possibility to provide students with a certain degree of freedom in the output format, if considered necessary.

It is also possible to evaluate automatically the functionality of smaller entities than complete programs. For example, Quiver (Ellsworth, Fenwick, & Kurtz, 2004) and the approach proposed by Bettini, Crescenzi, Innocenti, Loreti, and Cecchi (2004) can assess single functions and methods in Java. The methods are executed with Java reflection classes that provide means to invoke methods based on their signature.

WebToTeach (Arnow & Barshay, 1999) and ELP (Truong, Bancroft, & Roe, 2003) provide a possibility to assess even single statements. This is achieved by combining the student code fragment to an instructor template before compiling. Scheme-robo (Saikkonen, Malmi, & Korhonen, 2001) assesses programs implemented in Scheme language. Since the language follows the functional paradigm, the interpreter can execute any function. Thus, it is possible to test partial programs without special arrangements. When assessing a function, the correctness is usually decided by the result value of the function, instead of studying the program (function) output.

Typical computer programs are no longer batch processing programs invoked from the command line, but commonly have graphical user interfaces. Hence, also programming courses have such assignments. Assessing the functionality of a program with a graphical user interface requires a means to deal with and to measure actions and responses communicated through the user interface. JEWL (English, 2004) is a language library that is designed for Java programming with graphical user interfaces. It provides students with graphical components similar to those in the standard library. At the same time, the library provides teachers a possibility to manage the events of the program with input texts and to study the output as text of the actions. Therefore, program functionality can be assessed automatically by comparing text output with certain input events, which is similar to assessing a command line program.

3.1.2. Efficiency. Automated assessment approaches for program efficiency are typically based on executing the program against different test cases and measuring program behavior during the execution. The results are often compared to an existing model solution. Thus, the success of the efficiency evaluation depends heavily on the quality of the test case design and the model solution.

A simple efficiency measurement is the running time of the program, measured either by the clock or CPU time used. The clock time is often used to ensure that program terminates after a certain time limit. Measuring CPU time for efficiency is available, for example, in Assyst (Jackson & Usher, 1997) and Online Judge (Cheang et al., 2003). However, this kind of efficiency assessment is affected by several features of the program. For example, although the goal of the assignment is to implement an efficient data structure for storing and retrieving data, the efficiency measurement can be distorted by different implementations for data input/output actions. These problems can be reduced by designing the assignments to emphasize the required issue in implementation. For example, Hansen and Ruuska (2003) solved the problem by offering students a common input/output module for use in assignments that concentrate on efficient data processing algorithms.

Efficiency can also be assessed by studying the execution behavior of different structures inside the program. Ceilidh (Foxley, 1999) and Assyst (Jackson & Usher, 1997) provide dynamic profiling for efficiency assessment. This is done by calculating how many times certain blocks and statements are executed and by comparing the results to the values obtained from the model solution.

3.1.3. Testing Skills. The goal for developing efficient automatic testing approaches is not to let students be lazy with their work. Students should learn to design test cases and test their programs thoroughly before submitting. Testing is an essential phase in program development. Teachers have noticed that when offering automatic assessment tools for students' use, they have to assure by an assessment strategy that students design test data sets and learn to test their programs by themselves (Chen, 2004; Edwards, 2004). This is typically assessed by requiring students to submit test data sets together with the programming assignment and then assessing the quality of the submitted test data.

Assyst (Jackson & Usher, 1997) was the first tool that provided assessment of student test data. The assessment was based on measuring how well the student's test data set covered all the lines in the student's own program. Chen (2004) assesses the student test suite by running a set of buggy instructor programs against it. The grade is given according to the number of buggy programs revealed by the test data. Edwards (2004) describes a system that focuses on improving students' testing skills with automatic assessment. When a student submits a test data set, it is assessed with teacher's reference solution to check its validity against problem specification and to measure how well it covers all the different execution paths. Then the functionality of the student's program is measured with this test data and these three scores are multiplied together to give the final score. Hence, a student has to develop a comprehensive, valid test data set and a correctly functioning program in order to get full score.

3.1.4. Special Features. Above presented assessment features included general goals, that are most commonly mentioned as needs for program assessment. There are also other assessment experiments, designed to answer specific problems relating to dynamic execution of programming assignments. For example, in most systems the dynamic assessment is carried out against several test data sets, and each of them is evaluated individually, starting from the initial state and completing all the processing before the assessment of the output or the return value. This kind of approach does not allow assessment in the middle of processing, i.e. defining a test case with a planned relationship to the program state created during previous test input. For example, Quiver (Ellsworth et al., 2004) provides the instructor a possibility to define state persistence between test cases for chaining different tests together.

Language specific implementation issues can be difficult to learn and assess. A good example is dynamic memory management with C++ language. For reasons relating to both language syntax and program design, students often misuse memory management functions and pointers to memory areas, and do not deallocate all the reserved memory blocks. Tutnew library provides means for assessing these issues simply by including the library to a C++ program (Rintala, 2002). The library overrides normal memory management methods and thus can provide runtime assessment for program memory usage. The assessment results are printed after the program is finished, although in case of a serious memory management error the program is terminated with an error status. This is an example of an issue that is practically impossible to evaluate by a human assessor from a complex program, but

can be efficiently traced by the computer during runtime. Naturally, the test cases affect the coverage of this assessment, since they define the execution paths for the program.

3.2. Static Assessment

In this paper, static assessment means evaluation that can be carried out by collecting information from program code without executing it. In the old days, programming assignments were often assessed only statically. Students submitted printouts of the program code and output, and teachers based their assessment on visual inspection. With portable media and, later, internet communications, the submissions have turned electronic and assessment commonly includes execution of the program. There is still place for static assessment; there is far more to a good program than correct functionality. Furthermore, static analysis may reveal functionality issues that have been left unnoticed by the limited test cases. Another benefit of static analysis is that it can be carried out even if there were problems in the dynamic behavior of the program. However, most methods rely on the formal structure of the program language, thus they require the program to be syntactically and semantically correct.

3.2.1. Coding Style. The basic requirement for automatically analyzing a computer program is correct syntax. The most effective and obvious assessor for this feature is the language compiler or interpreter. Previously, compilers were often complemented with additional tools to find structural deficiencies, e.g., Lint for C language (Darwin, 1990). Nowadays compilers and their warning capabilities are very effective and should not be forgotten by the teachers and students. For example, GCC compiler (GCC) can provide feedback on unused variables, implicit type conversions, and language features that are not following the language standards, amongst other things. These are automatic assessment features that can easily be taken in use to any programming assignment using C++ language.

In addition to the technical requirements, there are also style requirements for a good program code. Programming style or coding style and its connections to readability, maintainability etc. were intensively researched in 1980's, see e.g. Oman and Cook (1990) who defined a taxonomy for programming style for education. Rees (1982) presented an automatic analyzer for Pascal with a configurable grade scaling model and 10 different code measurements relating to the readability of code. His work has been the basis for many tools performing automatic programming style assessment, e.g., Ceilidh (Foxley, 1999) and Assyst (Jackson & Usher, 1997).

Dromey (1995) published work on coding style from a different perspective, by connecting it to the software quality attributes classified in the ISO-9126 Software Product Evaluation Standard. He connected code-level issues to the general quality factors, such as reliability, functionality, and maintainability of the program. An automatic system PASS (PASS) has been implemented to assess these issues from programs in Ada, C, and Java languages. Style++ (Ala-Mutka, Uimonen, & Järvinen, 2004) is another tool that has been developed for assessing quality factors from C++

programs. In addition to the traditional coding style issues, the tool concentrates on the object-oriented programming conventions and on the complicated features of the C++ language that often lead to errors in the program functionality. Checkstyle (Checkstyle) is open source software for checking Java programs and can be combined to several programming environments. The author was not able to find articles describing experiences on using it in educational settings, but for example Edwards (2004) has planned to combine it to their assessment system.

3.2.2. Programming Errors. Although functional errors of programs are most commonly assessed dynamically by executing the program against test data, some errors or at least suspicious code fragments can also be recognized statically. This kind of error analysis often relates to assessing programming style, e.g., to the reliability aspects of program code.

With functional languages, the functional composition and the function implementations both define the style of the program as well as have direct effects to the functionality. Michaelson (1996) discussed the basics of style measurements and implemented a tool to automatically evaluate the functional patterns from SML programs. The tool was connected to Ceilidh system and was able to recognize several typical error types caused by students' background in imperative programming. For a similar goal, Schorsch (1995) developed a tool to recognize and to give descriptive feedback on both style issues and most common logical errors in Pascal programs. His CAP tool recognizes, e.g., mistakes in updating a loop control variable or inconsistencies between a parameter type and usage.

Another interesting approach is presented by Xie and Engler (2002), who used code redundancies for detecting errors. By implementing a tool to detect idempotent operations, redundant assignments, dead code, and redundant conditionals, they were able to find several errors from the well known Linux source code. Although their work is not directly aimed at educational usage, it could be used as an automatic assistant for teachers or students to detect possibly erroneous spots in the program.

3.2.3. Software Metrics. Software metrics are considered here as general measurements that characterize computer programs. If the metrics have been recognized to measure important characteristics of program code, they can also provide a basis for evaluating and comparing programs. Furthermore, numeric metrics can be easily obtained automatically. However, for educational use, the measurements also need to be relevant for learning and/or instruction. For example, there is no sense in requiring students to submit a program that has a complexity number X, or contains Y lines of code. Hence, these measurements are often connected to and reasoned with issues relating to program style and design. For example, Mengel and Ulans (1999) collected automatically several software metrics from students' assignments. They considered the metrics as clear indicators of student performance and also possible indicators of needs for instructional development.

Halstead's (1977) metrics are commonly acknowledged measurements that are based on counting different attributes, such as the number of operators and operands in a program. A theoretical program size can be calculated by the number of overall and unique attributes, and be used to recognize unnecessarily large programs. Program size can also be measured simply by counting the number of code lines. Hung, Kwok and Chan (1993) studied different metrics with programming assignments and came to conclusion that the number of code lines was a good measurement of students' programming skill. Another well-known metrics was presented by McCabe (1976), who proposed a cyclomatic complexity measure to define the complexity of a program by its control structure. This measurement has been used for automatic analysis, e.g., in Assyst (Jackson & Usher, 1997) by using it to compare the complexity difference between students' programs and the model solution. If a student's solution is much more complex than the model solution, there is surely something questionable in the program design.

There also exists several object-oriented metrics, e.g., surveyed by Purao and Vaishnavi (2003). Although the literature doesn't provide experience reports of using these metrics in education, these can be seen to have the same possibilities as the traditional software science metrics. The connections between classes, the number of member variables etc. can provide profiling information of the program design. This information could be compared to the model solution for recognizing clearly different design solutions for further inspection or for comparing the submitted programs to each other.

3.2.4. Design. Teachers often need to assess whether submitted programs conform to given interface or structural requirements. This can be assessed automatically, e.g., by comparing the design of the submitted program to the problem specification, teacher's model solution or to the set of applicable solutions.

Thorburn and Rowe (1997) implemented a system that automatically recognizes the functional structure of a C program. They call it the ''solution plan'' of the program and compare it to the solution plan of the model program, or to a set of possible plans. The equivalence of different functions is determined by comparing function outputs with a randomized set of inputs. The tool also has effectively localized errors in the student programs by noticing misplaced function calls. Truong, Roe and Bancroft (2004) implemented a structural similarity analysis that transforms a student's program to XML presentation and compares it to the set of model solutions. The approach is aimed at simple introductory programs in Java language. Similarly, Scheme-robo (Saikkonen et al., 2001) assesses whether students are following the structural requirements by abstracting a structural tree from the given Scheme function and comparing it to the required structure. MacNish (2000) used the reflection facilities of java.lang.reflect package for analyzing whether the class interfaces and method signatures in students' Java programs met the given requirements.

In addition to the completely automatic assessment tools, there are tools that produce measurement values and diagrams to assist the teacher in the final assessment

of the assignment. Especially software engineering research has produced interesting ideas and tools that could also be used for education. For example, Antoniol, Casazza, Di Penta, and Fiuten (2001) present a tool that recognizes common design patterns from either program code or UML design specification. This tool could be used to help in verifying that a student's program implementation matches the design document or the design pattern requirements given by the teacher.

3.2.5. Special Features. In addition to the more general issues, teachers sometimes need to search for a certain word or expression in the program code. This can be used, e.g., to test whether a student has used the required programming language structure or a certain library function. This kind of keyword search is implemented in Scheme-Robo (Saikkonen et al., 2001). In Scheme language, this feature can be used, e.g., to assess whether program structure is purely functional by searching for primitives `set!`, `set-car!`, and `set-cdr!`. A more flexible approach has been implemented in Ceilidh (Foxley, 1999) by defining regular expressions to be searched from the student's program code.

Plagiarism has always been a problem with programming assignments, since computer programs are text files that are easy to copy. Several automated tools have been implemented for program comparison purposes, and e.g. Online Judge (Cheang et al., 2003), CourseMarker (Higgins et al., 2003) and BOSS (Luck & Joy, 1999) include tools for detecting plagiarism. Verco and Wise (1996) compared automated tools based on attribute counting mechanisms as opposed to systems that utilize structural information from the program. The attribute counting approaches effectively recognized copies that were very close to each other, but generally the structural approach was more effective. Structural information is included in the methods of MOSS (MOSS) and JPLAG (JPLAG), the well-known plagiarism detection services of today. MOSS is based on document fingerprinting (Schleimer, Wilkerson, & Aiken, 2003) and JPLAG uses string tokenization with sub-string pattern matching (Prechelt, Malpohl, & Philippsen, 2002).

4. USING AUTOMATED ASSESSMENT FOR PROGRAMMING ASSIGNMENTS IN EDUCATION

CAA is often considered as means for administrating coursework submission and grades on programming courses. Although CAA can also be much more, this is a practical need for many programming teachers, and it is provided by many of the versatile assessment systems, e.g., CourseMarker (Higgins et al., 2003), Assyst (Jackson & Usher, 1999), Online Judge (Cheang et al., 2003), and BOSS (Luck & Joy, 1999). Effective administration provides teachers with a means to easily follow students' progress and to quickly recognize needs for improvements on the course.

Automated administration also supports the organization of the marking process, for example peer-reviewing, where students comment on each others' programs. In this kind of a process, students both learn to evaluate programs better and receive more feedback than would be given by instructors only. Successful peer-reviewing

approaches for programming assignments have been reported e.g. by Zeller (2000) and Sitthiworachart and Joy (2003). An innovative organizational approach was described by Ellsworth et al. (2004) who use the Quiver system so that software engineering students define interfaces and tests for methods and classes, and the students on programming courses do the implementation. In this way, both groups learn issues relating to specifying, implementing and testing software with the help of an automatic assessment system.

4.1. Semiautomatic vs. Automatic Assessment

Teachers often agree that it is not possible to assess automatically all the issues relating to good programming. On the other hand, although the quality of the automatic feedback may not be as high as if given by an instructor, it is at least partly compensated by its speed and availability. One educational approach is to settle for assessing only those issues that can be fully automated. This suits for small assignments, where the main goal is to teach students the basics of programming language and program construction. With larger assignments, a more common approach seems to be to combine manual and automatic assessment. If the assessment process and information management is well supported by tools, this kind of semiautomatic approach can be very efficient. It helps with the burden of simpler assessment and management issues, and gives teachers more time to concentrate on the demanding assessment tasks, e.g., giving feedback on the program design. It also provides a possibility to double check the results of the automatic assessment in case there is a possibility for erroneous assessment results.

Jackson's (2000) semiautomatic approach was based on a continuous interaction between the assessment system and the instructor. The system prompted questions and processed test cases etc. according to the answers given by the instructor, who was responsible for deciding each grade. BOSS (Luck & Joy, 1999) and Online Judge (Cheang et al., 2003) assess program functionality automatically in the submission phase, so that teachers need not spend their time in waiting for the program to compile and run against test cases. Afterwards, the systems support manual commenting and marking so that tutors can insert or change the assessment information to the automatically generated assessment results. Tutors assess manually, e.g., program style and maintainability. The author has been using an approach (Ala-Mutka & Järvinen, 2004) where several aspects are assessed automatically to meet the minimum criteria before accepting submissions. After passing the submission check, the assignments are assessed by a tutor. In this kind of approach, the final assessment results provide comprehensive feedback and the work is automatically checked to meet the minimum requirements on all aspects.

4.2. Formative vs. Summative Assessment

Some of the automatic assessment tools are designed mainly for summative assessment, e.g., BOSS (Luck & Joy, 1999) while others show the student the

results of the automatic assessment and allow resubmissions, if the student is not satisfied with the results, e.g. CourseMarker (Higgins et al., 2003). The latter approach can be seen as formative assessment, since its role is to help students by providing feedback on their work and let them improve it accordingly. Interestingly, although many approaches emphasize the possibility for iteration, they still expect the students to submit a complete version of the program already on the first submission. Edwards (2004) seems to be the only developer, who has designed an iterative process into the assignments; in his system the students can submit their program although it is not complete. Since the program is assessed against student's own test data, these can be developed at the same pace and student can anytime check how close he is in implementing and testing all the behaviors defined in the problem specification.

Automated assessment can be used for summative purposes both in homework assignments and in controlled programming situations, so called online examinations. These provide a means to assure students' personal programming skills, since there is less possibilities for cheating. English (2002) used an examination situation, where students had to debug and write small assignments with the help of a computer as a part of the exam. The programming assignments were assessed automatically, but problematic cases were flagged for manual inspection. However, it has been noticed that students need practice to work successfully in an online examination situation (Woit & Mason, 2003). The final online examination results are better, when the students have gained experience of an online exam situation previously on the course.

Several authors have reported how assessment tools with resubmission possibilities have been provided for students to help them develop their programming assignments (Ala-Mutka & Järvinen, 2004; Chen, 2004; Edwards, 2004; Foxley, 1999). This reduces the assessment workload of the teacher, since students do the assessment work at least partly by themselves. Automatic feedback also guides students with their work, giving more individual guidance than otherwise would be possible on a large course. The form of the feedback depends on the tool, it can be for example hints of the errors in the program, highlights of erroneous code fragments or listings of the test input data and expected output. The type of the feedback naturally affects the working strategy of the students. For example Chen (2004), has selected an approach that does not to give too detailed feedback on the errors, so that students learn to debug their programs themselves.

Tutoring programs are a special type of automatic assessment tools, since they are mainly developed to be used as learning support. These are usually not meant for official assessment purposes but for guiding the students to learn basic programming skills and problem-solving in introductory courses. ELP (Truong, Bancroft, & Roe, 2003) is a simple environment that combines automatic assessment, learning materials, and program development for supporting the learning of introductory programming. Intelligent tutors may use their own languages or programming interfaces, and often provide an environment, where students' actions can be monitored and guided. Hong (2004) presents a Prolog tutor that is based on recognizing several programming techniques which are used for feedback and error

analysis. Pillay (2003) provides a brief overview of the subject and suggests a general composition for the architecture of intelligent programming tutors.

5. DISCUSSION

The undeniable benefits of automated assessment include objectivity, consistency, speed and 24h availability. Some features of the programs may even be better evaluated by automated approaches than by a human eye, e.g., program execution or systematic checking of simple style rules. Furthermore, automation saves a lot of human work and time in tasks such as program compilation and running the test cases. However, the simplicity of getting a grade from an automatic tool is also dangerous. Teachers may become tempted to incorporate new assessment tools to the courses just because they consider them to produce relevant assessment information without increasing their workload. It has to be remembered that students need more than just assessment feedback and grades for learning. If new, moreover automatic, assessment is incorporated into courses and affects the students, the relevance of these issues to programming should to be clearly justified for the students. Failing to do so may lead to misunderstandings and even cheating (Ala-Mutka et al., 2004).

By relieving the teachers' burden of assessment work, automated assessment offers possibilities to make students program often, as opposed to having only few assignments. This is often recognized as a good practice for ensuring students' learning of programming (Woit & Mason, 2003). However, as discussed in Section 2, the assignment design and the problem setting have a profound effect to the level of cognitive skills and to the understanding of software quality issues that are developed in the task. Careful assignment design and efficient strategies for utilizing automated assessment could be used to construct a course, where several assessed programming tasks provide students with a strong and versatile practical programming experience.

Teachers use automatic assessment in different ways applying it to the needs and practices of the programming course in question. Assessment tools can be applied to specific features, or combined to a larger system with submission and program development facilities. It is important to remember that if students are expected to learn to use standard program development tools, e.g. compiler, editor, or debugger, a combined submission-assessment system may provide them with an environment where they are not necessarily required to learn these. This issue was mentioned, e.g., by Luck and Joy (1999) and also noticed personally by the author from a local student, who had (without plagiarism) submitted successfully two small programming assignments in C++, but did not know how to use a compiler.

Tool developers have noticed that students' learning culture changed when they were given an access to the assessment tools with a resubmission possibility, e.g. Ala-Mutka and Järvinen (2004), Chen (2004), Edwards (2004), Foxley, (1999). The changes are positive in the sense that students become better aware of the quality of their program. However, dishonesty can also occur when students have possibilities to see what is measured by the automatic tools. For example, the author has noticed that students may try to distract automated style assessment by inserting comments

that make no sense — although with the same work effort they could write useful comments. Educators have also noticed that students easily begin to rely on the automated testing and do less testing on their own. Ceilidh provided two technical solutions for fighting this problem; a possibility to limit the maximum number of submissions, or to set a minimum waiting time between two submissions. Edwards had experienced that this kind of limitations affected negatively students submission behavior. After changing the assignment design to emphasize test-driven development of the program and allowing unlimited resubmissions students began working earlier and learned to develop test data together with the program.

There are few systematic studies of the effects that automatic assessment tools and feedback have on students' learning results and working strategies. Therefore, it is difficult to say, how efficient these approaches really are for learning. Reported experiences suggest that the quality of the students' work often improves at least enough to reduce teachers' assessment needs on simpler issues (Ala-Mutka & Järvinen, 2004; Edwards, 2004; Foxley, 1999; Schorsch 1995). Well-designed assignments and automated assessment practices can also guide students' working strategies, as shown by Edwards. However, very little research has been published of how well these skills have been transferred to students' normal programming practices, i.e. to later courses or to working life.

5.1. Technology

Programming courses are well-positioned to develop and utilize automated assessment when considering the persons involved. Teachers are experienced with computers and capable of tailoring tools to support their teaching goals and course practices. Students, on the other hand, work on the computers continuously and are better prepared to work with automated feedback than students from a less computer-related field. However, one must not assume that it is easy to make use of automatic assessment tools. Creating automatically assessable programming assignments requires special attention, since even small mistakes in the marking definitions can cause problems. No ambiguities are allowed in the problem specification, especially when considering input/output formats, if the assessment tool cannot be set to filter them. Also the test cases need to be thoroughly designed, to reveal all the deficiencies as comprehensively as possible. Although these tasks are more burdensome than with traditional assessment, they also make teachers think more carefully the goals of the assignment. Morris (2004) gives a good description of the systematic practices required for developing assignments with automatic assessment.

An assessment tool or a system implements typically the assessment features that have been considered important by the tool designer. CourseMarker (Higgins et al., 2003) / Ceilidh (Foxley, 1999) is an exception in that it provides teachers a possibility to easily select the dynamic and static assessment tools to be used in each exercise. The assessment principles and the weightings of the different features in the total grade can be adjusted with configuration files. Furthermore, the system provides a possibility to use external marking tools. These features make it easy to configure the

assessment according to the needs of different courses. However, CourseMarker/Ceilidh has some fixed assumptions of assessment procedures that can make it difficult to use in the coursework administration and submission facilities in different settings. This was also noticed by the author and has been mentioned by Luck and Joy (1999). As always, when packing many issues into one large system, it is not as easily portable to other environments as smaller tools. For static analysis purposes, Truong, Bancroft, & Roe, (2004) state that their framework for gap filling programming exercises can be easily configured and extended with new analyses.

If universities used more similar tool approaches, good assignments and their assessment routines could be stored and reused in co-operation, even to create common assignment banks. Unfortunately, the systems today are in-house built and no common standards or interfaces exist. IMS Question & Test Interoperability specification (IMS, 2003) is a well-known effort developed for standardizing assessment descriptions for learning management systems. The specification concentrates on simple assessment techniques, but also provides facilities for creating proprietary scoring algorithms and passing information through parameters to the assessment engine. However, no examples or experiences have been published of using this feature for tasks requiring special execution environments etc., as would be the case with computer programs. Although not relating to the IMS QTI, recent approaches using XML descriptions for specifying simple program assessment (Bettini et al., 2004; Truong et al., 2003) open a promising area. Similar ideas could be applied to developing common interface definitions for describing inputs, outputs, and assessment methodologies for programming assignments.

Considering the skills and professional background of many computing educators, the development work for a common assessment system could be supported by an open source project similar to Linux. After defining the basic core of a program assessment system and interfaces for integrating assessment tools for it, developers could create specific assessment tools to suit their needs. Teachers would have a selection of tools to choose from, and a possibility to improve or otherwise modify them. The result would be a versatile, portable assessment system continuously developed and improved by the active practitioners of the field. BOSS (Joy & Griffiths, 2004) and Quiver (Ellsworth, 2004) are recently released as open source and could provide ideas and starting points for this kind of development work.

6. CONCLUSION

As formally defined structures, computer programs are amenable for automated measurement. Several automatic assessment approaches have been reported in the literature for both dynamic and static assessment of programs. Automated assessment is often recognized to offer faster, more consistent, absolutely objective, and tireless marking and feedback support for both teachers and students. However, all the aspects relating to programming cannot be assessed automatically. Semiautomatic approaches can be used to measure some issues automatically while leaving other issues for manual inspection.

Once good tools have been found or developed, the teacher should consider carefully how to incorporate them into education. The assessment needs to be educationally sound to the course context and not only inspired by the technology. Different assignment designs and assessment settings can provide versatile learning experiences even if using automatic tools for the actual assessment. Automatic assessment tools can also be offered for students to support their program construction process and self-assessment. However, easily available automated assessment can also have a negative effect on students' working strategies, e.g., on testing the programs. The teacher should prepare the assignment and the assessment strategies so that the students are encouraged to learn and work on their program on all desired aspects.

Unfortunately, many of the present assessment tools are developed for a local use and only for a certain type of assignments. Hence, they are often not available for a wider use and would be difficult to adopt to another university, anyway. Several different tools also make it impossible to develop common assignment and assessment configuration banks for promoting reuse and material sharing between developers. Developing interoperable tool approaches would offer new and concrete co-operation possibilities for teachers in different universities for sharing knowledge of good assignments and educational approaches in automated assessment.

REFERENCES

Ala-Mutka, K., & Järvinen, H.-M. (2004). Assessment Process for Programming Assignments. *Proceedings of the 4th IEEE International Conference on Advanced Learning Technologies*, Finland, 181 – 185.

Ala-Mutka, K., Uimonen, T., & Järvinen, H.-M. (2004). Supporting students In C + + Programming Courses with Automatic Program Style Assessment. *Journal of Information Technology Education*, *3*, 245 – 262.

Antoniol, G., Casazza, G., Di Penta, M., & Fiuten, R. (2001). Object-oriented design patterns recovery. *Journal of Systems and Software*, *59*, 181 – 196.

Arnow, D., & Barshay, O. (1999). WebToTeach: An interactive focused programming exercise system. *Proceedings of the 29th ASEE/IEEE Frontiers in Education Conference*, 12A9/39 – 12A9/44.

Becker, K. (2003). Grading programming assignments using rubrics. In *Proceedings of the 8th annual conference on Innovation and technology in computer science education*, Greece, 253.

Bettini, L., Crescenzi, P., Innocenti, G., Loreti, M., & Cecchi, L. (2004). An Environment for Self-Assessing Java Programming Skills in Undergraduate First Programming Courses. In *Proceedings of the 4th IEEE International Conference on Advanced Learning Technologies*, Finland, 161 – 165.

Bloom, B.S. (Ed.) (1956). *Taxonomy of Educational Objectives: The Classification of Educational Goals. Handbook I. Cognitive Domain*. New York: David McKay Company, Inc.

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

Cheang, B., Kurnia, A., Lim, A., & Oon, W.-C. (2003). On automated grading of Programming Assignments in an academic institution. *Computers & Education*, *41*, 121 – 131.

Checkstyle. [Computer Software] Retrieved November 15, 2004, from http://checkstyle.sourceforge.net/.

Chen, P. (2004). An Automated Feedback System for Computer Organization Projects. *IEEE Transactions on Education*, *47*, 232 – 240.

Cox, K., & Clark, D. (1998). The Use of Formative Quizzes for Deep Learning. *Computers & Education*, *30*, 157 – 167.

Darwin, I. (1990). *Checking C Programs with Lint*. New York: OReilly & Associates.

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

Edwards, S. (2004). Improving Student Performance by Evaluating How Well Students Test Their Own Programs. *ACM Journal of Educational Resources in Computing*, *3*(3), 1 – 24.

Ellsworth, C., Fenwick, J., & Kurtz, B. (2004). The Quiver System. In *Proceedings of the 35th SIGCSE technical symposium on Computer Science Education*, US, 205 – 209.

English, J. (2004). Automatic Assessment of GUI Programs using JEWL. In *Proceedings of 9th annual conference on Innovation and technology in computer science education*, UK, 137 – 141.

English, J. (2002). Experience with a computer-assisted formal programming examination. In *Proceedings of 7th annual conference on Innovation and technology in computer science education*, Denmark, 51 – 54.

Foxley, E. (1999). *Ceilidh Documentation on the World Wide Web*. Retrieved November 15, 2004, from http://www.cs.nott.ac.uk/ ~ ceilidh/papers.html.

GCC compiler. [Computer software] Retrieved November 15, 2004, from http://gcc.gnu.org/.

Halstead, M.H. (1977). *Elements of Software Science*. New York: Elsevier North-Holland.

Hansen, H., & Ruuska, M. (2003). Assessing time-efficiency in a course on data structures and algorithms. In *Proceedings of the 3rd Annual Finnish/Baltic Sea Conference on Computer Science Education*, Finland.

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

Higgins, C., Symeonidis, P., & Tsinsifas, A. (2002). Diagram-based CBA using DATsys and CourseMaster. In *Proceedings of the International Conference on Computers in Education*, 367 – 372.

Howles, T. (2003). Fostering the growth of a software quality culture. *ACM SIGCSE Bulletin*, *35*(2), 45 – 47.

Hong, J. (2004). Guided Programming and Automated Error Analysis in an Intelligent Prolog Tutor. *International Journal of Human-Computer Studies*, *61*, 505 – 534.

Hung, S.-L., Kwok, L.-F., & Chan, R. (1993). Automatic programming assessment. *Computers & Education*, *20*, 183 – 190.

IMS Global Learning Consortium. (2003). *IMS Question & Test interoperability specification* (Version 1.2.1). Retrieved November 15, 2004, from http:// www.imsproject.org/question/.

Jackson, D. (2000). A semi-automated approach to online assessment. *Proceedings of 5th annual conference on Innovation and technology in computer science education*, Finland, 164 – 167.

Jackson, D., & Usher, M. (1997). Grading Student Programs using ASSYST. *Proceedings of the 28th SIGCSE technical symposium on Computer science education*, USA, 335 – 339.

Joy, M. & Griffiths, N. (2004). Online Submission of Coursework – a Technological Perspective. In *Proceedings of the 4th IEEE International Conference on Advanced Learning Technologies*, Finland, 430 – 434.

JPLAG. [Computer software] Retrieved November 15, 2004, from http://www.jplag.de/.

Kay, D., Scott, T., Isaacson, P., & Reek, K. (1994). Automated grading assistance for student programs. In *Proceedings of the 25th SIGCSE technical symposium on Computer science education*, USA, 381 – 382.

Korhonen, A., & Malmi, L. (2000). Algorithm simulation with automatic assessment. *Proceedings of 5th annual conference on Innovation and technology in computer science education*, Finland, 160 – 163.

Lister, R. and Leaney, J. (2003). First Year Programming: Let All the Flowers Bloom. In T. Greening & R. Lister (Eds.), *Computing Education 2003 Fifth Australasian Computing Education Conference* (pp. 221 – 230).

Luck, M., & Joy, M. (1999). A Secure On-line Submission System, *Software – Practice and Experience*, *29*, 721 – 740.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y., Laxer, G., Thomas, L., Utting, I., & Wilusz, T. (2001). A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students, *ACM SIGCSE Bulletin*, *33*(4), 125 – 180.

MacNish, C. (2000). Java facilities for automating analysis, feedback and assessment of laboratory work. *Computer Science Education*, *10*, 147 – 163.

McCabe, T.J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, *SE-2*, 308 – 320.

Mengel, S.A., & Ulans, J.V. (1999). A case study of the analysis of the quality of novice students programs. In *Proceedings of the 12th Conference on Software Engineering Education and Training*, 40 – 49.

Michaelson, G. (1996). Automatic Analysis of Functional Program Style. In *Proceedings of Australian Software Engineering Conference*, 38 – 46.

Morris, D. (2003). Automatic Grading of Student's Programming Assignments: An Interactive Process and Suite of Programs. In *Proceedings of the 33rd ASEE/IEEE Frontiers in Education Conference*, S3F-1 – S3F-5.

MOSS. [Computer software] Retrieved November 15, 2004, from http://www.cs.berkeley.edu/~aiken/moss.html.

Olson, D.M. (1988). The reliability of analytic and holistic methods in rating students' computer programs. In *Proceedings of the 19th SIGCSE technical symposium on Computer science education*, USA, 293 – 298.

Oman, P.W., & Cook, C.R. (1990). A taxonomy for programming style. In *Proceedings of the 1990 ACM Annual Conference on Cooperation*, 244 – 250.

PASS Program Analysis and Style System. Retrieved November 15, 2004, from http://www3.sqi.gu.edu.au/pass/.

Pillay, N. (2003). Developing intelligent programming tutors for novice programmers. *ACM SIGCSE Bulletin*, *35*(2), 78 – 82.

Prechelt, L., Malpohl, G., & Philippsen, M. (2002). Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, *8*, 1016 – 1038.

Purao, S., & Vaishnavi, V. (2003). Product metrics for object-oriented systems. *ACM Computing Surveys*, *35*, 191 – 221.

Reek, K.A. (1989). The TRY system -or- how to avoid testing student programs. In *Proceedings of the 20th SIGCSE technical symposium on Computer science education*, USA, 112 – 116.

Rees, M.J. (1982). Automatic assessment aid for pascal programs. *SIGPLAN Notices*, *17*, 33 – 42.

Rintala, M. (2002). *Tutnew memory management library*. Retrieved November 15, 2004, from http://www.cs.tut.fi/~bitti/tutnew/english/.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion, *Computer Science Education*, *13*, 137 – 172.

Ruehr, F., & Orr, G. (2002). Interactive program demonstration as a form of student program assessment. *Journal of Computing in Small Colleges*, *18*, 65 – 78.

Saikkonen, R., Malmi, L., & Korhonen, A. (2001). Fully automatic assessment of programming exercises. *Proceedings of 6th annual conference on Innovation and technology in computer science education*, UK, 133 – 136.

Schleimer, S., Wilkerson, D., & Aiken, A. (2003). Winnowing: Local algorithms for document fingerprinting. *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, USA, 76 – 85.

Schorsch, T. (1995). CAP: an automated self-assessment tool to check Pascal programs for syntax, logic and style errors. In *Proceedings of the 26th SIGCSE technical symposium on Computer science education*, 168 – 172.

Sitthiworachart, J., & Joy, M. (2003). Effective Peer Assessment for Learning Computer Programming. *Proceedings of the 8th annual conference on Innovation and technology in computer science education*, UK, 122 – 126.

Thorburn, G., & Rowe, G. (1997). PASS: An automated system for program assessment. *Computers & Education*, *29*, 195 – 206.

Truong, N., Roe, P., & Bancroft, P. (2003). A Web Based Environment for Learning to Program. In *Proceedings of the 25th Australasian Computer Science Conference*, Australia, 255 – 264.

Truong, N., Roe, P., & Bancroft, P. (2004). Static Analysis of Students' Java Programs. In *Proceedings of the sixth Australian Computing Education Conference*, New Zealand, 317 – 325.

Verco, K., & Wise. M. (1996). A comparison of automated systems for detecting suspected plagiarism. *The Computer Journal*, *39*, 741 – 750.

Woit, D., & Mason, D. (2003). Effectiveness of online assessment. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, USA, 137 – 141.

Xie, Y., & Engler, D. (2002). Using Redundancies to Find Errors. In *Proceedings of SIGSOFT 2002/ FSE-10*, USA, 51 – 60.

Zeller, A. (2000). Making students read and review code. In *Proceedings of 5th annual conference on Innovation and technology in computer science education*, Finland, 89 – 92.