A Semi-Automated Approach to Online Assessment

David Jackson
Dept. of Computer Science
University of Liverpool
Chadwick Building, Peach Street
Liverpool L69 7ZF
United Kingdom
d.jackson@csc.liv.ac.uk

Abstract

Desirable though fully automated assessment of student programming assignments is, it is an area that is beset by difficulties. While it is not contested that some aspects of assessment can be performed much more efficiently and accurately by computer, there are many others that still require human involvement. We have therefore designed a system that combines the strengths of the two approaches, the assessment software calling upon the skills of the human tutor where necessary to make sensible judgements. The technique has been used successfully on a systems programming course for several years, and student feedback has been supportive.

1 Introduction

Recent trends towards teaching large numbers of students at low cost have led a number of researchers to examine ways in which the assessment of student assignments could be automated [1,3-6]. In particular, courses on computer programming would seem a natural testing ground for computer-based marking. It transpires, however, that the area is a minefield of problems. Existing systems tend to concentrate on those aspects of assessment that are straightforward to perform, whilst ignoring those that are difficult. For example, measuring the execution time of a student's program is a trivial operation that is easily automated; on the other hand, attempting to interpret the content of comments within the code is a task that is well beyond current capabilities. By focusing on the former at the expense of the latter, we are in danger of producing appraisals that are, at best, skewed and incomplete, and, at worst, totally unrelated to the work put in by the student.

ITiCSE 2000 7/00 Helsinki, Finland

To illustrate the problem further, consider the job of automatically validating a student's program. Mechanising the process of applying the program to a number of sets of test data poses few problems. The difficulty lies in devising a foolproof approach to judging whether the outputs produced by that program are correct. If a problem specification is to be at all open to interpretation – and this is often impossible to avoid – then it is unlikely that any two solutions to that problem will generate precisely the same output. This is especially true of assignments along the lines of 'Write a program to generate a table of monthly rainfall,' but it can also be true of more tightly specified exercises such as 'Write a program to calculate the value of pi to three decimal places.' Even assuming that an automatic assessor is sophisticated enough not to be distracted by minor deviations in formatting and layout, the student who precedes the answer with 'The value of pi is...' might give it more of a headache. Of course, a tutor could go to the extreme of making the problem specification so detailed that it becomes completely unambiguous, but then what room does that leave for innovation on the part of the student?

Such problem areas have been highlighted before, and a number of innovative approaches have been suggested to help overcome them [7]. Indeed, there are several aspects of assessment that can be performed much more accurately and efficiently by computer, and that can lead to more uniform marking across large groups of students. The fact remains, however, that there are still some aspects of assessment that are best performed by human tutors. Perhaps, then, the ideal solution is an approach which combines the talents of human and machine.

2 Semi-Automated Assessment

The idea of combining forces in this way has led us to develop a software system that takes responsibility for the more mechanical aspects of assessment, and which prompts

the human tutor as necessary for more advanced judgements. The system is implemented as a UNIX shell script. The main tasks it performs are:

- *Preparing submissions* – Student programming solutions are submitted by e-mail. The system prepares these for subsequent analysis by stripping e-mail headers, confirming the presence of expected files, and unpacking archives.

- *Compilation and building* – Students are normally asked to submit a UNIX makefile as part of their solution to an assignment. One of the system's jobs is to run this in order to build an executable program. It also has to act in a sensible way in cases where the makefile has been omitted.

- *Testing* – Despite the difficulties mentioned earlier, there are still many operations associated with the testing phase for which automation is straightforward (and therefore desirable). These include running the student executables over selected test data, collating output during those runs, and maintaining records of tests passed and failed. The need for human input cannot be obviated entirely, however, and as we shall see, it is during this testing phase that interaction between the system and the tutor is at its most intense.

- *Style analysis* – To an extent it is also possible to automate the analysis of programming style. Here, the system employs a number of metrics to assess things such as modularity, level of commenting, use of indentation, and so on.

- *Report generation* – All of this analysis is of little use if it cannot be properly summarised and presented. Another of the system's tasks, therefore, is to prepare and forward appropriate reports for the benefit of both the tutor and the student.

Where the system looks to the human tutor for assistance is for evaluation of:

- *Documentation* – Artificial Intelligence techniques are not yet advanced to the point of being able to make qualitative assessments of items such as README files and other supporting documentation. When necessary, the system presents the document and prompts the tutor for a view; this usually takes the form of a multiple choice classification.

- *Testing* – As mentioned above, the system can do much of the housekeeping work surrounding the testing procedure, but it is still reliant on the human tutor to handle the many cases of uncertainty that crop up.

- *Source Code* – There are many forms of analysis of source code that could be automated with ease, but the opportunity to cast a human eye over a program is still to be valued. Checking, for example, that the programming techniques specified in the problem have been employed, and that dubious practices have been avoided, is still best done by a human.

Appreciating the interplay between human and computer during the running of the assessment system is perhaps best illustrated by studying its application to a real student program.

3 A Sample Session

Here we present and narrate an abridged transcript obtained when marking a student's solution to a real exercise. The class was asked to write a version of the UNIX 'head' command (to be called 'fathead'), and then to submit it electronically as part of an archive that also contained a README file and a makefile for rebuilding the executable. The session proceeded as follows:

```
$ mark44 studentX
            2CS44 EX. 1
      REPORT ON studentX ("John Doe")


CHECKING SHAR FILE...
   <Start of submitted file displayed here>

HAVE -Cce OPTIONS BEEN USED?
```

The command to run the assessment system is 'mark44' followed by the username of the student. After some initialisation that includes looking up the real name of the student and copying all the necessary files into a test area, the system's first task is to check that the solution has been submitted in the required manner. Students are instructed to use the UNIX 'shar' command to bundle together their C files, a README file, and a makefile. They are also required to specify a number of options to the shar command (-Cce). The system therefore displays the first few lines of the submission, allowing the tutor to check that shar has been used correctly. In this particular case it is clear that not all of the –Cce options have been used, and the tutor will therefore respond negatively to the question. The reason for asking the tutor to make this check rather than automating it is that students quite often send all kinds of wrong files by mistake, including binary executables that could confound the system. In this case the error is not so serious, and in reply to the next question:

```
   Is file otherwise OK (y,n)?
```

the tutor can answer yes.

```
STRIPPING MAILER HEADER...

UNPACKING ARCHIVE...

     x - fathead.c
     x - README
     x - makefile


CHECKING SUBMITTED FILES...

README OK

MAKEFILE PRESENT

RUNNING MAKE...
   cc -Aa fathead.c -o fathead
MAKE RAN OK
FATHEAD CREATED


GETTING MARKS FOR README file...
Hit return to view
```

The system now strips off the header information inserted by the mailer, unpacks the archive, and confirms that all the expected files are present. Students are told what names to give their files; if they overlook this requirement, the system will give the tutor the opportunity to rename the files that have been sent. It then runs the student's makefile to build the executable. Students are asked to write a makefile that does not require any arguments. If, for any reason, this does not work, the system will try again with the name of the target as an argument; if this too fails, the system will try to build the system from scratch using an appropriate sequence of compile and link commands. An inability to proceed beyond this point is usually due to fatal compilation errors, meaning that the student has not even got as far as producing a working program.

Assuming that the executable can be generated, the system then goes on to consider the README file. There is little it can do here other than to present the contents of the file to the tutor.

```
Choose a category for the README file:

   1) awful
   2) poor
   3) fair
   4) good
   5) very good
```

Once the tutor has had a chance to study the README file, the system asks for an assessment. To simplify things, the tutor is asked to select one from the five categories shown.

```
***TEST 1... (simple call)
fathead f1

MODEL OUTPUT
   <Output of model solution displayed here>

STUDENT OUTPUT
   <Output of student solution displayed here>

DIFFERENCES
   0a1;2
   > ==> data1 <==
   >
   9a12
   >
ARE THESE THE SAME? n
TEST FAILED
```

The system is now in a position to begin black-box testing of the student's code. As mentioned earlier, making the automation of this foolproof is notoriously difficult, and it is for this reason that great emphasis has been placed on dividing the responsibility for the task between the assessment system and the human tutor. The system's role is to execute the student program a number of times on selected test data, redirecting the program output into a temporary work file which can then be compared with an 'ideal' output file. The tests are not made using random test data; rather, each set of data is designed to evaluate key properties of the student's program. The ideal file can either be prepared in advance by the tutor, or simply generated on the fly from a model solution. If the system finds no differences between the student program output and the expected output, it will accept that the test has been passed. If, on the other hand, it detects variations, no matter how slight, it calls upon the human tutor to act as the final arbiter.

In the listing above, the program is being given its first, most basic test. This simply applies it to a single file, without any command options. The system presents the output from the student program, followed by the output that should have been produced, and then summarises the differences between the two by making use of the UNIX 'diff' comand. This makes it much easier for the tutor to spot where the student's solution has deviated from the model solution, and to make a decision as to whether the difference is serious enough to warrant deducting marks. In this example, the student has preceded the command output with the name of the file given as its argument. Since the problem specification clearly stated that filenames should only be given in the case of multiple arguments, the tutor decides that the deviation is significant and the student is deemed to have failed that particular test.

The testing phase continues in this manner for a number of executions, using a variety of test data and command options. Additional testing is then performed to check that

the student has thought about possible error conditions that may arise. This bank of tests works differently from the previous ones in that there is no comparison of output. Instead, the system simply checks that an appropriate error message is produced in each case.

```
C FILE IS fathead.c
PRESS RETURN TO VIEW SOURCE
```

Once the testing phase is over, the system gives the tutor the opportunity to view the student's source code. It is at this point that the tutor can perform aspects of code quality assessment that are difficult or impossible to automate. The tutor may wish to check, for example, that comments are meaningful and that appropriate algorithms have been used. If necessary, the tutor can also elect to impose penalties at this stage.

```
APPLYING STYLE METRICS...

 12.7  characters per line     :   9.0
       (max   9)
  4.0  % comment lines         :   0.0
       (max  12)
 20.9  % indentation           :   9.7
       (max  12)
 38.5  % blank lines           :   0.0
       (max  11)
  1.6    spaces per line       :   1.5
       (max   8)
  4.2    module length         :   0.5
       (max  15)
 17.0    reserved words        :   6.0
       (max   6)
  4.2    identifier length     :   2.8
       (max  14)
  0.0    gotos                 :   0.0
       (max -20)
  5.0    include files         :   5.0
       (max   5)
  0.0  % defines               :   0.0
       (max   8)

Score  34.5
Style mark = 34
```

The final form of assessment takes the form of an analysis of the student's programming style. For this, the style metrics proposed by Berry and Meekings [2] have been adopted. As can be seen from the results above, this particular student does not score well on style, gaining a mark of only 34%.

```
FINAL SCORE = 65 (B)
Tidying up...
```

The scores from all the stages described above are ultimately combined to give a final mark and grade. Following this, a tidying up procedure ensues in which temporary files are deleted, a database of class marks is updated, and a final report is prepared for the student's consumption. When all students have been assessed in this way, a further script is run which will send reports to individual students via e-mail.

4 Conclusions

The system described here has been used successfully on a second year course on systems programming for several years, and across a range of exercises. Although the need for the human to do some work has not been entirely eliminated, assignments can be marked and returned to students very much faster than they were in the days when they were marked by hand. Moreover, the amount of analysis, especially with regard to testing, has been substantially increased using this approach: it is rare that other course tutors using traditional marking methods ever test student solutions online. A further advantage of using the system is that both the marking process itself and the reporting that is made back to students remain consistent across a large class. Questionnaire returns confirm that, while students would be less happy with a fully automated marking system, they are generally satisfied with the composite approach. Further enhancements planned for the future include building in analyses of complexity and efficiency, and improving the portability of the system to different forms of assignment.

References

[1] Benford, S., Burke, E. and Foxley, E. Courseware to Support the Teaching of Programming. *Proc. Conf. Developments in the Teaching of Computer Science* (1992), Univ. of Kent at Canterbury, 158-166

[2] Berry, R.E. and Meekings, B.A.E. A Style Analysis of C Programs. *Comm. ACM, 28(1)* (1985), 80-88

[3] Hung, S-L., Kwok, L-F. and Chan, R. Automatic Programming Assessment. *Computers and Education (Pergamon), 20(2)* (1993), 183-190

[4] Jackson, D. A Software System for Grading Student Computer Programs. *Computers and Education (Pergamon), 27(3/4)* (1996), 171-180

[5] Jackson, D. Computer-Based Evaluation of Student Software Quality. *Proc. 2nd Conf. Software Engineering in Higher Education (SEHE92)* (1992), Southampton, UK, 93-104

[6] Jackson, D. and Usher, M. Grading Student Programs using ASSYST. *Proc. 28th ACM SIGCSE Tech. Symposium on Computer Science Education* (1997), San Jose, California, USA, 335-339

[7] Jackson, D. Using Software Tools to Automate the Assessment of Student Programs. *Computers and Education (Pergamon), 17(2)* (1991), 133-143