The Ceilidh System for the Automatic Grading of Students on Programming Courses

S D Benford, E K Burke, E Foxley, C A Higgins

ltr @ cs.nott.ac.uk
Learning Technology Research
Computer Science Department
University of Nottingham
NOTTINGHAM NG7 2RD, UK

Keywords: Administration, Automatic Assessment, Courseware, Education, Software Metrics

Abstract

We give an overview of the Ceilidh courseware system. This is a general purpose system supporting the provision of student programming courses, and performing all the courseware assessment and administration related to a given course. It is currently distributed to over 100 universities in 15 countries, and supports 7 courses.

We will discuss first the overall structure of the system, and then summarise various different user views of the facilities it offers. This will give a good picture of the user view of the software. We will then describe more details the marking metrics and their implementation, and the effects (some of them unexpected) that the system has had on the student learning process.

1. Introduction

There are four main areas involved in teaching, which must therefore appear in a software to support teaching. We refer to such general purpose software as *courseware*. The four areas are

- The administration of courses
- The assessment of student achievement
- The support of tutorial learning activity
- The presentation of information to students

All of these represent areas in which computers can assist the learning process. Most applications of computers to education involve *Computer Assisted Learning* (CAL) systems; these concentrate on the last of these areas of activity. The first three, however, represent areas which generally provide the least job satisfaction for academic staff, and are areas in which the load on staff increases in proportion to class size.

1.1. The administration of courses

This heading includes many standard administrative tasks, including the following.

their breakdown) and by looking at samples of submitted work. Defaulting students can be automatically detected and active measures taken to follow them up.

- Monitoring overall course progress (class strengths and weaknesses). When the assessment is performed by computer, it is essential that the teacher be kept in touch with overall class strengths and weaknesses.
- Informing tutors of relevant information. The system can pro-actively email tutors and students when work is not submitted on time, or when marks fall below a certain threshold.
- Collection of on-line work of any sort. The system can collect essays, answers to questionnaires, and other types of student coursework. The use of Ceilidh simply as a work submission/collection facility has been welcomed by staff and students. Work in the form of essays/reports can be submitted on-line by the students (having been generated using a word processing system) and accepted and stored under Ceilidh. The work can then be marked by hand on or off the machine, and the marks

thus generated then entered by hand. The hassle of collecting 160 paper reports on a given date is considerable; this way we have no boxes of paper, and no possible complaints of "But I DID hand it in!".

- Searching for evidence of plagiarism. Copies of all submitted are stored in the system for future reference by tutors, teachers and examiners; we can therefore perform plagiarism tests on the work. The plagiarism pattern over a series of exercises can be significant; in general the known presence of plagiarism testing itself acts as a considerable deterrent to copying.
- Appropriate security control and reporting. Since the system both grades students, and stores details of their marks and copies of their coursework, absolute security is essential. Students should be able to inspect and comment on all marks and work related to them; tutors should be able to look at the marks and work of their tutees; and the teacher should be able to look at all marks and work. The system also allows for audit trails to be set to monitor either an individual student's activities, or the overall class use of compiling, debugging or printing activities.

1.2. The assessment of student achievement

We use the computer to perform automatic marking of student work submitted in various forms such as

- Computer programs in various languages, marked with static and dynamic metrics
- Multiple choice questionnaires
- Question/answer exercises
- Natural language exercises
- Essays or reports

The first of these is currently the most widely used Ceilidh facility.

With a typical class of 160 students, the marking of student work was in the past divided between members of staff and postgraduate students, giving each person perhaps 20 scripts to mark. This often resulted in slow response to the students from some markers, and in the absence of uniform standards. The computer can mark several exercises per student each week, with a response time of seconds, and with absolute impartiality.

Originally (when machines were not so powerful as now) the marking was done overnight; the student submitted the work during the day, the marking was done (and results emailed to the student) overnight. This implied at most one attempt per day.

Now we mark interactively. The teacher can set the level of instant feedback which the student receives; typically the student may be told an overall mark or grade, together with numeric or English language information on where marks were lost. The marks are stored and made available also to the course teacher and the student's tutor, and a copy of the student program is stored for possible future reference. Currently a complete mark history is stored (each mark complete with date and time stamp and component sub-marks), but only the latest program source is kept.

1.3. Tutorial help

Each week the student for each of several exercises (see below) will read a file specifying the problem to be solved, and will write a program to the given specification. The submission process can provide feedback in the form of numeric marks, alphabetic grades, or English language sentences telling the student where most marks have been lost.

By allowing repeated submission, the student can thus be lead to a better solution.

All marking is against software quality targets of which the student is made aware in lectures and documentation. The student thus gets into a habit of working to quality targets.

1.4. The presentation of information to students

The traditional rôle of computers in education has been based primarily on the presentation of information (text, graphics, multi-media animations) to a student, with the speed of progress determined by the student, and with different routes being followed depending on the student's choice and on the system's assessment of the student's progress. The assessments performed in these cases are not generally complex, and do not usually form the basis of the student's coursework assessment or count directly towards examination gradings.

We do not currently cover this aspect of courseware in any great sophistication. We provide on-line notes which can be viewed through

ASCII or PostScript viewers, with Acrobat viewers to be included soon. The Ceilidh project aims eventually to cover this area, and this work forms part of our on-going development; one approach is based the Modula 2 environment CLEM[1] already developed in the UK at Manchester Metropolitan University. and another involves the system being added to World Wide Web by Cardiff University[3] in Wales, UK.

2. User View

When a user enters the Ceilidh system, their identity is checked against a number of stored lists, and they are offered different facilities according to whether they are classified as

- student
- tutor
- teacher
- course developer
- system administrator

We will describe in some detail here only the student view; other views will be summarised very briefly. For further details see the Ceilidh guides (e.g.[2] available with the system or from the authors) for full details of the facilities offered to the other classes of user and the various interfaces and platforms on which the system is available. Originally the emphasis of Ceilidh was on computer program marking.

The student each week would

- Call up Ceilidh and select the chosen course
- Find which exercises have to be completed and their deadlines
- Select the chosen exercise
- Read the question(s) defining problems to solve
- Ask for a skeleton program outline if offered
- Develop a solution, compile and test it
- Submit it to the system for assessment

The last two steps can be repeated if the teacher permits. This can form a learning process if the teacher has set an appropriate level of marking feedback.

3. Other views

The Ceilidh system responds differently to different classes of user.

If a *tutor* calls the system, it will offer read access to all submitted work and marks. If a *course developer* calls the system, it will offer write access to all material in the course for which the developer is authorised. The developer can amend the notes, amend existing exercises, set new exercises, etc.

The *teacher* has facilities suited to the person running an existing Ceilidh course. They can look at the mark statistics for a particular class of students, for an exercise or for an individual student; they can find who hasn't submitted (and perhaps email them and/or their tutors); they can look at the overall class program metrics; and they can check for plagiarism in submitted work. The plagiarism pattern over a series of exercises can be significant; in general the known presence of plagiarism tests acts as a considerable deterrent to copying.

4. Marking metrics

4.1. The "oracle"

At many points in the assessment process, we will be looking for particular messages or information in program output (to check the results of a dynamic test), or in the output of an analysis program (to check for the presence of certain messages in output from *lint* for example), or in text files (to check for particular constructions in the source of a student program). We will therefore discuss the construction of such a program before we discuss the assessment process in detail.

A program which looks for correctness in output is called an *oracle* . Construction of an oracle is a non-trivial problem in any situation where the exact form of the text being searched is not exactly specified. We have chosen to implement an oracle based on *regular expressions* (REs), but with numerous extra features added by experience to assist the Ceilidh assessment process. The course developer essentially supplies a file of REs which are then used as a search pattern in the analysis. It is then up to the developer of the REs to allow for a variety of expressions representing the desired meaning.

In looking for particular messages from *lint,* for example, the format of the messages is fixed, and it is easy to search for them. In looking for meaning in the output of a program, the format may not have been accurately specified in the question, and we may have to search for all possible patterns.

4.2. The assessment of programming exercises

Our primary consideration in this section is the detail of the assessment process.

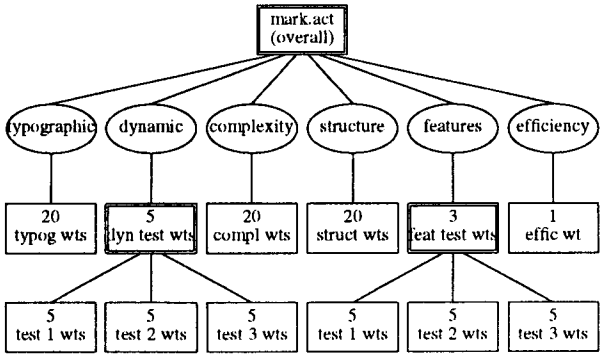

Figure 1 : The marking process

As indicated in figure 1, the overall assessed mark for traditional programming languages such as C, C++ and Pascal is currently made up from at most six major subcomponents, each of which provides a mark which is made up from a number of further subcomponents. At each level the course developer can choose the relative weighting of the particular subcomponents needed at any level to suit any required teaching emphasis. Extra components can easily be added into the general system.

At the top level, the main headings (some of which can be totally omitted if required) in the order in which we will explain them are

- typographic layout of the source
- structural complexity of the source
- structural weaknesses of the source
- exercise-related features of the source
- dynamic correctness of the executable
- dynamic efficiency of the executable

The developer must choose which of these are to be included, and give each of them a weighting factor. The developer can also specify the order in which the tests will be performed; this is significant, since students tend to attach most importance to the first figures they see. The control of the marking sub-programs is done using the file `mark.act` (marking actions). We will now discuss further the details of each of the sub-assessments.

4.3. Typographic source analysis

The Ceilidh system program `typog_c` reads C program source (`typog_C` reads C++ source, we need one program for each language we wish to analyse) and computes various statistics associated with maintainability and readability of the source, such as

Average characters/line	% blank lines
Average spaces per line	
Average function length	% good function
Average identifier length	% names with good length
% define's	
% number comments	% chars in comments
indentation errors	
% indentation	% indent errors
global variables	internal denotations

For further discussion of the typographic and complexity metrics, see Zin and Foxley[5] in which more details are given. The metrics we have chosen relate particularly to software maintainability and readability criteria. The students can obtain the system definition of features such as *good indentation* for C from the *help* facility in Ceilidh.

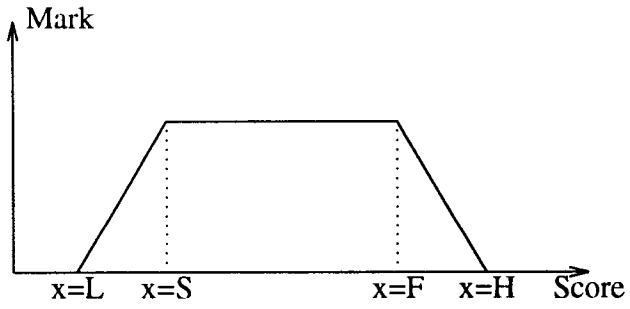

Figure 2 : Mark awarded versus parameter

Each of the general metric factors is measured, and compared with stored values. For each of the parameters we are marking, there will be a range of values within which full marks are awarded (from "S" to "F" in figure 2), and a wider range within which part marks will be awarded (from "L" to "S" and from "F" to "H" in figure 2). Beyond the outer range (below point "L" or above point "H", no marks are awarded.

About 20 components are measured in the typographic analysis.

4.4. Complexity analysis

This follows a similar pattern to the typographic analysis, but in this case the metrics for the student program are not compared with absolute values, but with those for the developer's model solution program. The factors involved include the following.

Number of reserved words
Number of gotos
Number of conditionals
Number of loops
Number of braces
Number of square brackets
Number of round brackets
Number of function calls
Number of numeric denotations
Number of includes
Number of operators
Depths of conditionals
Depths of loops
Max depth of braces
Max depth of square brackets
Max depth of round brackets
Number of function calls
Number of other denotations

If there are significantly different possible solution programs to a given problem (e.g. a recursive and an iterative solution), it should be possible for the developer to supply more than one model, and for the complexity marking to compare the student program with all of these models, and choose the best fit. This is not yet implemented.

About 20 measurements are involved in the complexity analysis.

4.5. Program structure

To look for structural weaknesses in the student's program, it can be re-compiled in the present C++ GNU compiler with

```
g++ -c -Wall
```

(all warnings) or for C programs we can use the *lint* command. Certain warning messages concerning aspects of the program structure which we wish to consider (such as variables declared and not used) are picked up by an oracle, and a score awarded. The oracle file of REs is applied to the output of the *lint* type program. We would welcome a volunteer to write a structural analysis program for C++ more directly related to our teaching needs than *lint* or its counterparts.

4.6. Features of the program

The Ceilidh system is intended to replace the assessment of programs by hand. When assessing program sources by hand/eye, one typically looks at the general layout (the typographic marks above), and for particular problem dependent features such as

- occurrences of particular numeric denotations within the code which should really be set as constants (in C++) or "`#define`"s (in C);
- the use of incorrect denotations such as "<= 59" where "< 60" is better practice;
- the definition of specific ASCII characters by integer denotations such as 64 or octal 0100 rather than by using the correct character denotation in C or C++ as 'A'.

The marking system therefore allows the setting up of exercise-specific files of REs in the exercise directory, which will then be marked using the oracle against the program source with comments and strings removed. Other files can be set up to be matched against the comments or strings in the program if required. Experience can be gained by looking critically by eye at student solutions to a given problem, and the features oracle can then be improved for future years.

For a typical exercise, there might be 3 features files each containing 5 REs, a total of 15 weighted assessments.

4.7. Dynamic correctness

These are the most difficult tests to set up. The student executable program is run against various sets of test data (or driven from a number of shell scripts) provided by the developer, and an appropriate `oracle` searches the program output for signs of correctness in each test. Alternative oracle programs (e.g. for searching for numeric values or for semantic meaning in a sentence) may be specified in place of the standard oracle based on REs.

The dynamic testing may typically consist of about 5 tests, each with up to 10 REs to be searched for in the program output.

4.8. Dynamic efficiency

The C system on SUNs (not the C++ system) can examine the number of times each line of the code is executed (using *tcov* on the SUNs, the name relates to the concept of *total cover-*

age of code during execution of tests), and compares the maximum line execution count with those for the developer's model solution.

4.9. Summary of the marking process

The overall marking thus consists of a tree structure hierarchy of weighted assessments as shown in figure 1 above, where the number of weighted components is roughly as follows.

topic	file containing weights	no of weights
overall	`mark.act`	6
- typographic	`model.tv`	20
- complexity	`model.cv`	20
- structure	`model.st`	20
- features	`model.fv`	3
- - each feature file	`model.f?`	5
- dynamic	`model.dv`	5
- - each dynamic test	`model.k?`	10
- efficiency	`model.d`	1

We may thus easily (depending on the number of features and dynamic tests implemented) have 120 measurements combined into a final assessment using 130 weighting factors.

5. Educational implications

We have described above the details of the methods used to arrive at a quantitative assessment of a submitted program. There are several other aspects of the assessment process which have been found to be important, and which will be summarised here. These arise from the interactive nature of the student submission process.

5.1. Public availability of information

If a student is being graded by computer,

> all the formulae being used must be publicly visible; and
>
> all the student's submitted work must be visible to that student. All this information must be equally visible to an internal auditor or external examiner.

5.2. Exercise weighting

In order to produce a final grading for a student over a complete programming course, the teacher can nominate certain exercises for completion by certain dates, and can assign a weight to each exercise. The total final mark is then the weighted average of the student's mark for each nominated exercise.

In addition, the final marks can be scaled in order to produce a mark distribution as required by the academic institution. Ceilidh tends to award high marks, since in general the program is awarded a mark for everything which it does which is correct. The institution in which the course is run may require that a mark of 40% represents a bare pass and 60% an average mark, whereas in Ceilidh the average will be perhaps 90%, with 70% representing the minimum which should be achieved.

5.3. Interval between submissions

Because of the interactive nature of Ceilidh, students find the mark awarded to their program immediately. In order to discourage the students from making a small change to a program without thinking, merely to re-submit and see whether the mark has gone up or down, the teacher can set a minimum time interval between consecutive submissions.

5.4. Total number of submissions

For the same reason as the previous paragraph, we may wish to limit the number of resubmissions. The teacher can control the maximum number of submissions; if the work is primarily to help the student learn programming, unlimited submissions and extensive feedback may be permitted. If the main intention of the exercise is the assessment of the student's ability, the teacher can restrict the student to a maximum of only one submission. The two possibilities can be summarised as follows.

- From the student learning view, we would like extensive feedback on weaknesses and errors, and the possibility of repeated resubmissions.
- From the teacher's point of view, student assignments which count towards the course assessment should perhaps be completed with less help given to the student, and perhaps without the possibility of resubmission.

5.5. Viewing the test data and using a model executable

It may be helpful for the students to be able to view the test data which will be used to test their program. However, there may be occasions when this is not wanted, and the student must work from a written specification.

Similarly, the teacher can if wanted provide a model executable, which the students can use to see the required effect in particular cases of, perhaps, erroneous data.

Both of these features can be controlled by the teacher.

5.6. Saving the marks

For security reasons, a record in the form of an audit trail of every mark submitted is retained, together with the course, unit, exercise, date, time, the name of the student concerned, the name of the person submitting, the overall mark, and details of the complete mark breakdown. In our system, it is the most recent mark for a particular student which matters, not the best mark.

5.7. Saving student source and executables

For many reasons, we have always retained a copy of the student program at each submission. However, for reasons of disc space limitation, we generally keep only the most recently submitted copy. There are now features to enable all the versions of the program to be kept, so that examiners or researchers can follow the history of the development of the program.

The system has now been extended so that we can save any number of files, useful in exercises where the student may have to create a multi-module program, perhaps together with a text report on the exercise.

5.8. Communication with students

Every Ceilidh menu has an option

```
Send message to teacher?
```

This results in more contact with the teacher rather than less contact. On a recent course for 130 students, the teacher received 300 communications (and answered every one!), coming from more than 80 different students. The communications varied from disbelief that their program was not perfect, to genuine hard-luck stories as reasons for non-submission.

6. Summary

The system is now in use at many sites. Some have taken and run it exactly as supplied, other have tailored to their own needs either by writing new exercises, rewriting existing courses, or adding completely new courses. The feedback from teachers is that much mundane effort in administration and marking is saved. The students report (after initial concern that a mere computer program cannot possibly grade their own programs) that they are happy with the system. Independent evaluations show that the programming abilities and understanding of the students after a Ceilidh course is at least as good as under traditional teaching methods.

We are therefore happy that the system is satisfying its original needs.

References

1. Tom Boyle et al, *The CLEM System for teaching Modula 2.* Manchester Metropolitan University
2. Steve Benford, Edmund Burke, and Eric Foxley, *Student's Guide to the Ceilidh System (2.4)*, 1995. LTR Report, Computer Science Dept, Nottingham University
3. A D Marshall, S Hurley, S N McIntosh-Smith, R R MArtin, and N M Stephens, *Courseware on the Cardiff Information Server*, September 1994. 2nd All-Ireland Conference on the Teaching of Computing, Dublin
4. E. W. Weyuker, "On Testing non-testable program," *The Computer Journal*, vol. 25, no. 4, pp. 465-470, Nov 1982.
5. Abdullah Mohd Zin and Eric Foxley, "Automatic Program Quality Assessment System," *Proceedings of the IFIP Conference on Software Quality*, March 1991. S P University, Vidyanagar, INDIA
6. Abdullah Mohd Zin and Eric Foxley, *The Oracle program*, 1992. LTR Report, Computer Science Dept, Nottingham University