Grading Student Programs using ASSYST

David Jackson & Michelle Usher
Department of Computer Science
University of Liverpool
P.O. Box 147, Liverpool L69 3BX, U.K.
e-mail: d.jackson@csc.liv.ac.uk

Abstract

The task of grading solutions to student programming exercises is laborious and error-prone. We have developed a software tool called ASSYST that is designed to relieve a tutor of much of the burden of assessing such programs. ASSYST offers a graphical interface that can be used to direct all aspects of the grading process, and it considers a wide range of criteria in its automatic assessment. Experience with the system has been encouraging.

1 Introduction

The task of grading student programs is neither simple nor entirely mechanical; rather, it is often a tedious and laborious process that is prone to human error. Recently, a number of researchers [1,2], including ourselves [3,4], have begun to investigate the possibility of using computers to *automate* the job of grading student software. Of course, it is acknowledged that there are a number of aspects of assessment that are still not amenable to computer implementation: understanding the comments contained in program listings, for example, is still beyond the ability of the most advanced artificial intelligence techniques. There are, however, many areas in which an automatic grading system can do at least as well, and often better, than a human tutor. Furthermore, such a system can be consistently accurate and efficient in its analysis of many programs; the same cannot be said of most human graders.

The aims of our research have been to identify those parts of assessment that are open to automation, and then to build software tools to implement them. Gradually, these have evolved into a single extensible system with a graphical user interface; we call this tool ASSYST (ASsessment SYSTem). There are two user views into this system: the first and simplest view provides the means by which a student submits a program electronically for subsequent grading; the second view is that of the tutor, who is able to oversee and direct the assessment process. The fallibility of computers does not go unrecognized, and it is still very much the human who is in charge and who can refuse to accept the diagnosis of the system when it seems inappropriate; the name ASSYST was carefully chosen to reflect this working relationship.

In this paper, we concentrate on the tutor's view of the system. In the limited space available we will not be able to do full justice to the capabilities of ASSYST, but we hope to give a flavour of what can be achieved with it, and the way in which it performs its assessment of programs.

2 Using ASSYST

Figure 1 shows a screen dump from the terminal of a tutor using the ASSYST software to assess a group of students. It will be seen that its interface comprises a number of windows for different aspects of the assessment.

When the system is started up, the tutor is presented with the initial window shown at the top of the screen dump. For a given course (in this case an introductory programming course with code 2CS21), this window displays information on each of the programming exercises attempted by the students. If a tutor now mouse-clicks on one of these exercises, the

SIGCSE '97 CA,USA

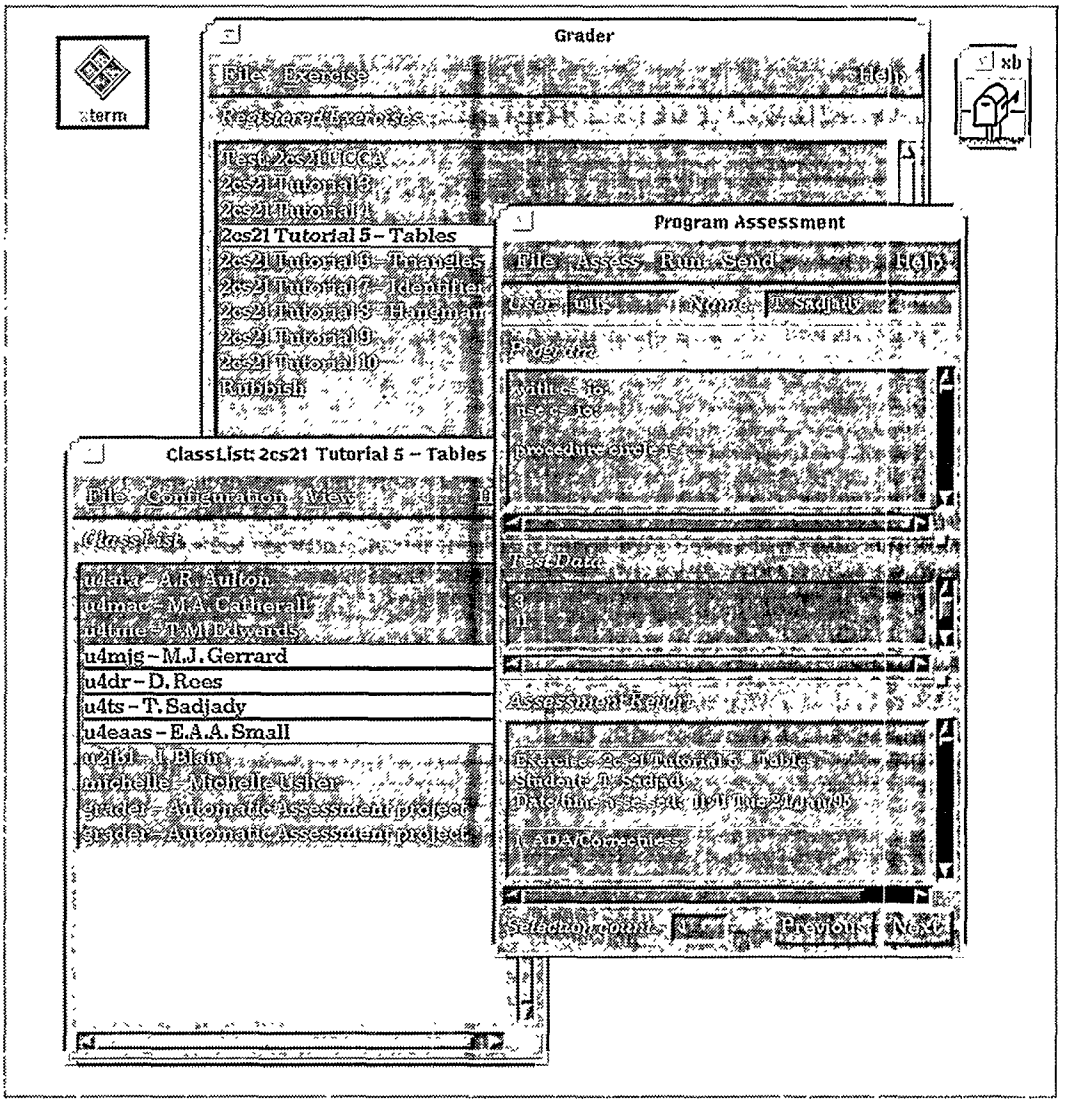

Figure 1: ASSYST in action

names of the students contained in that tutor's group are presented in a separate window; in Figure 1 the tutor has selected 'Tutorial 5 - Tables', causing the window in the bottom left corner to be displayed.

Again using the mouse, the tutor may now select any student or collection of students from this group, and from within the 'view' menu bring up a window from which the assessment itself may be initiated. This window, shown on the right of Figure 1, is divided into three vertical sections: the top section contains a listing of the student's program code; the middle area shows any test data the student has used to validate the program; and the bottom section will contain the assessment report when it is generated by the system.

Running the automatic assessment is simple; it is performed from the 'Assess' menu at the top of the window, and it produces a report such as that seen in Figure 2. It is possible for the tutor to send a copy of this report to the student via e-mail; again, a menu is available for this.

3 What ASSYST assesses

An important component of ASSYST's function is related to the 'housekeeping' operations associated with running practical classes for a programming course; these include handling the submission of ex-

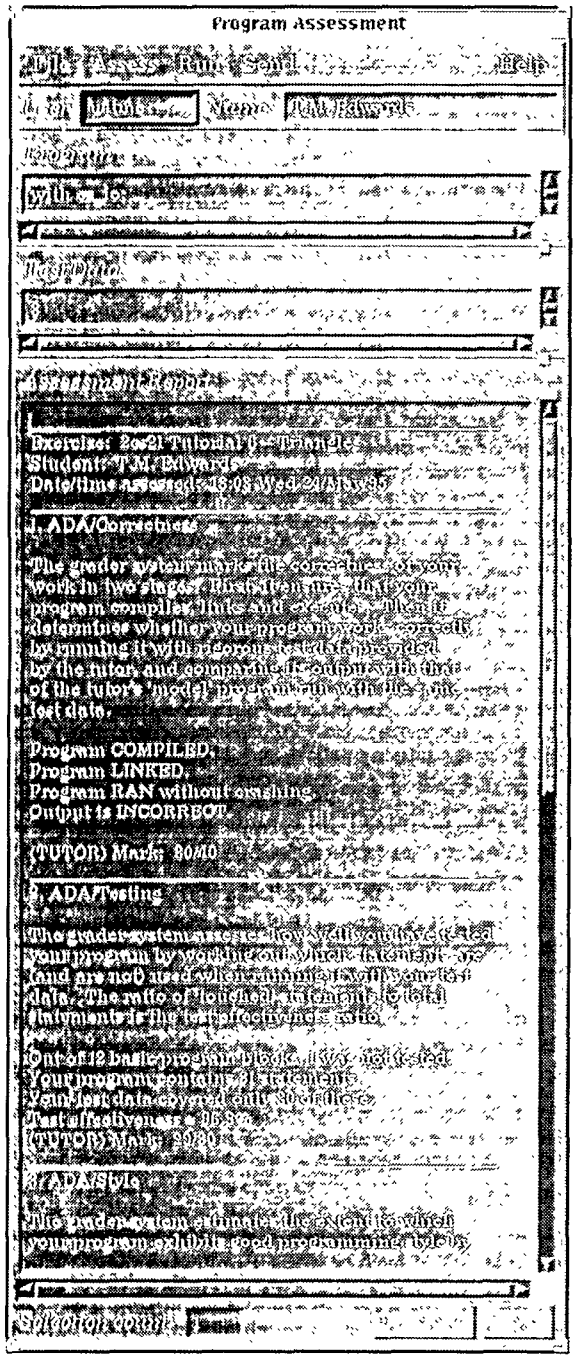

Figure 2: An assessment report

ercises by students, managing the directories and filestore for organizing these exercises, sending reports back to students, allowing weightings to be associated with different parts of the assessment, and so on [11, 12]. The main *engine* of ASSYST is of course devoted to performing the grading task itself. In its current form, this engine applies metrics in five important areas of assessment, as shown in Figure 3.

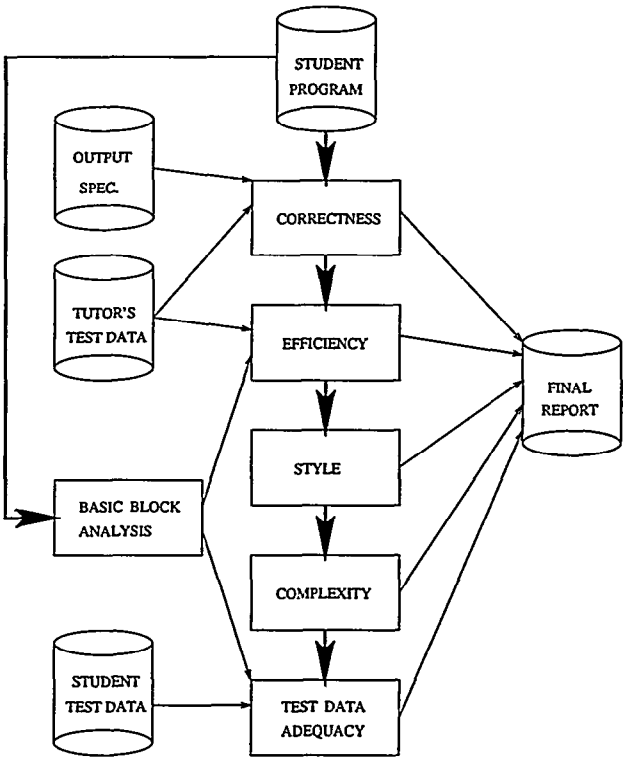

Figure 3: The assessment process

For most assessment metrics, it would be iniquitous to apply a simplistic 'right or wrong' approach that gave either full marks or nothing at all to a student. Suppose, for example, that for a given exercise it is judged that the 'ideal' amount of program commenting equates to ten percent of the complete text. While it might be thought reasonable that a student whose code contained either no comments or nothing but comments should be given zero marks, it would be somewhat harsh to apply the same penalty to a student whose comments accounted for eight or twelve percent of the text. For this reason, the system uses a scaling approach in awarding marks. The method adopted is based on one suggested by Berry and Meekings [9] in their work on program style analysis; whenever possible, we have also employed it in the other metrics. The idea is that, for each criterion,

the tutor specifies four points which define a simple graph that is consulted in determining the mark obtained. Such a graph is shown in Figure 4. Hence, a score lying between points B and C will gain full marks for the student; a score lying between A and B, or between C and D, will result in some fraction of the full marks, according to where it lies on the graph; a score less than A or greater than D obtains nothing for the student.

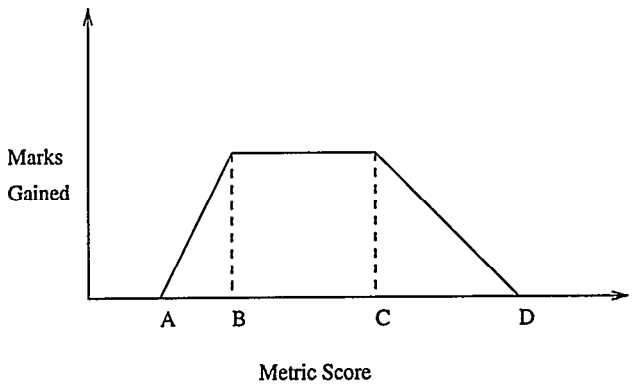

Figure 4: Marking graph

3.1 Correctness

On submission of a program solution by a student, the first thing that the system will do is to confirm that the program compiles and links. If this is the case, the resulting executable version will be applied to both the test data provided by the student, and the data supplied to the system in advance by the tutor. The next step of determining whether the program output is correct is a research topic in itself. The approach used in ASSYST is a novel one which involves treating the analysis of output as a pattern matching problem [5]. The tutor produces a specification of the output that a correct program will be expected to generate, and ASSYST invokes the Unix Lex and Yacc tools to create a program that checks that the output from the student solution conforms to this specification.

3.2 Efficiency

This is measured in two ways. The first is simply an indication of the CPU time spent in executing the program, and is obtained by making calls to a system timer routine. If the tutor has supplied a model solution to the system, then marks are assigned on the basis of a comparison of the timings of this and the student's program. The tutor may also specify an amount by which the student's execution time may differ from the tutor's and still get full marks. Beyond this point, marks are reduced as execution time increases.

The problem with this approach is that for most of the programs written for an introductory programming course the execution time is vanishingly small, and the unit of time recorded by system timer routines may not be sufficiently fine-grained to uncover any variation in the timings of two different programs. Thus, the second method of measuring efficiency is often more revealing, and takes the form of a statement execution count. Prior to compilation, ASSYST performs a static analysis of the student's program to determine its structure in terms of basic blocks (this is also used in the test data adequacy metric – see below). The number of times each block is entered is gathered at run-time. A count of the number of statements in each block is recorded by the system, so that after the student's code has been run it can calculate the total number of statements executed. Once again, this may be compared to the tutor's model solution, and the boundary values for gaining full marks and zero marks can be specified.

3.3 Complexity

This is determined using McCabe's metric [6]. While it is acknowledged that this has been criticised for its lack of theoretical foundation [7], and that many other metrics have been proposed [8], it does have the advantages of being well-known and easy to automate. McCabe defines the cyclomatic complexity of a program as

$V(G) = e - n + 2$

where e is the number of edges and n the number of nodes in the control flow graph G of the program code. To be more precise, the version of the metric used in ASSYST is that which breaks compound predicates into separate branches.

3.4 Style

For this, the style metrics suggested for C programs by Berry and Meekings [9] have been adapted for use on Ada code. The program characteristics which these metrics assess include module length, number of comment lines, use of indentation, and so on. In Berry and Meekings' system, each of these characteristics is assigned a weighting to indicate its contribution to the final mark, and a set of boundary points indicating the values that will gain full marks, zero marks and marks in between these limits. These are all hard-coded into

the system. ASSYST offers more flexibility by providing a window via which these values may be altered by the tutor.

3.5 Test Data Adequacy

Students who submit programs are normally expected to do their own testing, and to submit their test data as part of the whole solution. It can be quite time-consuming for a human tutor to assess the adequacy of this test data, unless some automatic assistance is available. The final part of the evaluation, then, is a measurement of the extent to which the student's data leads to statement coverage on execution. For this, we apply Test Effectiveness Ratio 1 (TER1) as defined in [10]. TER1 is given as the number of statements exercised at least once, divided by the total number of executable statements. Since we already perform a basic block analysis of the student's code in order to measure efficiency, the calculation of TER1 is straightforward: we simply add together the number of statements in all of the basic blocks which attain an execution count greater than zero, and then divide this by the total statement count for the whole program.

4 Conclusions

It will probably be evident from the screen shots given earlier in the paper that ASSYST is in use on one of our programming courses. This is an introductory course in which Ada is used as the teaching language. Our trust in automation is not such that we yet allow ASSYST to do all of our grading (tempting though that sometimes is), but experience with the system to date has been extremely encouraging. ASSYST can often produce surprises by spotting things that a human has completely missed: programs judged to be correct from visual examination of the listings have been found to fail miserably when automatically executed by ASSYST.

Recently, some of the automatic marking techniques used in the system have also been applied to a second-year course on systems programming in which C is the main language. Positive feedback included appreciation of the much faster turn-around of exercises than was previously possible, and the thoroughness of the analysis. A small number of students required reassurance that the human element was still very much present in the marking process (which it was). One or two expressed dissatisfaction with the weightings applied in the marking scheme (they were not used to having their programs executed by others!), but this was not a criticism of the approach itself and could easily be modified.

Initial reactions to ASSYST have been highly positive. It is simple to use, and does much that a human tutor would find difficult or impossible to do without its help. Since it is built to be extensible, it is hoped that it may be applied to a number of computer science courses using a variety of programming languages, and that its powers of assessment can be honed as more experience with its use is gained.

References

[1] Benford S., Burke E. and Foxley E. (1992), Courseware to support the teaching of programming, Proc. Conf. Developments in the teaching of computer science, Univ. of Kent at Canterbury, pp 158-166

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

[3] Jackson D. (1996), A Software System for Grading Student Computer Programs, Computers and Education (Pergamon), to appear

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

[5] Jackson D. (1991), Using Software Tools to Automate the Assessment of Student Programs. Computers and Education (Pergamon), vol. 17 no. 2, pp. 133-143

[6] McCabe T. A. (1976), A complexity measure. IEEE Trans. Softw. Eng. vol. SE-2 no. 4, pp 308-320

[7] Sheppard M. (1988), A critique of cyclomatic complexity as a software metric. IEE Software Eng. Jou., pp 30-36

[8] Waguespack L. J. and Badlani S. (1987), Software complexity assessment: an introduction and annotated bibliography, ACM Sigsoft Software Eng. Notes vol. 12 no. 4, pp 52-71

[9] Berry R. E. and Meekings B. A. E. (1985) A style analysis of C programs, Comm. ACM vol. 28 no. 1, pp 80-88

[10] Woodward M. R., Hedley D. and Hennell M. A. (1980), Experience with path analysis and testing of programs. IEEE Trans. Softw. Eng. vol. SE-6 no. 3, pp 278-286

[11] Isaacson P. C. and Scott T. A. (1989) Automating the execution of student programs. SIGCSE Bulletin vol. 21 no. 2, pp 15-22

[12] Reek K. A. (1989) The TRY system – or – how to avoid testing student programs, SIGCSE Bulletin vol. 21 no. 1, pp 112-116