SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper., **29**(8), 721–740 (1999)

A Secure On-line Submission System

MICHAEL LUCK* AND MIKE JOY
Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK
(email: {Michael.Luck,M.S.Joy}@dcs.warwick.ac.uk)

SUMMARY

As student numbers on computer science courses continue to increase, the corresponding demands placed on teaching staff in terms of assessment grow ever stronger. In particular, the submission and assessment of practical work on large programming courses can present very significant problems. In response to this, we have developed a networked suite of software utilities that allow on-line submission, testing and marking of coursework. It has been developed and used over the course of five years, and has evolved into a mature tool that has greatly reduced the administrative time spent managing the processes of submission and assessment. In this paper, we describe the software and its implementation, and discuss the issues involved in its construction.

KEY WORDS: submission; assessment; coursework; testing; security; course management

INTRODUCTION

Large numbers of students on computing courses in further and higher education present a distinct problem for those involved in the delivery of the courses, especially in relation to the assessment of practical work. The teaching of programming in particular demands the preparation and assessment of practical exercises which, in the face of such large numbers, can be prohibitive in terms of time and effort. For an effective course, however, the assignments must be processed and graded quickly so that students receive useful feedback that can benefit their progress.

The solution to this problem, as has been recognised by several institutions, lies in the potential for the processes of submission and assessment of programming assignments to be automated, at least in part. In essence, there are two important aspects here that can be labelled as *information management*, relating to the submission and organisation of assignments, and *assessment techniques*, relating to the testing of the submitted assignments. While the obvious issues involved in the introduction of such systems are concerned with pedagogical matters of how to employ the assessment techniques, user-interface design, and so on, issues such as *security*, especially with a user base of technically competent students, are equally important.

The problem

As has been documented elsewhere [1–3], it is generally not possible to assess the correctness of a program with a large degree of accuracy simply by inspecting the source code

*Correspondence to: Michael Luck, Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK.

CCC 0038–0644/99/080721–20$17.50

Received 26 June 1998
Revised 3 March 1999
Accepted 18 March 1999

listings. While this is obviously true of large programs, it also holds even for small programs on introductory programming courses. The only way to arrive at an accurate assessment of a program is by running the program against several sets of test data, yet this is time-consuming, and can be prohibitive if done manually with large classes. It is possible to require students to provide evidence of their own testing, but this requires further skills on the part of the students that are not typically covered by introductory programming courses. Moreover, such tests might easily be *faked* by students modifying the output from their programs to give the desired results.

By automating the processes of submission and testing, these problems, at least to some extent, can be addressed. Indeed, several distinct requirements of systems for such a purpose were identified by a recent panel discussion [1], and elsewhere [4,5].

(a) The system must copy the student's source code program to a location accessible only by the instructor, noting the date and time of submission.
(b) It should allow for multiple source files of various types, including documentation.
(c) It should allow late submission of assignments.
(d) Before submission, the system should compile and run the program against public test cases to alert the student to obvious errors.
(e) After submission, the program should be compiled and run against several sets of test data.

Related work

There have been several attempts to address some of the concerns outlined above, with varying degrees of success. MacPherson [6] describes a system that allows students to work in a special course directory in their own filestore, but at the appropriate time transfers ownership to an instructor who can then subsequently run and test the programs. Canup and Shackelford [7] have developed a suite of programs for assisting in automatic submission, but do not address automated testing of programs. Isaacson and Scott [8] use a C shell script for automating the compilation and testing of student programs against sets of test data once students have placed their program files in an appropriate directory structure. Similarly, Reek's TRY program [2] copies student programs into the instructor's filestore, runs them against sets of test input data, and produces a log file that can be used to provide feedback to students. All of these systems address some of the main functionality requirements described above, but are rather rudimentary and, more importantly, do not provide adequate *security*. In particular, opportunistic attempts by students to exploit loopholes in the systems cause special concern given that learning environments are generally intended to encourage and stimulate experimentation.

Alternative approaches, such as that by Dawson-Howe [3], manage the process of submission and testing through the sending of email messages containing programs, data and results. While this avoids some of the security loopholes, it does not address many of the key requirements of such a system as described above. However, Dawson-Howe's work does go further than the others in that it includes some simple database management facilities for maintaining information about submissions and grades, and generating simple reports.

A third class of system, exemplified by one package in particular, deserves some discussion because of its size and distinct approach. *Ceilidh*, a system developed by Benford *et al.* at Nottingham [9], is a large system that contains many features including on-line exercises and teaching aids. Each course has several distinct components, all of which are accessible

from Ceilidh. First, general course information, such as lecture notes, assignment deadlines, specifications and solution outlines, can be viewed or printed. Second, programs can be edited, compiled and tested, with details of the compilation process hidden to a greater or lesser extent from the student. Third, the programming assignments can be marked by the system, and the marks provided as feedback, with completed work being submitted and retained together with mark details for further analysis. Finally, model solutions and test data can be made available to be viewed, run and tested.

While such a system is indeed a formidable achievement – and is gaining wider use [10] – in encapsulating in one monolithic structure nearly all of the possible forms of interaction that a student can have on a programming course, it sanitises the learning environment by restricting the nature of interaction. For example, familiarity with underlying tools, programs and shells is typically gained through experience of using them. This *learning by doing*, by which compilers and editors are encountered and understood, is denied in Ceilidh, which instead offers its own set of commands for editing and compilation that do not apply beyond this particular system. In this way, the variety of useful experience that students might otherwise have, and which is valuable in gaining familiarity with general software, is no longer available. In our approach, by contrast, we do not constrain the use of existing tools, but merely provide extra utilities that are used in conjunction with them to offer increased functionality. Indeed, there is a great temptation to use technology to develop such systems, yet we must be aware of the potential difficulties that they can cause beyond the immediate benefits, and of the barriers to confidence and familiarity that they can create in attempting to provide a rich, flexible and stimulating learning environment.

An integrated course management tool

This paper describes the design of an integrated software system for the task of submission and assessment, focusing on the technical issues and the lessons learned. Known as *BOSS*, the system comprises a suite of programs that allows students to submit coursework on-line – typically, but not exclusively, programming assignments – and allows them to be run against test data, the results of such tests then being made available to staff marking the submitted work. Some aspects of the software are similar to some of the systems described above (e.g., Reek [2]), but these are part of an overall integrated system that addresses the weaknesses identified. The approach taken has been to isolate tasks that can be fully automated, such as *submission* and *testing* of programs, to provide modules to implement each such task, and to provide a graphical user interface to these modules. In general, it is not feasible to automate *all* of the process of assessment – *marking* a program is a non-trivial process, best performed by a human. In contrast to alternative approaches elsewhere [9,10], we have attempted to solve the problems of security and reliability with the intention of seamlessly integrating the modules into a robust coursework management tool.

The next section provides a functional overview of the *BOSS* submission system, and identifies the main components. Then, the way in which the wide variety of information and raw data needed by *BOSS* is structured, and how it is managed, is described in detail. The important security aspects of the system are addressed, covering issues of data integrity, privacy and rogue programs. The main algorithms relating to the submission and testing of student programs are then presented, and the administration of marking is explained. The following two sections provide details of the plagiarism detection component and the user-interface, before concluding with an overall summary of the system and its benefits.

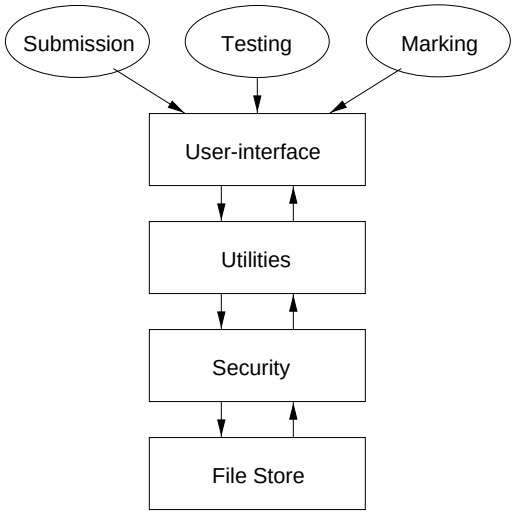

Figure 1. System overview

THE BOSS SUBMISSION SYSTEM

The *BOSS* system is intended to perform several distinct tasks within a single overarching framework. It is aimed at course management rather than instruction, in that it incorporates facilities for submission of assignments, their subsequent testing and marking, and the provision of feedback on assignments to students. Thus, it does not constrain in any way the delivery of instructional materials, which is a completely separate problem. To be effective in addressing these aims, the system must also ensure that security issues are sensibly dealt with, and that the overall system is well designed and organised.

The structure of the system reflects the conceptual division of the software into three core modules, each well-defined, which can be treated as largely independent components. These are represented as the ovals at the top of Figure 1, and address submission of assignments, their testing and their marking.

(a) The *submission module* allows students to submit a piece of coursework, and handles the task of copying that coursework to a secure location where it can be accessed subsequently.
(b) The *testing module* runs and tests a single piece of coursework against one data set, and reports success or failure according to the given expected output.
(c) The *marking module* assists an instructor in marking a collection of coursework after the submitted programs have been run and tested against several sets of data.

Each of these components provides information through the user-interface that is processed and managed by lower level utilities that access the central file store through careful and secure techniques. The arrows in the figure indicate the flow of information through the system and illustrate its general organisation. Before considering the distinct aspects of these components, we describe the aspects of the system that cut across them.

INFORMATION MANAGEMENT

A large part of the system is concerned with information management, as explained briefly in the introduction. Each student submits files containing programs and possibly other documentation, and these must be stored so that they may subsequently be accessed, compiled and tested. It should also be possible to store multiple submissions by a single student, backing up older copies to a suitable desired depth. All of this, of course, must be done in a secure fashion so that the stored data is not generally accessible, but only accessible to the instructor for the purposes of assessment.

The organisation of the stored data is structured around user-centred concepts relating to high level course organisation. While notions of courses, assignments, etc., are loose and may differ from institution to institution and from one degree programme to another, in the context of system design, distinct and specific meanings have been adopted. A *course* denotes a unit of a degree programme that contributes to that programme, such as an introductory programming course on a computer science degree programme. An *assignment* denotes a piece of work that forms part, or all, of the credit for a course, typically with a given deadline, and containing one or more *exercises*, each of which is a specified task such as writing a program. These come together with the convention that, for a given assignment, a student completes exactly one exercise, so that an assignment requiring students to complete two or more exercises would be specified as two or more separate assignments.

Using these notions of courses, assignments and exercises, the detailed file system can then be described. The entire file system structure for *BOSS* is illustrated in Figure 2, with the store for submitted programs forming a UNIX directory hierarchy by course, assignment and exercise, as shown in the bottom half of the figure. Each such directory contains a configuration file named `settings`, which specifies the properties for the course, assignment or exercise that directory represents. To facilitate discussion of the data, we have chosen to give specific illustrative names to the files and directories, but naturally these might be different in a real installation of the software.

Two usercodes are needed, `boss` to own the software and the submitted programs, and `slave` to run programs when they are being tested, as discussed below. Both of these are indicated in Figure 2, which shows the topmost directory as the `boss` home directory.

At the top level of the file hierarchy there are standard UNIX directories such as `man`, `src`, and `doc`, for storing the utilities and their related documentation, together with the following application specific files and directories:

(a) A file containing an *audit trail*, with entries for each submission, or attempted submission, is maintained to track submissions for use in the case of conflicts, as described below. This file is named `audit.log`.
(b) A *global* configuration file, `settings`, is used to temporarily *disable* the software should that be required as, for example, with updates or other software maintenance.
(c) For ease of addition of new courses to the system file space, a template of the course data files is stored in the `skel` directory.
(d) In order to prevent simultaneous submission of the same exercise, a directory labelled `locks` contains *lock files*. Whenever a user runs the program, a file in the directory is created; when the program terminates, the file is deleted. If the user attempts to run a second copy of the program, and a lock file is detected, the new program will terminate.
(e) The `slaves` directory is used as temporary workspace when running tests on programs during the assessment phase, and is owned by user `slave`, as discussed further below.
(f) Finally, the `bin` directory stores the program modules. This also contains the

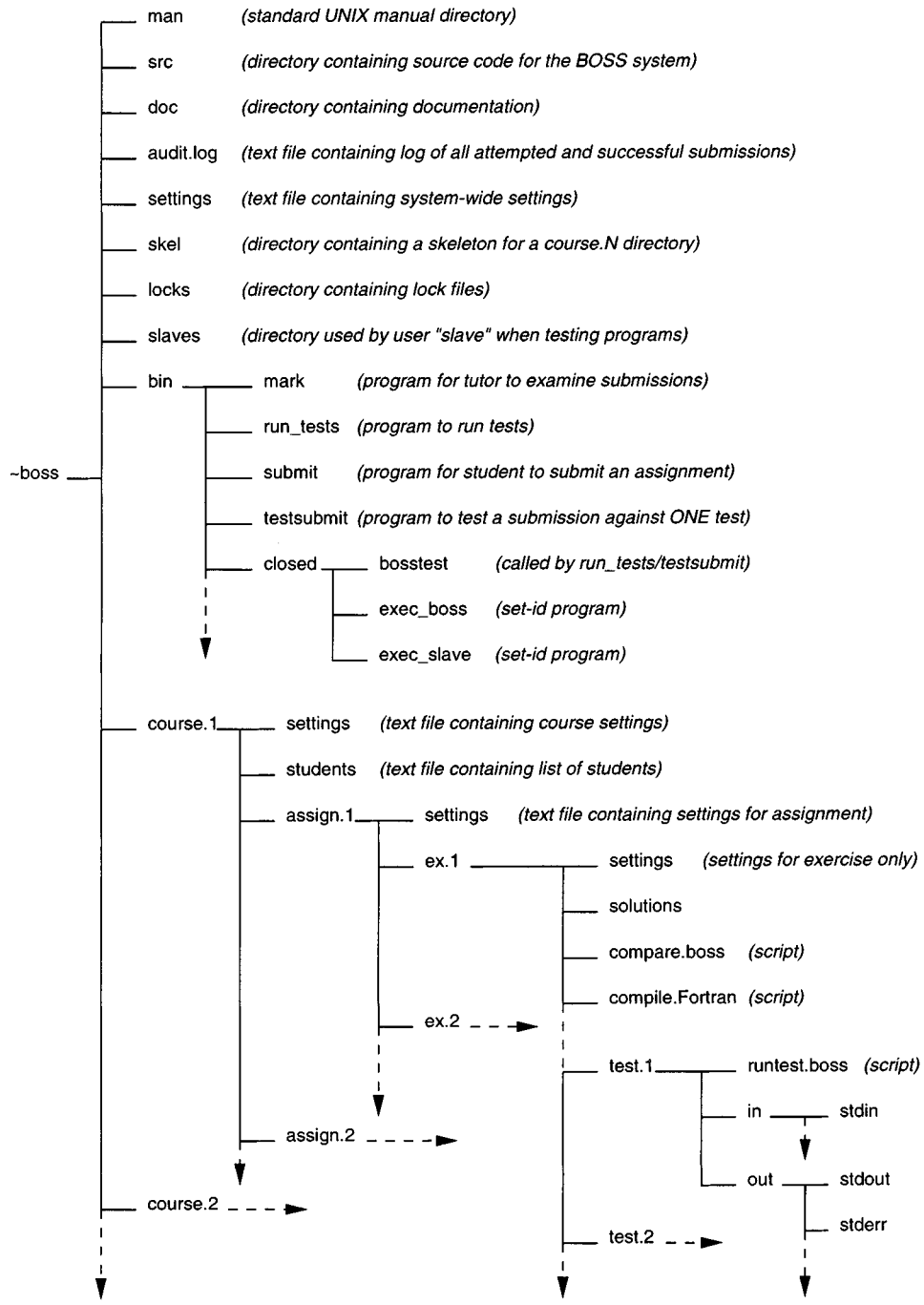

Figure 2. The BOSS file space structure

Table I. Information contained in the `settings` configuration files

Course	Assignment	Exercise
course code	submissions allowed?	CPU time limit
exam code	multiple files allowed?	file size limit
instructor	maximum mark	language selection
multiple submissions allowed?	penalty regime	real time limit
student restrictions	penalty per day	exercise title
course title	assignment title	
course tutor list		

subdirectory, `closed`, with no general access permissions, and which itself contains programs that can only be called from programs in the `bin` directory that are *set-uid* user `boss`. That is, when run, they inherit the file access permissions of `boss` rather than those of the calling program.

As described above, there are other top-level directories for each course, named `course.1`, `course.2`, and so on. In each such directory is another configuration file, again named `settings`, containing course properties including the course title, the course code number, the usercode of the instructor, and the usercodes of other staff assisting. The course may be configured so that only students from a given cohort may submit work, in which case a related text file called `students` contains the identification numbers of those students.

Within these course directories are subdirectories for each assignment, which are named `assign.1`, `assign.2`, etc. In addition to a title, the deadline and the maximum mark for the assignment are included in the properties detailed in the `settings` file.

Similarly, each assignment directory contains subdirectories for each exercise, `ex.1`, `ex.2`, etc., with each of these containing several further subdirectories:

(a) The `solutions` subdirectory simply contains a directory for each student who has submitted work for the exercise, comprising copies of the submitted files and the files used by *BOSS* to store the test results.
(b) There are several scripts in separate files, initially set to defaults, for compiling a program in various languages, such as `compile.Fortran`, and for comparing two data sets, such as `compare.boss`. The latter may simply contain the UNIX command `diff`, or may be made more complex as required.
(c) Finally, there is a directory for each test that is to be carried out on submissions for the exercise. These contain a script, `runtest.boss`, to run that test, and directories `in`, containing data for the test, and `out` containing the *expected* output for the data. The input includes the standard input stream, the output includes the standard output and standard error streams, and both may contain other files as specified.

The contents of the *settings* file at each level is given by Table I. In part, this summarises and illustrates the organisation of the whole system in that the levels at which the different aspects of assignments are considered determine the nature of the system itself. For example, only at the exercise level is the system concerned with the choice of programming language or time and space limits for program execution, as these are only relevant at this point. Moreover,

this allows complete flexibility in terms of all these choices until the last point at which a commitment to them must be made.

SECURITY

With a system such as this in which student work is stored on a central file system, security is paramount. The importance of this is heightened by the fact that many of the students involved in submitting work through the system are technically very competent. The broad heading of security covers several different aspects, each providing distinct possibilities for exploitation by dishonest and cunning students. By addressing the security issues described below, and by developing software that correctly implements the corresponding checks, no security failures have been suffered during the deployment of the *BOSS* software.

Data integrity

From the moment the student instructs *BOSS* to complete the process of submitting a piece of coursework, they must be confident that their files have been copied intact:

(a) All submissions must be *logged*.
(b) A mechanism for *verifying* that a stored document used for assessment is the same as the document actually submitted by a given student is needed to prevent claims of discrepancies between the two.
(c) A *feedback* mechanism must be in place to confirm to students that submission of their work has been successful.
(d) The *identity* of a student submitting coursework must be established so as to uniquely identify them.

Logging of all submissions is trivial, and details are recorded in the `audit.log` file mentioned earlier. Verification and feedback are tackled by forming an *authentication code* for each file submitted using the *Snefru* algorithm [11]. The Snefru algorithm is a *secure hash function* that maps an input file to a fixed-length byte array. Small changes in the input file generate completely different output that cannot be easily predicted, thus preventing a user who has submitted work subsequently altering their submission and claiming it was the same one they had previously submitted. Other algorithms are available, such as MD5 [12], but at the time of coding, an implementation of Snefru was the most readily available.

A file containing a *receipt* for the submitted files that includes their authentication codes is created, and is sent by email to a student *after* their files are submitted successfully. If the student does not receive such a receipt after submission, they are told to contact the instructor to resolve the problem. The authentication code for the receipt is written to the log file, and a copy of the receipt is stored together with the student's submitted files. Finally, the problem of student identification is solved by using an existing and available mapping between usercodes and university identification numbers, with all submissions being stored in directories labelled with these numbers.

Privacy

Submitted coursework must be stored with sufficient read-protect *security* mechanisms to prevent unauthorised access. This is achieved through allowing access only by the users identified in the appropriate `settings` files as described in the previous section.

Furthermore, writing and reading to and from these directories is only allowed through the submission and marking modules, and not in any other way.

Rogue programs

Once a program has been stored securely, there can still be problems that arise at the point of testing the program, and the *test harness* must guard against a number of scenarios.

(a) It is easy for programs to contain infinite loops and space leaks that cause the software to crash. The system must be *resilient*, in that it avoids such situations.
(b) Some programs may, by design or accident, contain code that adversely affects any of a number of other systems or processes. The test harness must therefore be *secure*, so that such *Trojan horses* are prevented from causing damage.

The resilience criterion is handled by standard UNIX signalling. That is, whenever an undesired event occurs, such as the program running beyond a certain time or using more than a pre-set amount of memory, the operating system sends a *signal* to the program that the program acts upon by terminating cleanly. The security criterion can be handled by running programs under a *dummy* usercode, the previously mentioned `slave`, which is denied all possible permissions, including a home directory. Furthermore, during testing, a temporary directory whose name is randomly generated, is created within the top-level `slaves` directory in which to store a *copy* of the program, which is then used for testing. This temporary directory has read and write access denied to *all* users, including `slave`, so preventing discovery of its location by a malicious program.

SUBMISSION AND TESTING

The core of the system consists of three modules. First, the `submit` module fulfils the function of copying a student's assignment to the relevant location in the directory hierarchy. The second module, `onetest`, runs a program against a set of data, reporting the results back to the user. Third, the `mark` module allows an instructor to enter a final mark for a student into an SQL database. Additionally, there are a number of smaller modules, with limited functionality, which typically call `onetest`, and which are used to implement the full Graphical User Interface (GUI) as described below. In this section, the first two of these modules is described.

Coursework submission

The main functional requirements for coursework submission include the selection by a student of the files containing coursework they wish to submit, and the copying of those files to a location where they can subsequently be accessed by authorised staff.

The coursework submission module forms a set-uid program owned by `boss`, with arguments of the student username, the course, assignment and exercise numbers, and the names of the files to be submitted. The main control algorithm of the program is shown in Table II, with any error being fatal, causing the program to terminate immediately with a message displayed to the student. The effective user-id changes between the student and `boss` repeatedly depending on whether the files in the student's file space are being accessed, or data in the *BOSS* store, the latter being minimised.

Table II. Algorithm for coursework submission

- **if** username is valid **then**
  establish corresponding ID
  else
  error
- **if** course, assignment and exercise are not valid
  or submitted files are not readable **then**
  error
- **if** settings file does not permit submission **then**
  error
- **if** student has already submitted **then**
  if not allowed to re-submit **then**
  error
  else
  move previous submission to back-up location
- create directory in BOSS file space for new files
- **for each** file
  copy file to BOSS file space
  calculate authentication code for each file
- create receipt file in BOSS file space
- calculate authentication code for receipt
- add entry to log file
- email receipt to student

The algorithm starts by checking several items of data before proceeding. First, the username is checked for validity, and if it is valid, the corresponding University identification number is established. Then, the course, assignment and exercise numbers are checked for validity, and the specified filenames are checked for read permission. At this point, the assignment `settings` file is checked for submission permission at this date and time by the student. If the student has already submitted, then the assignment `settings` file must be checked for permission to resubmit, and any resubmission must cause the previous submission to be moved to a backup location. A new subdirectory of the appropriate `solutions` directory named by the University ID number is then created, and each file to be submitted is copied to that subdirectory. The final part of the submission process requires an authentication code for each file to be calculated, a receipt file and an authentication code for the receipt to be created, an entry to be added to the log file, which includes the receipt authentication code, and an email message containing the receipt to be sent to the student.

Program testing

As described earlier, one of the key problems in dealing with programming assignments is that it is difficult to assess the correctness of a program simply by inspecting a paper copy. On-line submission of assignments not only streamlines the administration of submission, it also facilitates what might be regarded as the primary requirement of such a system, the automated testing of a program against specified data sets. This is particularly important, since it enables different parts of a program to be tested, and the program's limits to be explored, by providing multiple data sets and running multiple tests.

Modes of testing

The testing module is used in two circumstances. First, it is used during the process of assessment, which is described in more detail later. Secondly, students can access the testing program with a *single* data set for each exercise, so that they can try out a program prior to submission. The aim is not for the module to assist substantially in their debugging activity, rather to ensure that a program they *think* works in fact does so. It is not unusual for programs to generate output containing, for example, control characters that are not seen when the program is run, but which the testing module would notice. Also, since students can modify their system environments – and this is especially common when using UNIX – it is possible for a program to run correctly for a student and fail under the configuration used by the testing program.

The testing process

A testing protocol for each submitted exercise is needed. First, a collection of data sets containing the input data and the corresponding expected output must be created. Then, a *test harness* satisfying two constraints additional to the security conditions detailed above is used to run programs against the data. First, the test harness must run *automatically* without user intervention, otherwise it would generate its own administrative overheads. Secondly, for subsequent inspection to take place, information must be stored regarding failed tests.

The *expected output* from a program is stored as a set of files, including two files for the standard output and standard error streams. Simply by use of indirection, a program requiring interaction via the standard input and output streams can run without manual intervention. The actual output from a program is compared file-by-file with the expected output. If an *exact* match between the two is needed, a utility such as `diff` is used to perform the check. For more complex processing, a short UNIX shell script can be written to preprocess the submitted program's output prior to comparison with the expected output files as, for example, if small variations between the files regarding whitespace or case sensitivity can be ignored. The results of applying `diff` are used, after processing, to generate reports on the differences between the files.

Comparison of the actual and expected outputs has perhaps been the most problematic part of the software, however. First, any non-printing characters in the input to `diff` cause it to terminate immediately, with a non-zero exit status. Unfortunately, especially when using Pascal or C, it is very easy for a program to generate unexpected control characters in its output. Second, for the marker, and consequently the student, to appreciate the correctness of a program, they must know exactly how the files differ. However, presenting the output of `diff` in such a form as to display that information in a clear and simple to understand form is not an easy task.

The possibility of automatically assigning marks to programs that are partially correct in terms of output is one that was considered, but which has no good solution. In general, where the output generated by a student's program differs from the expected output, it must be inspected manually to determine the nature of the inconsistency. There are, however, some cases where minimal differences such as incorrect formatting, incorrect case of letter, etc., might lead to partial marks, and these might easily be incorporated through the `diff` options. Similar utilities might be developed for enhanced capabilities in this respect as, for example, suggested by Reek [2]. Nevertheless, the general case is still problematic, and assessing the correctness of output when it differs more than just minimally from the expected output can

be a sophisticated issue of judgement for which automatic processes are unsuitable.

The test harness is used by the instructor to run tests on an unlimited number of data sets. Using a setuid program, the results of the tests are stored in files in the same directory as the student's submitted work. Typically, this program is run once, after the deadline for the assignment in question has passed.

Assessment

It is becoming increasingly important that the quality and consistency of marking should be high and should be seen to be high. Double-marking is often desirable, and at some institutions is becoming mandatory. Once a marking scheme has been chosen for a given exercise, marks awarded must be justifiable *relative to that scheme*. The *BOSS* software implements a relatively strong version of this paradigm by distinguishing between a *marker* and a *moderator* for an exercise.

Running the tests

The first stage in the marking process is to run the tests on all submitted coursework for a given exercise, as described above. This may be done by any authorised staff, and can be repeated later if, for example, some students have been allowed to submit late, in which case only newly submitted coursework will be tested.

Assuming that tests have been run on a set of programs submitted by a class, the results of the tests are available to assist in the allocation of marks for that piece of work. While automation of this allocation of marks reduces the time and effort involved, there may be individual circumstances where the automatically generated test results require human intervention. For example, a failed test might be due to an obscure system bug or *feature* for which it would be inappropriate to penalise the student. Alternatively, a program that *almost worked* might be worthy of partial credit.

Using the test harness discussed above, an instructor can run tests on all work for a given exercise, on multiple sets of data. The results of the tests are then made available via either a text-based or a graphical interace. The latter allows the instructor to specify other marking criteria beyond the automatic tests, and for a marker therefore to assign marks without recourse to a paper marksheet. Several independent markers can be involved to facilitate the double marking increasingly required by many institutions.

Marksheets

Instructors simply need to specify the categories for which marks are awarded, and the weight attached to these categories, and a graphical marksheet is constructed. The marksheet integrates marks resulting from running and testing the program with those relating to other aspects of the program, such as style, for example.

The marks resulting from the automatic tests, by which a student's program is run on several sets of data and the output compared with expected output, are incorporated into the marksheet directly. If the output is correct and the program passes, then full marks for that category are declared on the marksheet. If the program fails, then no marks are awarded, but the tutor or instructor may subsequently adjust the automatically assigned marks to give either full, half or zero marks. This is shown by the bottom four mark categories on the marksheet of Figure 3, which shows a typical window presented to a marker.

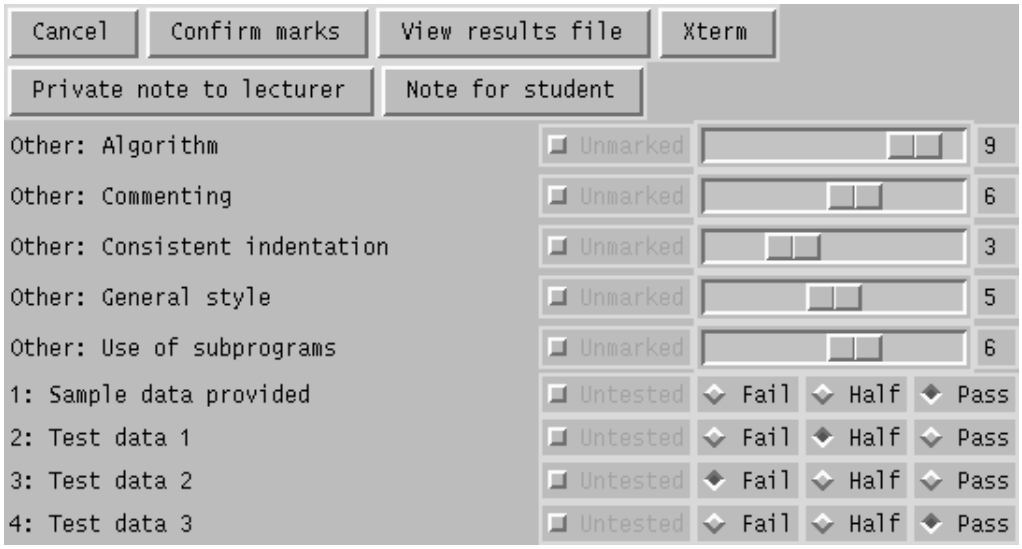

Figure 3. Electronic marksheet

Each marking category is assigned a weighting by the instructor, and this is hidden from the individual markers to prevent bias. The other capabilities, accessed by the buttons at the top of the window, allow the marker to perform the following actions:

(a) Examine the output from the testing program.
(b) Start a UNIX shell in a separate window and in a new, temporary directory into which all the student's files have been copied, so that manual examination, and execution if need be, can take place.
(c) Write a note to the instructor if there are any matters that the marker considers should be brought to their attention.
(d) Edit a file containing feedback for the student.

The remaining categories of marks are awarded by the instructor interacting with the marksheet and moving the slider along on a scale of zero to ten. Only when this mark is combined with the weight, which is not shown to the marker, is the final mark calculated. This allows independent assessment of various aspects of the program without the marker being biased by the number of marks to be awarded. Before a category is assigned a mark, the *unmarked* box is highlighted so that it is obvious which parts of the marksheet need addressing. At present, these marks are awarded manually, but it is possible for various automated measurements of source code to be made to arrive at concrete indicators of programming style such as modularisation, commenting, consistency of indentation, and so on [13–15].

Subsequently, a *moderator*, who is typically the course organiser, has access to all of the first pass marks in order to arrive at a moderated mark *for each marking criterion*. The moderation window is illustrated by Figure 4, which shows the relevant buttons at the top as before, but now displays the marks of the individual markers, here named by usercodes `smith` and `jones`, as well as the automatically assigned marks, indicated by `auto`. On the right-hand side of the window, the system offers a suggested average as the final mark, which

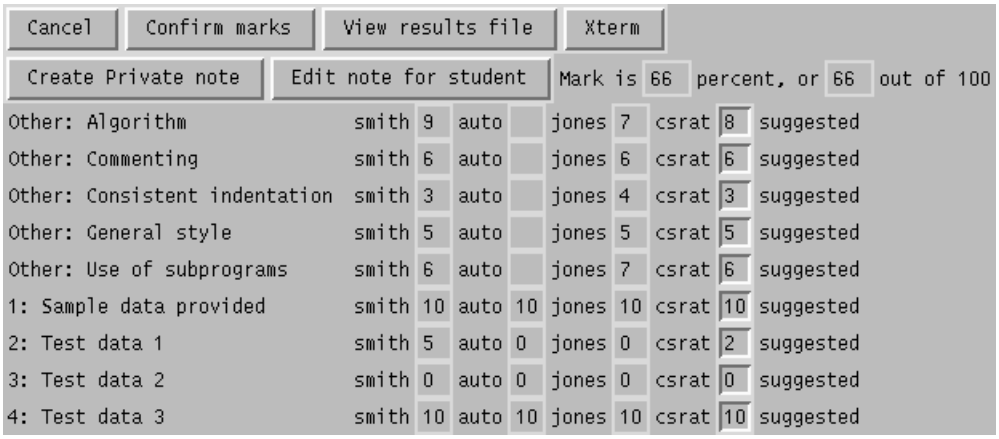

Figure 4. Electronic moderation sheet

the moderator, with usercode `csrat`, can adjust if appropriate. The final marks are shown in the top right corner.

Once a grade for a student has been established for a given exercise, that grade is stored in an SQL database, which also contains data such as a student's name, department and degree programme. Security here is twofold. Not only does the security of the *BOSS* system forbid access to the database by unauthorised users, but the database itself restricts access to data. There is thus double confidence that marks are kept confidential. The data in the database can be used to produce marksheets for both the instructor and examination secretaries, or other administrators as necessary, minimising the administrative time needed for the collation of grades.

PLAGIARISM DETECTION

There is a danger inherent in any on-line submission system, that some weaker or dishonest students may be tempted to copy, and edit, each other's work prior to submission. Within the *BOSS* system, software has been included, called SHERLOCK, to assist in the detection of similar programs submitted for the same exercise.

The approach adopted, which we call *incremental comparison*, involves the comparison of each pair of submitted programs five times:

(a) in their original form;
(b) with the maximum amount of whitespace removed;
(c) with all comments removed;
(d) with all comments *and* maximum amount of whitespace removed; and
(e) translated to a file of *tokens*.

A *token* is a value, such as `name`, `operator`, `begin`, `loop-statement`, that is appropriate to the language in use. The tokens necessary to detect plagiarism may not be the same as those used in the parser for a real implementation of the language – it is not necessary to parse a program as accurately as a compiler. The scheme will work even with a very simple choice of tokens, and a rudimentary parser, and it is simple to update it for a

new programming language. Each line in the file of tokens will usually correspond to a single statement in the original program.

If a pair of programs contains similarities, then it is likely that one or more of these comparisons will indicate as much. By examining the similarities and the corresponding sections of code in the original program, it should be possible to arrive at a preliminary decision as to whether the similarities are accidental or not. This scheme has been implemented in the SHERLOCK utility, which allows an instructor to examine a collection of submitted programs for similarities. SHERLOCK is described in detail elsewhere [16].

There are other approaches to plagiarism detection. *Attribute counting* [17–19] involves assigning to each program a single number capturing a simple quantitative analysis of some program feature; programs with similar attribute counts are potentially similar programs. Similar, but more complex, techniques include cyclomatic complexity [20] and scope number [21]. *Structural comparison* of programs [22,23] is a potentially more complex procedure than comparing attribute counts, and depends fundamentally upon the language in which the programs are written.

Whale [24,25] and Verco [26] have carried out a detailed comparison of various attribute count and structure comparison algorithms. They conclude that attribute count methods alone provide a poor detection mechanism, outperformed by structure comparison, while the structure comparison software developed by Whale, *Plague* [24] and Wise (*Yap*) [27] report a high measure of success. Although the software discussed by Whale was not available, SHERLOCK has been run on some of the data sets for which the results generated by Plague were available. In that instance, SHERLOCK detected *all* the instances of possible plagiarism found by Plague, and more, and it appears that the effectiveness of SHERLOCK is similar to that of Plague. Furthermore, attempts to deceive SHERLOCK by running it on test data failed.

Use of SHERLOCK has decreased the amount of detected plagiarism in our department. In the first year of its use in 1994, out of more than 550 submissions for programming assignments, over 6.5 per cent were both detected by SHERLOCK and subsequently established as genuine instances of plagiarism. Two years later this had fallen to under 1 per cent. The number of *false hits* is usually small.

It is clear that the volume of *detected* plagiarism has decreased substantially. This is due either to a reduced level of plagiarism, or to a greater proportion of students being able to hide the changes they have made. The latter, as has already been remarked, is a difficult exercise, and we therefore claim that the incidence of plagiarism has decreased.

USER INTERFACE

The first version of the program featured a text based interface only, and the paradigm an instructor would use would be to view the text files containing the marks and transcribe them to a paper marksheet. A GUI was considered but rejected on the grounds that with the available tools – the relatively low-level *Xlib* graphics toolkit – the time necessary to develop the GUI would be disproportionately long. The current user interface is a GUI written in Tcl/Tk [28], which was relatively straightforward to implement. Unfortunately, large programs written using Tcl/Tk are difficult to maintain due to the fact that it is an easy-to-use scripting language with few modular syntactic features.

Nevertheless, the GUI has provided us with the opportunity to develop a paper-free environment for staff marking coursework, with the benefit that transcription errors between the *BOSS* software and marksheets are eliminated. Figure 3 shows a typical window presented to a marker, and Figure 4 shows one presented to a moderator. An important observation is

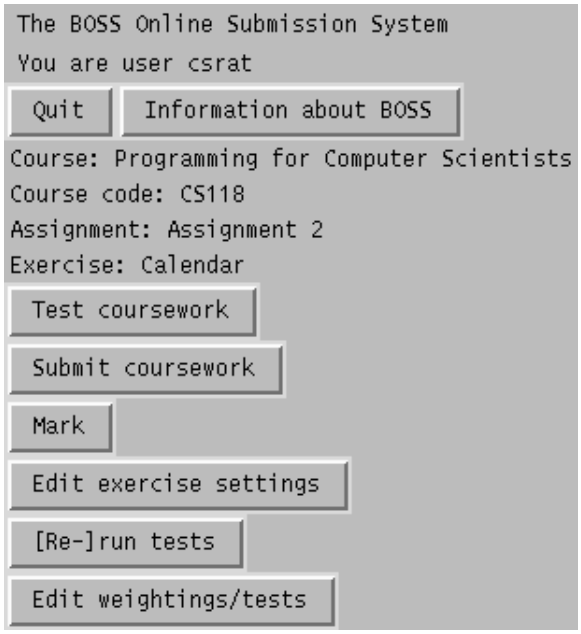

Figure 5. The submission and marking window

that the number of keystrokes required is kept low, so helping both the marking and the moderation phases to be completed rapidly.

The various different parts of the system are treated in a similar way, so that coursework submission requires the student to select the course, assignment and exercise, then select the files to be submitted, and finally either test or submit the work, using a series of interactive windows constructed using Tcl/Tk. Users with a special status are provided with extra options in their windows for marking, for example, or for changing the default parameters in the settings files. Thus, while the interface is consistent in *look and feel* across all users, it provides extra functionality when required.

For example, Figure 5 shows the window that offers the possibility of submitting an assignment, testing it against the public sample data provided, marking the assignment, editing the exercise settings such as CPU time allowed, etc., testing the submitted programs against the unseen data, or editing the weightings attached to each marking category for the exercise. Of course, this is only presented to the markers. For students, the same window would appear with only the first two options of testing or submitting. The window that results from the last of these options is shown in Figure 6 as an illustration of the ease with which the weightings are attached to particular categories of marks. The user interface thus provides a high degree of flexibility in allowing users to tailor the parameters of a particular course to their needs.

CONCLUSIONS

Evaluation

In comparison with the related work considered briefly at the start of this paper, *BOSS* is a significant improvement. In Dawson-Howe's system [3], the student's program is compiled

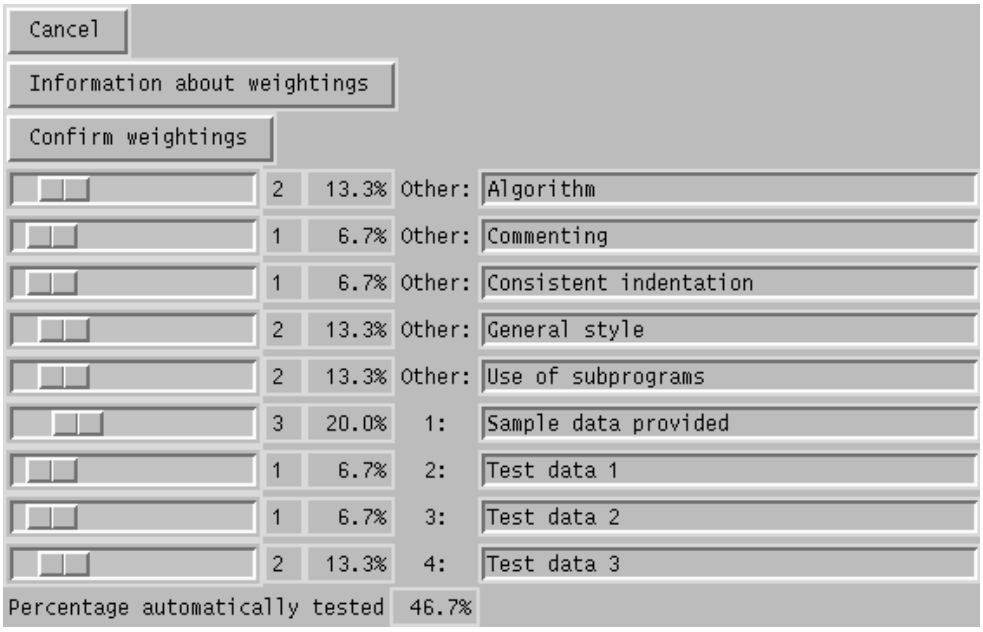

Figure 6. Modification of weightings for marking categories

and executed under the students' supervision, using data suggested by the instructor, with email messages containing the program and its results being sent to the instructor. This avoids security problems, but leaves a large burden on the instructor, because students will invariably make mistakes in the testing process, and submissions will require further evaluation without tools to assist in that process. MacPherson [6] simply transfers ownership of files from students to the instructor at an appropriate time, while Isaacson and Scott [8] require students to place files readable by the instructor in directories under the instructor's filestore. In this latter work, some limits are imposed on CPU time and file sizes, but both of these have minimal security features, and still require significant manual effort in terms of marking beyond efforts to compile and run the submitted programs. Reek's TRY system [2] compiles and runs programs for students against the instructors protected data, comparing results with the expected results, and generating a log file of all such tests. Again, however, student programs are copied to the instructor's filestore for compilation and execution. Ceildih [9] is distinct in that it provides an entirely separate environment in which a student can compile and test programs and, while offering great functionality, insulates students from standard tools and utilities.

By contrast, *BOSS* provides a complete and effective submission system that allows the testing and submission of programs, and their subsequent evaluation by instructors in an integrated fashion. In the standard case, no extra effort is needed to process the submissions, but the capability for them to access and manipulate submissions for further evaluation is provided. Extensive marking facilities and plagiarism detection software are also included. Perhaps most importantly, however, is the great effort made to ensure that security is paramount, and that the possibility of student programs corrupting the instructor's files or doing other damage is minimised. Thus, not only does *BOSS* surpass the range of functionality of the existing systems, it does so based upon a much stronger model of security.

Benefits

The *BOSS* system has assisted us by speeding up and making more accurate the process of assessing students' programming coursework. Over the course of six years, it has been used successfully on several first year programming courses using Pascal and the UNIX shell, each attended by between 100 and 200 students, and by students on second-year courses on software engineering using C++, and programming with automata using `lex` and `yacc`. By adopting a limited but well-defined set of criteria, and ensuring that the software meets the specifications, we have created a robust and efficient system that minimises the repetitive administrative tasks faced by instructors.

As it stands, the system is functioning well. There has been a generally favourable student response, and this has improved as the culture of automatic submission has become established within the Department. Specifically, students have had virtually no difficulty in using the system. This seems to imply that the newly established culture has taken root, and that initial efforts at integrating the system into the fabric of the degree courses are paying off. In addition, instructors have also found the system to be simple and easy to use, and marking times have been reduced significantly with a corresponding increase in consistency throughout.

The *BOSS* system has provided us with a number of benefits without compromising the general approach taken of maximising exposure to standard tools and utilities. Large numbers of students have been handled efficiently by the system, with security of assignment submission being assured. Programs submitted cannot be copied by other students, and the possibility of paper submissions being accidentally lost is removed. Secretarial staff do not need to be employed at deadlines to collect assignments, making more efficient use of secretarial time, and the volume of paperwork involved can be reduced to almost zero both for the instructor and for administrative and secretarial staff.

More importantly, perhaps, the time needed to mark an assignment is reduced considerably, while the accuracy of marking, and consequently the confidence enjoyed by the students in the marking process, is improved. In addition, consistency is improved, especially if more than one person is involved in the marking process.

User feedback

In terms of students, we sought feedback by means of questionnaires that required students to comment on their experiences of using the system. These were generally favourable, and most students considered it an easy system to use. The ability to use the utility to test programs in advance of submission to check the conformance of their programs to the specification was also widely appreciated.

The principal concerns expressed fell into two categories. The first of these covered minor criticisms about the user-interface and the specific messages that the system provides to students when a program fails the test utility. Many of these criticisms have since been addressed in the latest version of the *BOSS* system, and development is continuing so that the user-interface is improved still further. The second – and perhaps more interesting – category of criticism was that the output expected was *too precisely specified*. *BOSS* is far too 'fussy'. This criticism relates to the format of the output specified – as in the precise layout of tabular output, for example – and also to some students' desire to design their own user-interfaces by establishing interactive prompting for input. This is an important point, for it seemed to reflect the preference of first year undergraduates who had considerable programming experience

prior to joining the course. Many of them were thus used to programming in an unstructured fashion, and were unused to being required to follow precise specifications. This seems to imply that resistance to the system is only significant in those students who have already adopted particular styles of programming that may not be appropriate for the particular problem at hand, but who may have preconceptions about the nature of programming.

Standards and constraints

The core modules themselves were coded in ANSI C [29] using only system library calls as specified in the POSIX standard [30], and great care was taken to follow those standards and to perform exhaustive compile-time and run-time checks on the code. The resulting programs passed both Sun Microsystems' `cc` compiler and the Free Software Foundation's `gcc` compiler cleanly with maximum checking enabled. It is interesting to observe that, although roughly half the code is concerned with mundane tasks such as ensuring all function return values are as expected, the resulting programs required minimal debugging. Furthermore, the programs were able to be ported to machines running different versions of the UNIX operating system with no changes being required to the code.

Future work

We are conscious of the rapid changes in technology affecting the discipline of programming, and the impact of the Internet on users' interaction with computers. In consequence, these tools are being actively developed, via the use of Java [31], to create a networked and platform-independent version of *BOSS*, in order to maintain step with these paradigm shifts.

One unanswered question that will become ever more pressing, is how a system such as *BOSS* can be adapted to handle arbitrary input and output, rather than just being text-based. As the use of windows, icons and other graphical devices becomes the normal paradigm for communicating with a program, the functionality of a program must be specified in such a way that its output can be accurately and automatically checked. It may be possible to replace arbitrary GUI front ends with different ones to be used in testing, but this can constrain the nature of the programs themselves. Such constraints may, nevertheless, be a small price to pay for the benefits of a system that can adequately address the problem of submission and assessment of student programs in a secure and effective fashion.

ACKNOWLEDGEMENTS

The authors wish to thank Geoff Whale for providing the test data and William Smith and Chris Box for the initial software development.

REFERENCES

1. D. G. Kay, P. Isaacson, T. Scott and K. A. Reek, 'Automated grading assistance for student programs', *ACM SIGCSE Bulletin*, **26**(1), 381–382 (1994).
2. K. A. Reek, 'The TRY system – or – how to avoid testing student programs', *ACM SIGCSE Bulletin*, **21**(1), 112–116 (1989).
3. K. M. Dawson-Howe, 'Automatic submission and administration of programming assignments', *ACM SIGCSE*, **27**(4), 51–53 (1995).

4. M. Joy and M. Luck, 'Software standards in undergraduate computing courses', *Journal of Computer Assisted Learning*, **12**, 103–113 (1996).
5. M. Luck and M. Joy, 'Automatic submission in an evolutionary approach to computer science teaching', *Computers and Education*, **25**(3), 105–111 (1995).
6. P. A. Macpherson, 'A technique for student program submission on UNIX systems', *ACM SIGCSE*, **29**(4), 54–56 (1997).
7. M. J. Canup and R. L. Shackelford, 'Using software to solve problems in large computing courses', *ACM SIGCSE*, **30**(1), 135–139 (1998).
8. P. C. Isaacson and T. A. Scott, 'Automating the execution of student programs', *ACM SIGCSE Bulletin*, **21**(2), 15–22 (1989).
9. S. D. Benford, K. E. Burke and E. Foxley, 'A system to teach programming in a quality controlled environment', *The Software Quality Journal*, 177–197 (1993).
10. S. D. Benford, K. E. Burke, E. Foxley, N. H. Gutteridge and A. Mohd Zin, 'Experience using the Ceilidh system', *Monitor*, **4**, 32–35 (1993/94).
11. R. C. Merkle, 'A fast software one way hash function', *Journal of Cryptology*, **3**(1), 43–58 (1990).
12. B. Schneier, *Applied Cryptography*, Wiley, New York, 1994.
13. R. E. Berry and B. A. E. Meekings, 'A style analysis of C programs', *Communications of the ACM*, **28**(1), 80–88 (1985).
14. S. Hung, L. Kwok and R. Chan, 'Automatic programming assessment metrics', *Computers and Education*, **20**(2), 183–190 (1993).
15. M. J. Rees, 'Automatic assessment aid for Pascal programs', *SIGPLAN Notices*, **17**(10), 33–42 (1982).
16. M. Joy and M. Luck, 'Plagiarism in programming assignments', *IEEE Transactions on Education*, (to appear 1999).
17. G. Rambally and M. Le Sage, 'An inductive inference approach to plagiarism detection in computer programs', *Proceedings of the National Educational Computing Conference*, 23–29. Nashville, TN. ISTE, Eugene, OR 1990.
18. J. Faidhi and S. Robinson, 'An empirical approach for detecting program similarity within a university programming environment', *Computer Education*, **11**, 11–19 (1987).
19. S. Grier, 'A tool that detects plagiarism in pascal programs', *12th SIGCSE Technical Symposium*, 15–20. St. Louis, MI, 1981.
20. T. McCabe, 'A complexity measure', *IEEE Transactions on Software Engineering*, **2**(4), 308–320 (1976).
21. W. Harrison and K. Magel, 'A complexity measure based on nesting level', *ACM SIGPLAN Notices*, **16**(3), 63–74 (1981).
22. S. Robinson and M. Soffa, 'An instructional aid for student programs', *ACM SIGCSE Bulletin*, **12**(1), 118–129 (1980).
23. K. Magel, 'Regular expressions in a program complexity metric', *ACM SIGPLAN Notices*, **16**(7), 61–65 (1981).
24. G. Whale, 'Identification of program similarity in large populations', *The Computer Journal*, **33**(2), 140–146 (1990).
25. G. Whale, 'Software metrics and plagiarism detection', *Journal of Systems and Software*, 131–138 (1990).
26. K. L. Verco and M. J. Wise, 'Plagiarism à la mode: A comparison of automated systems for detecting suspected plagiarism', *The Computer Journal*, **39**(9), 741–750 (1997).
27. M. Wise, 'Detection of similarities in student programs: Yap'ing may be preferable to plague'ing', *ACM SIGCSE Bulletin*, **24**(1), 268–271 (1992).
28. J. K. Ousterhout, *Tcl and the Tk toolkit*, Addison-Wesley, Reading, MA, 1994.
29. ANSI, *Programming Language – C*, American National Standards Institute, New York, NY 1990.
30. IEEE, *Information Technology – Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language]*, IEEE, New York, NY 1990.
31. M. Campione and K. Walrath, *The Java Tutorial*, Addison-Wesley, Reading, MA, 1996.

