An Automatic Evaluation System with a Web Interface

Chittaranjan Mandal
Chittaranjan.Mandal@iitkgp.ac.in
School of IT, IIT Kharagpur, India

Christopher M P Reade
Chris.Reade@kingston.ac.uk
Kingston Business School, Kingston University, UK

Vijay Luxmi Sinha
Vijaya.Luxmi.Sinha@sit.iitkgp.ernet.in
School of IT, IIT Kharagpur, India

Abstract

We describe a scheme and implementation for the automatic evaluation of programming oriented assignments. The implementation described covers both the internal working of the automatic evaluation and the web interface. Presently, this system can handle only C programs.

Keywords: automatic evaluation, course management, large class

1. Introduction

In this paper we describe a scheme and implementation for automatic evaluation of C programming assignments.

The scheme was motivated by the need to handle programming assignments with large cohorts of students (approximately eight hundred students). In a semester each student is required to complete about nine assignments and three exams. That amounts to 9600 submissions of programs of varying levels of complexity. Even after the load is distributed among 21 evaluators, each person is required to evaluate over 450 programs per semester. This is a particularly difficult task if the work is to be done assiduously.

The work on the scheme is a continuation of the development of our web-based course management tool (Mandal et al, 2004). WBCM essentially helps an instructor to disseminate assignments, put up course material, collect student submissions and evaluate these on-line to mark them and to provide feedback. The tool also has a number of other

interfaces to handle auxiliary tasks such as adding or deleting courses, populating the course with students, creating a compact snap-shot of class performance, etc.

There is considerable prior work on automatic and semi-automatic evaluation of programs and we discuss some of the key approaches reported in the literature. Although our evaluation scheme is not particularly generic (currently only dealing with C programs) it is designed for simplicity of use and is implemented with a system that is usable over the web.

The paper explores some of the limits of what can be automated and how this can impact on the design of assignments. It also discusses the associated benefits. In particular: (i) the savings in evaluation effort which more than compensate for the increased effort involved in careful design of assignments that can be automatically evaluated; (ii) benefits for rigour and consistency when replacing subjective judgements with objective, automated evaluation, and (iii) the potential for quick and detailed feedback being returned to the user immediately after assignment submission. (iv) the potential for reusing the effort to design an assignment.

The work also focuses on four key areas of concern that were considered and dealt with in the design of the scheme.

1. Choices for automation of the evaluation process
2. Security of the process
3. Marking issues
4. Overheads for assignment setup

In the following section we discuss related work before outlining our own approach. We then use a running example, which we use to illustrate more details of our system. The final section discusses conclusions and further work.

2. Related Work

A variety of systems have been developed for automated and semi-automated evaluation of programming tasks. The number of systems is clearly influenced by the nature of programming tasks, where there is a need to run programs to inspect behaviour as well as to inspect source code. A realistic inspection of behaviour also requires a significant number of tests, but the potential for automation of testing is considerable.

Some noteworthy earlier systems include TRY (Reek, 1989) - a Unix based utility that grades PASCAL assignments by testing the correctness of the final result of the program; ASSYST (Jackson and Usher, 1997) a software tool providing a graphical user interface to help tutors oversee and direct the assessment of Ada programs; Scheme-robo (Saikkonen et al, 2001) which checks for correctness of functions by comparing return values against a

model solution; and Ceilidh (Benford et al, 1993) which is a large system that was quite widely used in the mid 1990's and subsequently evolved into the 'CourseMarker' system. More recent systems such as GAME (Blumenstein et al, 2004), Submit (Pisan et al, 2003), and (Juedes, 2003) have addressed web-based design for increased universality (ease of deployment, wider access and platform independence). Some of the recent systems such as Baker et al (1999), Pisan et al (2003), Juedes (2003), focus on providing rapid feedback for students to practice whereas others such as Luck and Joy (1999), Benford et al (1993) are integrated with complete course management systems and are concerned with information management and keeping records rather than just feedback. Some of the systems such as Jackson and Usher (1997), Luck and Joy (1999), Juedes (2003) do not attempt full automation, but are designed to assist human evaluators.

What can vary significantly in these systems is what they attempt to evaluate. Most (Jackson and Usher 1997, Saikkonen et al 2001, Baker et al 1999, Blumenstein et al 2004, Pisan et al 2003, Juedes, 2003) attempt to test or evaluate more than simple behaviour of single programs, but analysis of structure is still very difficult and usually quite limited. The nature of the evaluation is often linked to whether components are considered as well as whole programs and whether more then one programming language can be considered. For example GAME (Blumenstein et al, 2004) is designed with genericity in mind, but in the context of Java, C, C++ programs, only simply style and structure issues are considered (such as commenting, indentation, number of functions/methods).

Other issues include: (i) Security - this is specifically addressed by Luck and Joy (1999) and this covers robust environments, privacy, and data integrity; (ii) Plagiarism detection - this is discussed as a useful additional evaluation tool by Blumenstein et al (2004) and Luck and Joy (1999).

3. Our Approach

Our focus is on extending the types of evaluations supported, flexibility and ease of setting up new assignments and evaluations. In particular, we address the role of performance evaluation and the assignment set up costs. The system is designed for use with very large numbers of assignment submissions, so complete automation rather than grading assistance is addressed.

3.1 Whitebox testing

A particularly important decision is whether submissions should be treated as single 'black boxes' with only overall input/output behaviour to be tested; 'grey boxes' that allow components and functions to be exercised, or 'white boxes' which allow structure as well as behaviour to be evaluated. The first approach is, in general, unworkable in a class where students are learning to program. This is because conformance to overall input/output requirements is particularly hard to achieve and evaluations, which relied on this, would exclude constructive evaluation for a majority of students. The other approaches involve exercising individual functions within the student's programs and are, unfortunately,

language sensitive. This entails per language support for automatic evaluation. At present we have restricted ourselves to programs written in C.

We have chosen the *whitebox* approach which is more general than the *blackbox* approach to testing programs. Blackbox testing does not help the tester determine whether the program has been written in a particular way, following a particular algorithm and using certain data structures. But those are essential objectives for evaluating student programs.

3.2 Performance and marking

Evaluation of behaviour may need to work with several objectives which we may classify under: (i) evaluating correctness - which covers both correctness of overall behaviour and correct behaviour of components such as functions/procedures; and (ii) evaluating performance - which covers the run time of the program and the space usage of the program.

However, it is important to note that these classes of objectives are related. Consider, for example, the problem of sorting numbers. There are several methods of sorting numbers, varying in time and space efficiency. The main sorting techniques are: bubble sort (in-place, but inefficient); merge sort (efficient if extra space is used but inefficient if an in-place version is used); quick sort (in-place and usually very efficient); heap sort (in-place and efficient). If students are required to implement a particular strategy, it will not be enough to test for correctness alone. Intermediate results may be checked to determine that a particular technique has been used, but that requires additional effort on part of the teacher setting the assignment. In some cases an analysis of performance could provide sufficient additional information to determine the acceptability of a submitted assignment.

From a marking point of view, we need to consider the balancing of assessments for partially working programs or parts of programs, as well as considering performance based marking.

3.3 Overheads for set up

The overheads for assignment set up to allow for automatic evaluation include: (i) overall design in terms of functions/parts that can be specified for separate evaluation, and (ii) supplying a minimal body of code to exercise the parts/functions supplied in a submission. Our system provides support to reduce the work that an instructor needs to do to set up an assignment and it also encourages re-use of set-up for other assignments.

3.4 Security

Security is a non-trivial concern in our system because automatic evaluation, almost invariably, involves executing potentially unsafe programs. One cannot rule out the possibility of a malicious program or one that accidentally causes damage to the execution

environment. This system is designed under the assumption that programs may be unsafe and executes programs within a 'sand-box' using regular system restrictions.

We discuss more details of the system below with the aid of a running example.

4. An Example Assignment Problem

C programs consist of a `main` function and possibly several other functions. A desirable programming practice is to code specific activities as appropriate functions. The `main` function serves as the entry point to the program from where the highest-level activity may be performed directly or achieved by invoking a function that performs that activity.

Suppose we would like the students to do an assignment on *merge sort*. An important step here is merging two sorted arrays so that the resulting array is also sorted. Thus the student may be asked to do the merging in a separate function. The overall mergesort may be performed by another function (say `mergesort`) that invokes the `merge` function.

Thus the assignment needs to be defined so that the student is required to perform different activities in different functions. This gives the tester the ability to invoke the constituent functions of the program so that they can be tested independently. The student may use another sorting technique to merge the two sorted sub-sequences instead of directly merging them. This makes the procedure highly inefficient and contrary to the objectives of *merge sort*. With a black box testing of the `merge` procedure this fallacy would not be revealed. At least two solutions could be given to this problem.

For the first solution, the requirements of the `merge` routine could be modified to stop after merging k of the elements. That way the intermediate results as the merging progresses could be observed. This would easily enable automatic identification of faulty merge routines. For the second solution the time to perform merging the two arrays could be monitored. Merging takes time that is directly proportional to the total number of elements being merged. The sorting techniques require a much longer time to sort a given sequence of elements.

A question that naturally arises is which of the above two possibilities is more suitable for automatically evaluating programs? The difference between the two approaches is very clear. The first approach requires the assignment designer to write down routines to carry out the necessary checks. The second method is more amenable to automatic handling. Fixed routines could evaluate the time it takes for a function to execute on supplied data. The runtime for various sample sets could be compared against a (known) reference function, to determine whether the runtimes are acceptable. This results in saving human effort to develop the testing procedure. The complexity of testing procedures of the first category will vary with the problem at hand. In cases where the runtime alone cannot reveal the correctness of the applied technique, the second approach should not be used.

5. Use of the System

5.1 Student interaction with the system

A marking scheme needs to be defined. It is desirable to have some comments or justification associated with marks awarded or deducted. Such comments will help the student to understand how the program has been evaluated. Corrective action may be taken on the basis of the comments. Since this scheme is integrated with WBCM (Mandal et al, 2004), students have the option to resubmit the assignment within the deadline for the assignment. This will typically be the case, as initial submissions by students are often faulty.

Typically, after the first round of submission there will be compilation errors. Compilation errors could be reported even if the student succeeded in getting his/her program to compile and run. This is because the student might not have conformed to the function prototypes given in the specification. At the time of testing additional code is compiled along with the user code. This additional code will expect certain functions, conforming to the prototype supplied in the problem specification, to be available in the student's submitted code. If those functions are absent or if they are present and do not conform to the required prototype the compilation will fail. The student will receive appropriate feedback, indicating details of the compilation problem, as generated by the compiler. The student will then have to correct his code so that required functions are in the appropriate form. On compilation failure, later tests cannot be undertaken. However, the assignment can be designed to avoid compilation problems at the time of testing. This is illustrated later.

In the later phases of testing, failures for specific test cases could occur. If we consider the example of *merge sort* the test cases could be: (i) successful merging by the `merge` function; (ii) successful sorting by the `mergesort` function. If the first test, for successful merging, fails completely then the testing for the later parts could be aborted. However, if the merging is correct but inefficient, testing could proceed. In this case marks allotted for merging may not be awarded. However, marks could be awarded if the second test goes through properly, indicating that decomposition has been done correctly. In each case, appropriate feedback is generated to indicate the outcome of the test and how marks have been awarded as a result of the test.

5.2 Assignment design

This is essentially a creative activity on the part of the instructor. We illustrate the process by means of the *merge sort* problem.

The scheme for *merge sort* may be expressed compactly by the `mergesort` function given below:

```
/************************************/
mergesort (int *a, int size, int *temp) {
int m;

if (n==0 || n==1) /* nothing to do */ return;
m = n/2;
mergesort (a, m, temp);       /* sort the first m elements */
mergesort (a+m, n-m, temp);   /* sort the next  m elements */
merge (a, m, a+m, n-m, temp); /* merge the two sorted sub-sequences */
transfer (a, temp, n) ;       /* transfer n sorted elements from temp to a */
}

merge (int *a1, int n1, int *a2, int n2, int *a3) {
if (n1==0 && n2==0)    return;
if (n1==0 || *a2<*a1)  { *a3=*a2 ; merge (a1, n1, a2+1, n2-1, a3+1); return; }
if (n2==0 || *a1<=*a2) { *a3=*a1 ; merge (a1+1, n1-1, a2, n2, a3+1); return; }
}

transfer (int *a, int *temp, n) {
if (n) { *a=*temp; transfer(a+1, temp+1, n-1);
}
/************************************/
```

Our aim is to test that the student writes the `mergesort` function correctly and the `mergesort` function correctly. The `transfer` function is trivial and need not be tested.

5.3 Assignment plan

We decide to test the *merge sort* as follows:

Testing *merge sort* (`mergesort`):
(i) Test correctness on random integers
(ii) Test proper invocation to `merge` for proper decomposition

Testing merging (`merge_to_k`):
(i) check intermediate merging results

A possible assignment statement for the *merge sort* problem is given in figure 1. A `Makefile` and auxiliary programs are supplied as part of the assignment statement to make it easier for the student to reach submission stage with a program that will pass through the automatic evaluation scheme without compilation errors.

Write a program to sort a given array (sequence) of *n* integers by **merge sort**.
The prototype for the mergesort function should be:

```
  mergesort (int *a, int size, int *temp);
```

This function should invoke the function merge_to_k to merge two given sorted arrays a1 with n1 elements and a2 with n2 elements into a third array a3. The function should return after k elements have been placed in a3. The following prototype is to be used:

```
  merge_to_k (int *a1, int n1, int *a2, int n2, int *a3, int k);
```

Within mergesort the merged elements should be transferred back to a from temp using the transfer function given below.

```
/************************************/
transfer (int *a, int *temp, n) {
if (n) { *a=*temp; transfer(a+1, temp+1, n-1);
}
/************************************/
```

You should write your main function in file main.c, the mergesort function in a file mergesort.c and the merge function in a file merge.c. You are advised to compile your program by means of the make command using the Makefile given below. Use the supplied files for prot_merge_to_k.c and prot_mergesort.c.

```
************** Makefile ***************
all: main prot_merge_to_k prot_mergesort
       rm -f prot_merge_to_k prot_mergesort

main:            main.c mergesort.o merge_to_k.o
       cc -o mergesort main.c mergesort.o merge_to_k.o

prot_merge_to_k: test_merge_to_k.c merge_to_k.o
       cc -o test_merge_to_k test_merge_to_k.c merge_to_k.o

prot_mergesort:  test_merge_to_k.c merge_to_k.o
       cc -o test_mergesort test_mergesort.c mergesort.o
************************************

/********* prot_merge_to_k.c *********/
main() {
int *a1; int n1; int *a2; int n2; int *a3; int k;
merge_to_k (int *a1, int n1, int *a2, int n2, int *a3, int k);
}
/************************************/

/******** prot_mergesort.c ***********/
main() {
int *a; int size; int *temp;
mergesort (int *a, int size, int *temp);
}
/************************************/
-----------------------------------------------------
```

FIGURE 1: Assignment statement

5.4 Function invocation and trapping

A most elegant and general trapping mechanism for doing this is seen in the unix/linux program `strace` for tracing system calls and signals. We may ask the student to write certain functions in different files. That way, at the time of testing, various programs can be built by compiling different sets of files from the student and from the files supplied by the teacher to assist automatic evaluation of the assignment.

5.5 Input generation for testing

For effective testing we cannot rely on fixed data. This makes the testing procedure vulnerable to *replay attacks*. While it is possible for the instructor to write a program to generate all necessary random inputs, the system provides some assistance in generating random inputs for testing purposes. Three types of inputs are currently supported: integers, floating point numbers and strings. The generation can be performed subject to several options, as shown below. The first option of array or single enables the generated values to be all returned in a single array or individually in each iteration of a loop.

```
random integers:
  (array / single)
  (distinct / non-distinct)
  (un-sorted / sorted ascending / sorted descending)
  (positive / negative / mixed / interval)
random floats:
  (array / single)
  (distinct / epsilon-apart)
  (un-sorted / sorted ascending / sorted descending)
  (positive / negative / mixed / interval)
strings:
  (array / single)
  (distinct / non-distinct)
  (fixed length / variable length)
```

The type of inputs to be generated is with respect to the immediate scope of a loop. We illustrate this by means of an example in figure 2. The XML specification represents a test that invokes `test_merge_to_k` 50 times with distinct positive integers in arrays `a1` and `a2`. If all the invocations to `test_merge_to_k` are successful then the test passes.

The result of processing the XML specification is a C program that can carry out the test. The instructor is saved the additional burden of writing a program that is compatible with the requirements of the evaluation system. Instead, the instructor only has to write the test function and the XML specification, for the particular test.

5.6 Test specification

The original WBCM had a web interface for setting up submission details. Far too many inputs needed to be specified for automatic evaluation to make a web interface a good choice for loading the specifications directly. Instead, the web interface just takes an XML file where the complete requirements are entered. This XML files specifies what files need

```
<!--------- input generation and test invocation ------------>
<loop loopVar="x" start="1" end="50">
  <input inputVar="a1" type="integer" varType="array" sequence="ascend"
         range="positive" duplication="distinct">
       <array_size>50</array_size>
  </input>
  <input inputVar="a2" type="integer" varType="array" sequence="ascend"
         range="positive" duplication="distinct">
       <array_size>50</array_size>
  </input>
  <distinction_class> a1 a2 </distinction_class>
  <test_function>
     test_merge_to_k (a1, 50, a2, 50)
  </test_function>
</loop>
<!------------------------------------------------------------>
```

FIGURE 2: XML specification for input generation

```
<auto_eval>
  <source_files>
    <item handle="main">
         <text> C file containing only the function: main</text>
       </item>
       <item handle="mergesort">
         <text> C file containing only the function: mergesort</text>
       </item>
       <item handle="merge_to_k">
         <text> C file containing only the function: merge_to_k</text>
       </item>
  </source_files>
  <test marks="10" abort_on_fail="true">
    <text> Evaluation of merge_function </text>
       <exit_status>
         <item value="0" factor="1.0" fail="false"><text>Okay</text></item>
         <item value="1" factor="0.0" fail="false"><text>Improper</text> </item>
         <item value="2" factor="0.0" fail="true" ><text>Wrong</text> </item>
       </exit_status>
    <makefile handle="test_merge_to_k">
main: test_merge_to_k.c <handle_base>merge_to_k</handle_base>.o
       cc -o test_merge_to_k test_merge_to_k.c \
                          <handle_base>merge_to_k</handle_base>.o
    </makefile>
    <testing>
      <!------ input generation and test invocation as above ------>
    </testing>
  </test>
  <test marks="10" abort_on_fail="true">
    <!------ similar specification for testing mergesort ------>
  </test>
</auto_eval>
```

FIGURE 3: XML specification for a test

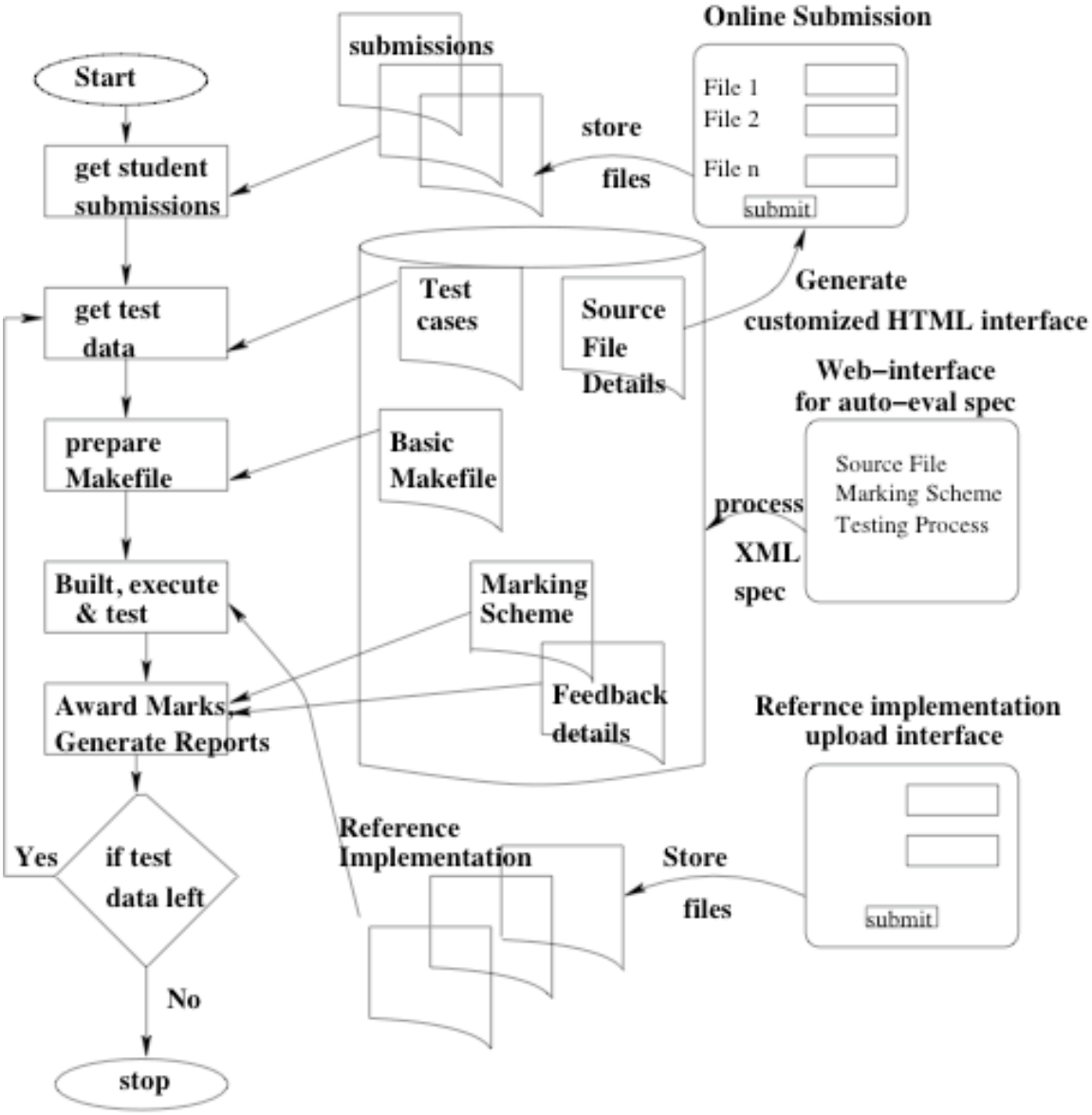

FIGURE 4: Flowchart for Tester Script

to be submitted by the student, what items are going to be tested, the full marks of that item, the marks distribution for testing that item, how the program to test that item will be generated and the testing scheme. Figure 3 shows an example specification in XML and figure 4 shows the flowchart for the generic tester script.

5.7 Sand box

A very important aspect to be taken into account is that the routines to be tested could contain malicious code. For example, the `mergesort` function might have calls to delete arbitrary files. While such programs may not affect the host, as such, they could certainly undermine the automatic evaluation system. Also, a faulty program could run forever, in an infinite loop. Such programs need to be terminated after a certain period of time. Because of the risks of running test programs, those are run under a separate login (test login), distinct from the login under which automatic evaluation is performed.

Fortunately, support now exists to handle these matters rather easily. The situations that are handled by our system are listed below.

1) File deletion: Certain files need to be present in the test login and these should not be deleted as the programs are tested. Such protection is achieved by protecting the files and directories using the `chattr` command. Proper file attribute settings protect the file/directory from both deletion and modification.

2) Resource limited execution: A new program called `timelimit` can limit the time for which a program executes. The time limit has to be specified in seconds. However, `softlimit` may be used to limit both execution time and memory usage. `Softlimit` can also be used to limit several resources, such as data and stack segment sizes, number of processes per uid and also the execution time. The last limit is not enforced, only at `SIGXCPU` signal is received. A signal handler needs to be installed in the test program to terminate the program on receiving this signal. In this case it would be desirable to test that the submitted code does not install a different signal handler. This can be tested using the `nm` command, which lists symbols from object files. This can be checked during the make phase. Submitted code containing disallowed symbols gets rejected.

5.8 Test scheduling

Program submissions are scheduled in the sequence they are received. Only one queue is maintained (assuming a uni-processor system). Re-submissions cause a queued job to be removed and a new job to be queued. The results of the automatic evaluation are stored in an html page linked to the page for the student's submission.

6. Conclusions and Further Work

Our system has been developed to explore the potential for more flexible program evaluation mechanisms (within a web-based course management system), which are

relatively simple to use. This has required consideration of programmable interfaces and the use of assessment specifications in XML to simplify the burden for designers of assessments. Our system focuses on evaluation of both behaviour and performance of parts of programs for greater flexibility of use, and we have argued that performance can be an indirect measure of design in some cases. We have illustrated our approach with an extended example, but a full evaluation of the benefits of our approach will require further research with more extensive use as well as studying the experiences of assessment designers using the system.

We have limited ourselves to work with a single programming language (C) at this stage in order to focus on component evaluation. It will be necessary to explore generalisations to include other languages so that we can better understand what additional language specific support is required and what can be made generic more easily.

The system has necessarily had to address security issues and uses a 'sand box' to protect against rogue programs. For more extensive testing of performance and behaviour in the context of specific environments (e.g. graphical user interfaces) this approach should be easy to generalise.

References

R. S. Baker, M. Boilen, M. T. Goodrich, R Tamassia, and B. A. Stibel (1999) "Tester and Visualizers for teaching Data structures", in proceedings of ACM 30th SIGCSE Tech. Symposium on Computer Science Education, pp 261-265, New Orleans, LA, USA.

S. D. Benford, K. E. Burke and E. Foxley (1993) "A system to teach programming in a quality controlled environment". The Software Quality Journal, pp 177–197.

M. Blumenstein, S. Green, A. Nguyen and V. Muthukkumarasamy (2004) "An experimental analysis of GAME: a generic automated marking environment", Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education, Leeds, United Kingdom, pp 67-71

D. Jackson and M. Usher (1997) "Grading student programming using ASSYST", Proceedings of 28th ACM SIGCSE Tech. Symposium on Computer Science Education, San Jose, California, USA, pp 335-339.

D. W. Juedes (2003) "Experiences in Web-Based Grading", 33rd ASEE/IEEE Frontiers in Education Conference November 5–8, 2003, Boulder, CO

M. Luck and M. Joy (1999) "A secure on-line submission system". In Software - Practice and Experience", 29(8), pp721--740

C. Mandal, V. L. Sinha, C. M. P. Reade (2004) "A Web-Based Course Management Tool and Web Services", in Electronic Journal of E-Learning, Vol 2(1) paper no. 19

Y. Pisan, D. Richards, A. Sloane, H. Koncek and S. Mitchell (2003) "Submit! A Web-Based System for Automatic Program Critiquing". In Proceedings of the Fifth Australasian Computing Education Conference (ACE 2003), Adelaide, Australia, Australian Computer Society, pp. 59-68

K. A. Reek (1989) "The TRY system or how to avoid testing student programs", Proceedings of SIGCSE, pp 112-116.

R. Saikkonen, L. Malmi, and A. Korhonen (2001) "Fully automatic assessment of Programming exercises", Proceedings of the 6th Annual Conference o Innovation and Technology in Computer Science Education (ITiCSE'01), Canterbury, United Kingdom, pp. 133-136