A SYSTEM FOR AUTOMATIC EVALUATION OF PROGRAMS FOR CORRECTNESS AND PERFORMANCE

Amit Kumar Mandal, Chittaranjan Mandal
School of Information Technology
IIT Kharagpur, WB 721302, India

Chris Reade
Kingston Business School
Kingston University

Keywords: Automatic Evaluation, XML Schema, Program Testing, Course Management System.

Abstract: This paper describes a model and implementation of a system for automatically testing, evaluating, grading and providing critical feedback for the submitted programming assignments. Complete automation of the evaluation process, with proper attention towards monitoring student's progress and performing a structured level analysis is addressed. The tool provides on-line support to both the evaluators and students with the level of granularity, flexibility and consistency that is difficult or impossible to achieve manually.

1 INTRODUCTION

The paper presents both the internal working and external interface of an automated tool that assists the evaluators/tutors by automatically evaluating, marking and providing critical feedback for the programming assignments submitted by the students. The tool helps the evaluator by reducing the manual work by a considerable amount. As a result the evaluator saves time that can be applied to other productive work, such as setting up intelligent and useful assignments for the students, spending more time with the students to clear misunderstandings and other problems.

The problem of automatic and semi-automatic evaluation has been highlighted several times in the past and a considerable amount of innovative work has been suggested to overcome the problem. In this paper we will be discussing the key approaches in the literature. Although our approach is currently limited to evaluating only C programs, the design and implementation of the system has been done in such a way that it takes over almost all of the burden from the shoulders of the students and the tutors. The tool can be controlled and used through the Internet. The evaluator and students can be in any part of the world and they can freely communicate through the system. Hence, the system helps in bringing alive the scenario of distant learning which is a crucial improvement for any large educational institute or university because they might have extension centers anywhere across the world. The same tutors can provide services to all the students in any such extension center.

2 MOTIVATION

The scenario that motivated us to build such a system was the huge cohorts of students in almost all big educational institutions or universities across the world. In almost all the big engineering institutes or universities the intake of undergraduates is around 600 - 800 students. As a part of their curriculum, at the place of development of this tool, the students need to attend labs and courses and in every lab each student has to submit about 9-12 assignments and take up to three lab tests. That amounts to nearly 10000 submissions per semester. Even if the load is distributed among 20 evaluators, each evaluator is required to test almost 500 assignments. Without automation, the evaluators would be busy most of time in testing and grading work at the expense of time spent with the students and also on setting up useful assignments for them.

3 RELATED WORK

A variety of noteworthy systems have been developed to address the problem of automatic and semi-automatic evaluation of programming assignments. Some of the early systems include TRY (Reek, 1989) and ASSYST (Jackson and M., 1997). Schemer-

Kumar Mandal A., Mandal C. and Reade C. (2006).
A SYSTEM FOR AUTOMATIC EVALUATION OF PROGRAMS FOR CORRECTNESS AND PERFORMANCE.
In *Proceedings of WEBIST 2006 - Second International Conference on Web Information Systems and Technologies - Society, e-Business and e-Government / e-Learning*, pages 196-203
DOI: 10.5220/0001251601960203

obo (Saikkonen et al., 2001) is an automatic assessment system for programming exercises written in the Scheme programming language, and takes as input a solution to an exercise and checks for correctness of function by comparing return values against the model solution. The automatic evaluation systems that are being developed nowadays have a web interface, so that the system can be accessed universally through any platform. Such systems include GAME (Blumenstein et al., 2004), SUBMIT (Pisan et al., 2003) and (Juedes, 2003). Other systems such as (Baker et al., 1999), Pisan et al(2003) and Juedes(2003) developed a mechanism for providing a detailed and rapid feedback to the student. Systems like (Luck and Joy, 1999), (Benford et al., 1993) are integrated with a course maker system in order to manage the files and records in a better way.

Most of the early systems were dedicated to solve a particular problem. For example, some systems are concerned with the grading issue thus turning their attention from quality testing. Others are concerned with providing formative feedback and loosing their attention towards grading. We are concerned with addressing all aspects of evaluation whether it is testing, grading or feedback. The whole design and implementation of our system was focused toward bringing comfort to the evaluators as well as students, without being partial towards the quality of testing and grading being done. Our aim was to test the programs from all possible dimensions i.e. testing on random inputs, testing on user defined inputs, testing on execution time as well as space complexity, performing a perfect style assessment of the programs. Secondly, security was given much attention because, as the system was to be used by fresh under graduate students who do not have many programming skills, they might unknowingly cause harm to the system. Our third focus of attention was grading, which should not impose crude classification such as zero or full marks. Efforts have been made to make the grading process simulate the manual grading process as much as possible. Fourthly, the comments should not be confined to just a general list of errors that any student is likely to commit. All the comments should be specific to a particular assignment. As has been already mentioned, the system currently supports only the C language, so during implementation, the fifth and foremost focus was to make the model and design as general as possible so that the system could be easily upgraded to support other popular languages like C++, Java etc.

4 SYSTEM OVERVIEW

4.1 Testing Approach

Black Box testing, Grey Box testing, and White Box testing are the choices available in literature to test a particular program. In Black Box testing, a particular submission is treated as a single entity and the overall output of the programs are tested. In Grey Box testing, component/function final output is tested. White box testing allows structure, programming logic as well as behavior to be evaluated. In an environment where the students are learning to program, testing only the conformance of the final output is not workable because conformance to an overall input/output requirement is particularly hard to achieve. Evaluations that relied on this would exclude constructive evaluation for a majority of students. Grey Box approaches involve exercising individual functions within the student's programs and are, unfortunately, language sensitive. Looking at the pitfalls of the above two strategies our decision was to choose the White Box approach, which is more general than the Black Box approach. White Box testing involves exercising the intermediate results generated by the functions, therefore, whether the program has been written in a particular way, following a particular algorithm and using certain data structures can be ascertained with greater probability.

4.2 High Level System Architecture

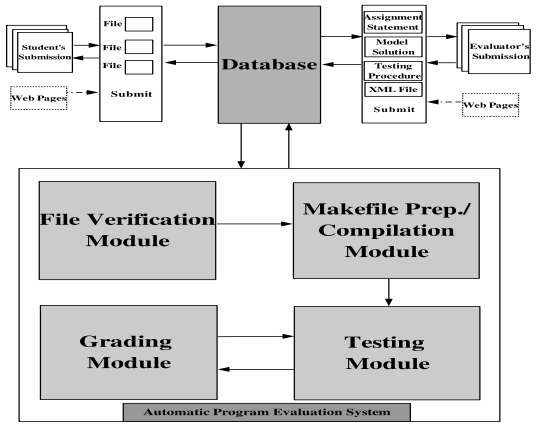

Figure 1: High Level System Architecture.

Fig.1 shows the High Level System Architecture diagram, the whole process of automatic evaluation begins with the evaluator's submission of the assignment descriptor, model solution of the problem stated

in the assignment descriptor, testing procedure and an XML file. After the evaluator has finished the submission for a particular assignment, the system checks the validity of the XML file against an XML Schema. If the result of XML validation is negative, the system halts any further processing of the file and the errors are returned to the evaluator. The evaluator has to correct the errors and then resubmit the files. If the validation of the XML file against the XML Schema is positive, then the assignment description becomes available for students to download or view and subsequently submit their solutions.

These submissions are stored in a database. The system accepts the submissions only if they are submitted within the valid dates specified by the evaluator. The next criteria that should be passed for completion of submission is that proper file names are used, as specified by the evaluator in the assignment description. The above operations are performed by the File Verification module. After its operation is complete the next module which comes into the picture is the Makefile Prep/Compilation module with the task of preparing a makefile and then using this makefile to compile the code. This module generates feedback based on the result of the compilation. I.e. if there are any compilation errors, then the system rejects the submission and returns the compilation errors to the student. Thus the submission is complete only if the student is able to pass the operations of both file verification module and the compilation module. After the submissions are complete the next module executed is the testing module which includes both dynamic testing as well as static testing. The dynamic testing module accepts students' submissions as input, generates random numbers using the random number generator module [1](not shown in figure). It then uses the testing procedures provided by the evaluator to test a program on the randomly generated inputs. Sometimes it is not enough to test the program on randomly generated inputs (e.g. where the test fails for some boundary conditions). By using the randomly generated inputs we cannot be sure that the program undergoes specific boundary tests on which the test may fail.

To overcome this drawback, facilities have been provided by which the evaluator can specify his/her own test cases. The Evaluators have the freedom to specify the user defined inputs either directly in the XML file or he may also specify the user defined inputs in a text file (if the number of inputs is significant). This is an optional facility and the evaluator may or may not be using it based on the requirements of the assignment. If the results of testing the program on random input as well as user defined inputs are positive, then the next phase of testing that begins is the correctness of the program on execution time as well as space complexity. During this testing, the execution time and the space usage of the user program are compared against the execution time and the space usage of the model solutions submitted by the evaluator. During this testing, the execution time of the student program as well as the evaluator's program are determined for a large number of inputs. The execution time of the student's program and the evaluator's program are compared and analyzed to determine whether the execution time is acceptable or not. If the test for space and time complexity goes well then the next phase is the Static Analysis or the Style analysis. Static testing involves measuring some of the characteristics of the program such as, average number of characters per line, percentage of blank lines, percentage of comments included in the program, total program length, total number of conditional statements, total number of goto, continue and break statements, total number of looping statements, etc. All these characteristics are measured and compared with the model values specified by the evaluator in the XML file. Testing and Grading modules works hand in hand as testing and grading are done simultaneously. For example, after testing a program on random inputs as well as user defined inputs is over, the grading for that part is done at that moment, without waiting until all testing is complete. If at any point of time during testing, the programs are rejected or aborted, then the grading that has been done up to that point is discarded (but feedback is not discarded).

Security is a non-trivial concern in our system because automatic evaluation, almost invariably, involves executing potentially unsafe programs. This system is designed under the assumption that programs may be unsafe and executes programs within a 'sand-box' using regular system restrictions.

5 AN EXAMPLE

Let us start with an example that is very common for a data structures course, for example: the Mergesort Program. As the regular structure of a C program consists of a main function and a number of other functions performing different activities, we have decided to break down the Mergesort Program into two functions (1) Mergesort function - This function performs the sorting by recursively calling itself and the merge function. (2) Merge function - This function will accept two sorted arrays as input and then merge them into a third array which is also sorted. As a White Box testing strategy is followed, testing and awarding marks on correct output generated by the program will be of no use because this will not distinguish be-

[1]This module generate the random numbers required for testing the programs

tween different methods of sorting (Quick sort, Selection sort, Heap sort, Insertion sort etc.).

5.1 Our Approach

Our approach is to test the intermediate results that the function is generating, instead of a function final output or the program final output. This section explains one approach by which the mergesort program can be tested. Our aim is to test that the student has written the mergesort and the merge function correctly. Our strategy is to check each step in the mergesort program. Fig.2 explains the general working of the mergesort program. Our approach is to catch data at each step

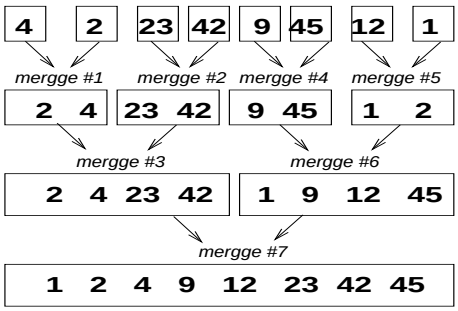

Figure 2: Working Principle of Mergesort program.

(i.e. merge#1 to merge#7 of the example). This can be done by using an extra two dimensional array of size (n) * (n-1). The idea is to transfer the array contents to this two dimensional array at the end of the merge function.

	0	1	2	3	4	5	6	7
0	2	4	23	42	9	45	12	1
1	2	4	23	42	9	45	12	1
2	2	4	23	42	9	45	12	1
3	2	4	23	42	9	45	12	1
4	2	4	23	42	9	45	1	12
5	2	4	23	42	1	9	12	45
6	1	2	4	9	12	23	42	45

Figure 3: Two Dimensional Array.

After the execution of the student‘s program, a two dimensional array similar to Fig.3 will be created, this array can be compared with the two dimensional array that is formed by the model solution. If both the arrays are exactly the same then the student is awarded marks, but if the arrays are not similar, then the action initiated depends upon the action specified by the evaluator in the XML file. As has been mentioned previously, the array contents need to be transfered to the two dimensional array at the end of the merge function. This operation should be performed by the function as shown in Fig.4.

```
transfer(int *a, int size, int *b)
{int i;
 for (i=0;i<size;i++)
  *(b + depth*size +i) = *(a+i);
 depth = depth + 1;
}
```

Figure 4: Auxiliary Transfer Function.

5.2 Assignment Descriptor for the Mergesort Program

The assignment descriptor is one of the crucial submissions necessary for the successful completion of the whole auto-evaluation process. If the assignment descriptor is not setup up in a correct manner, it would be difficult for the student to understand the problem statement properly and most students will end up with a faulty submission. We have considered some of the general-purpose properties that an assignment descriptor should have: (1)Simple and easy to understand language should be used, so that it becomes easy for the student to understand the problem statement; (2)Prototypes of the function to be submitted by the student must be specified clearly in the descriptor; (3) The necessary makefile and the auxiliary functions (for example, the transfer function shown in Fig.4) should be supplied as a part of the assignment descriptor to make it easier for the student to reach the submission stage.

5.3 Evaluator’s Interaction

The Automatic Evaluation process cannot be accomplished without the valuable support of the evaluator. The evaluator needs to communicate a large number of inputs to the system. Since the number of inputs are large, providing a web interface is not a good idea to accept the values because it is cumbersome both for the evaluator to enter the values and for the system to accept and manage the data properly. Our idea is that, the evaluator will provide the inputs in a single XML file. In this XML file the evaluator is required to use predefined XML tags and attributes to specify the inputs. The XML is used to specify the following information: (1)Name of the files to be submitted by the students. (2)How to generate the test cases. (3)How to generate the makefile. (4)Marks distribution for each function to be tested (5)Necessary inputs to carry out static analysis of the program

Along with the XML file, the evaluator is also required to submit the model solution for the problem, Testing Procedure and the Assignment descriptor. The XML file is crucial because it controls the working of the system and if the XML file is wrong then the whole automatic evaluation process becomes unstable. Therefore it is necessary to ensure that the XML file is correct which can be done, to some extent, by validating the XML file against the XML Schema. [2]. Fig.5 shows only a part of the main XML file that is submitted. This portion of the XML is used by the module that checks for proper file submissions. Fig.6 shows the part of the XML schema that is used to validate the part of the XML file as shown in Fig.5.

```
<source_files>
<file name="main_mergesort.c">
<text>File containing the main
                function</text>
</file><file name="merge.c">
<text>File containing the merge-
                 function</text>
</file><file name="mergesort.c">
<text>File containing the merge-
            sort function</text>
</file></source_files>
```

Figure 5: XML Spec. for checking file submission.

```
<xs:element name = "source_files">
 <xs:complexType><xs:sequence>
 <xs:element name="file" maxOccurs-
                    ="unbounded">
 <xs:complexType> <xs:sequence>
 <xs:element name="text" minOccurs
             ="0"> <xs:simpleType>
 <xs:restriction base="xs:string">
 <xs:minLength value="0"/>
 <xs:maxLength value="75"/>
 </xs:restriction></xs:simpleType>
 </xs:element></xs:sequence>
 <xs:attribute name="name" type=
                   "xs:string">
 </xs:attribute></xs:complexType>
 </xs:element>  </xs:sequence>
 </xs:complexType> </xs:element>
```

Figure 6: XML Schema validating XML in Fig.5.

[2]The purpose of the XML Schema is to define legal building blocks of an XML document. An XML Schema can be used to define the elements that can appear in a document, define attributes that can appear in a document, define data types of elements and attributes etc

5.4 Input Generation

The Automatic Program Evaluation System is a sophisticated tool, which evaluates the program using the following criteria:

- Correctness on dynamically generated random numbers
- Correctness on user defined inputs
- Correctness on time as well as space complexity
- Performs style assessment of the Programs
- Number of Looping statements
- Number of Conditional statement.

Initially the programs are tested on randomly generated inputs. For effective testing we cannot rely on fixed data because they are vulnerable to replay attacks. Evaluators have the option to write down their own routines in the testing procedures to generate inputs or, alternatively, the system provides some assistance in the generation of inputs. Currently the options are supported are as follows:

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

The evaluator can express his/her choice of the random numbers on which he/she wants the programming assignments to be tested in the XML file. Fig.7 shows the example of the XML statement that is used to generate input for the mergesort program.

```
<testing>
<generation iterations="100">
<input inputvar="a" vartype="array"
type="float" range="positive"min= "10"
max="5000"sequence="ascend">
<arraysize>50</arraysize></input>
</generation>
<user_specified source="included">
<input values="6,45,67,32,69,2,4">
</input>
<input values="5,89,39,95,79,7">
</input></user_specified></testing>
```

Figure 7: XML Specification for Input Generation.

The evaluator/tutor should specify the inputs in the *testing* element, the *generation* element is used to specify random inputs to be generated. This element has only one attribute named *iterations* and the *iterations* attribute can be used to specify the number of times the evaluator needs to test a program on random inputs. Any number of *input* elements can be placed between its starting and closing *generation* tag. The *input* element offers a lot of attributes to specify the type of random number to be generated. The *vartype* attribute of an *input* element is used to specify whether the randomly generated value should be an *array* or *single* value. The *min* and *max* attributes can be used to generate the inputs within a range. If the attribute is not mentioned explicitly in the XML then the system sets these attributes to the default values. The *type* attribute can be set to one of the three values *integer*, *float* or *string*. Another important attribute of *input* elements is the *range* attribute. This attribute is used to specify whether *positive*, *negative* or *mixed* (mixture of positive and negative) values is to be generated. The *inputvar* is the attribute that is used to name the variable to be generated. For example in Fig.7, the array which is generated randomly will be stored in *a*. Sometimes the evaluator may choose to generate the values in some order (ascending or descending) and this problem is resolved by the *sequence* attribute which can be set to any of the two values (*ascend* or *descend*). Every element that is specified between the tags of user_specified is used to supply user defined inputs to the program. The *user_specified* elements have only one attribute named *source* which can take one of the two values (*included* or *file*). If the attribute is set to *included* then the system looks for the user defined inputs in the XML itself, on the other hand if the attribute is set to *file*, then the system looks for the user defined inputs in a text file. Similar to the *generation* element, the *user_specified* element can have any number of *input* elements, but here the *input* element can have only one attribute named *values*. All the user defined inputs should be specified following the same sequence as followed for the generation of random inputs.

In Fig.7 an array(vartype="array") of size 50(<arraysize>50</arraysize>), containing ascending(sequence="ascend") positive(range="positive") float(type="float") values in the range of 10(min="10") and 5000(max="5000") is generated and supplied as input to the mergesort program. The above procedure is iterated 100(iterations="100") times. Fig.7 also shows that, two user defined inputs have been provided by the evaluator, these user defined inputs are supplied *as is* to the testing procedures.

5.5 Grading the Programs

The grading process is made as flexible and granular as possible. During testing of a program on user defined inputs, suppose the program is unsuccessful in satisfying the output of the model solution. At this moment the XML is parsed to determine the choice mentioned by the evaluator. The evaluator may have chosen to abort the test and award zero marks to the student or, alternatively, the evaluator/tutor may have chosen to move forward and test the program on other criteria. If the evaluator had decided to test the program on other criteria, then the tutor has the flexibility to either award full marks, zero marks or a fraction of the full marks meant for testing on random inputs and user defined inputs. Fig.8 shows the example XML for grading the Mergesort program.

```
<status marks="50" abort_on_fail="true">
<item value="0" factor="1"
fail="false"><text>Fine</text></item>
<item value="1"factor="0.6"
fail="false"><text>Improper</text>
</item> <item value="2" factor="0"
fail="true"><text>Wrong</text></item>
</status>
```

Figure 8: XML Specification for Grading.

Based on the result of the execution, some value is awarded to the program. For example, if the result of execution of the mergesort program is correct for both the user defined inputs as well as random inputs then the value awarded to the program is 0. If the result is incorrect for user defined inputs or random inputs, then the value awarded is 1. If the result of execution is incorrect for both the tests, then the value awarded is 2. Based on the value awarded, marking is done, i.e. if the value is 0 full marks are awarded for the test (factor = "1"), if the value is 1, then only 60% marks are awarded (factor="0.6") and if the value is 2 then no marks are awarded, the program is given *fail* status(fail="true") and the program execution is aborted(abort_on_fail="true").

5.6 Commenting on Programs

The system generates feedback comments, which are specific to a particular assignment. During testing the programs on random and user-defined inputs, if the test fails for some particular random input, then the comment would contain the specific information about the random input on which the test has failed. The student can run his/her program on the random input and find out what is wrong in the program. If

the test fails for user defined inputs, then the system never returns the user-defined inputs because that may cause serious security flaws in the system. Instead, the system returns an error message mentioned by the tutor in the XML for that particular input.

5.7 Testing the Programs on Time and Space Complexity

Testing programs on their execution times and space complexity is an important idea because different algorithm often have different requirements of space and time during execution. For example, bubble sort is an in-place algorithm but inefficient, the mergesort algorithm is efficient if extra space is used, the quick sort algorithm is an in-place algorithm and usually very efficient, the heap sort algorithm is an in-place algorithm and efficient. This section shows the analysis between two mergesort programs. One is submitted by the student and the other is submitted by the evaluator. These programs are different in the number of variables declared and the dynamic memory allocated during the execution of the programs. Fig.9 shows the execution time of the two mergesort programs.

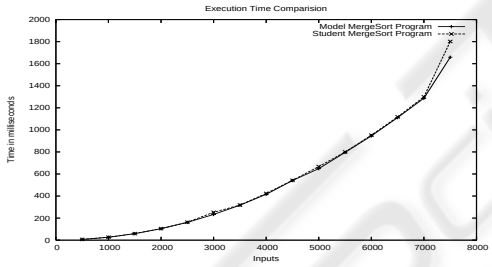

Figure 9: Execution times of Mergesort program.

As the execution time of the programs are nearly same, this suggests that we need to test the programs further. For a particular input the system executes the model program N times and determines the acceptable range of execution time for the student's program. For the same input the system then executes the student's program, determines the execution time and then checks whether it falls within the range determined after the execution of the evaluator's mergesort program. The above procedure will be iterated for a large number of inputs. After getting results for all the inputs, the system decides whether the program is acceptable or not. The same procedure is followed to test the programs on space usage. Fig.10 shows the space complexity of the above two programs i.e. the model mergesort program and the student's mergesort program. Fig.10 also shows the space requirement of another mergesort program where the student has not used the memory properly.

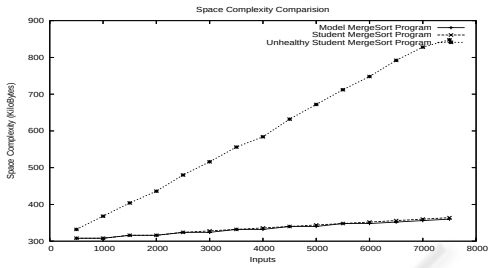

Figure 10: Space Complexity of MergeSort programs.

Here the space required shows the virtual memory resident set size which is equal to the stack size plus data size plus the size of the binary file being executed. It is evident from Fig.10 that if the student is using the memory properly, then the amount of memory required for the execution of the student's program will not vary drastically from the memory requirement of the Model Solution. When the student is not using the memory properly, for example when the student is not freeing the dynamically allocated memory, then the memory requirements of the program will increase drastically as compared to that of the model solution. These problematic programs get easily caught with the help of the same procedure as explained for testing the programs on execution times.

6 SECURITY

From the outset of the implementation we have been concerned with making the system as secure as possible because we cannot rule out the possibility of a malicious program, that may intentionally or unintentionally cause damage to the system. Therefore, the system has been implemented under the assumption that the programs may be unsafe. The first aspect that has been taken into consideration is that, programs may intentionally or unintentionally have calls to delete arbitrary files. To override the above possibility, a separate login named as 'test' login has been created. During testing, the system uses Secure Shell (ssh) with RSA encryption-based authentication to execute necessary commands in this login. Necessary files are transferred to this login ('test' login) securely using the Secure Copy Protocol. As the test login do not have access to other directories, the programs executing in this login will not be able to delete them. To save the files that are present at the test login we have used the change file attributes on a Linux second extended file system. Only superusers have access to these attributes, and no user can modify them. After

the file attributes are changed they cannot be deleted or renamed, no link can be created to this file and no data can be written to the file. Only a superuser or a process possessing the CAP_LINUX_IMMUTABLE capability can set or clear this attribute.

The next aspect under consideration is the problem of the presence of infinite loops in the student's program. In order to solve the problem of infinite loops, the system limits the resources available to the programs. Both the soft limit and the hard limit have to be specified in seconds. The soft limit is the value that the kernel enforces for the corresponding resource. The hard limit acts as a ceiling for the soft limit. After the soft limit is over the system sends a SIGXCPU signal to the process, if the process does not stop executing then the system sends SIGXCPU signal every second till the hard limit is reached, after the hard limit is reached the system sends a SIGKILL signal to the process. The *setrlimit* command has been used to limit both execution time and memory usage, same command can also be used to limit several other resources, such as data and stack segment sizes, number of processes per uid and also the execution time. Care has been taken so that the student is not able to change the resources available to his/her program.

7 CONCLUSION

Our system has the potential to open up new horizons in the field of Automatic Evaluation of programming assignments by making the mechanism relatively simple to use. We have illustrated the flexibility of our system and explained the details of the automatic evaluation process with an example in a systematic way, ordering the phases serially as they occur in the practical environment. This covers: deciding the testing approach, designing the assignment descriptor, the teacher's interaction with the system, and testing programs. The tool pin-points each possible dimension for testing the programs i.e. testing the programs on random inputs, testing the programs on user defined inputs, testing the programs on time complexity, space usage and style assessment of the student's program. Students benefit because they know the status of their program before final submission is done. Evaluators benefit because they do not have to spending hours for grading the programs. But achieving peak evaluation benefits of our approach, requires further research and with more extensive use as well as studying the experience of the evaluators and students using the system. Currently we have built the system to work with a single programming language i.e. the 'C' language. This is done in the first instance to gain experience with the interactive nature of the system in a simple form. Implementation doors have been kept open, so that we can extend the system to test programming assignments in other popular languages such as C++, Java etc by making small changes in the code.

REFERENCES

Baker, R. S., Boilen, M., Goodrich, M. T., Tamassia, R., and Stibel, B. A. (1999). Tester and visualizers for teaching data structures. In *Proceedings of the ACM 30th SIGCSE Tech. Symposium on Computer Science Education*, pages 261–265.

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

Blumenstein, M., Green, S., Nguyen, A., and V., M. (2004). An experimental analysis of game: A generic automated marking environment. In *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*, pages 67–71.

Jackson, D. and M., U. (1997). Grading student programming using assyst. In *Proceedings of 28th ACM SIGCSE Tech. Symposium on Computer Science Education*, pages 335–339.

Juedes, D. W. (2003). Experiences in web based grading. In *33rd ASEE/IEEE Frontiers in Education Conference*.

Luck, M. and Joy, M. (1999). A secure online submission system. *In Software-Practice and Experience*, 29(8):721–740.

Pisan, Y., Richards, D., Sloane, A., Koncek, H., and Mitchell, S. (2003). Submit! a web-based system for automatic program critiquing. In *Proceedings of the fifth Australasian Computing Education Conference (ACE 2003)*, pages 59–68.

Reek, K. A. (1989). The try system or how to avoid testing students programs. In *Proceedings of SIGCSE*, pages 112–116.

Saikkonen, R., Malmi, L., and Korhonen, A. (2001). Fully automatic assessment of programming exercises. In *Proceedings of the 6th annual conference on Innovation and Technology in Computer Science Education (ITiCSE)*, pages 133–136.