Automatic Grading of Programming Exercises

José Paulo Leal   Nelma Moreira

Technical Report Series: DCC-98-4

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150 Porto, Portugal

Tel: +351+2+6078830 – Fax: +351+2+6003654

http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

Automatic Grading of Programming Exercises

José Paulo Leal   Nelma Moreira

{zp,nam}@ncc.up.pt
DCC - FC & LIACC, Universidade do Porto
R. do Campo Alegre, 823 4150 Porto – Portugal

July 1998

1 Introduction

During the 1997-1998 school year, answering programming exercises in the computer was introduced in the curricula of several Computer Science undergraduate courses in our department. The courses in question involved such distinct programming languages as the assembly language for the 68000 Motorola microprocessor, Prolog programming, and Graphical Users Interfaces (GUI) construction using tcl-tk [Ous94].

For this purpose a grading tool was developed whose goal was to evaluate the students programming skills better than just with written exercises. The motivation is that outside academia, program development is mostly seen as a cycle that includes program design, coding and debugging, possibly leading to recoding (or even redesign) until a program meeting the requirements is obtained. Traditional grading methods test only the design and coding skills. We wanted to test whether the students had acquired the skills required to complete the full programming cycle.

A grading environment allows the creation of more evaluation moments since it simplifies the task of grading the exercises. Moreover, by centralizing the evaluation in a single environment it is possible to monitor the learning progress, hence allowing the teacher (and the student) to detect learning difficulties earlier.

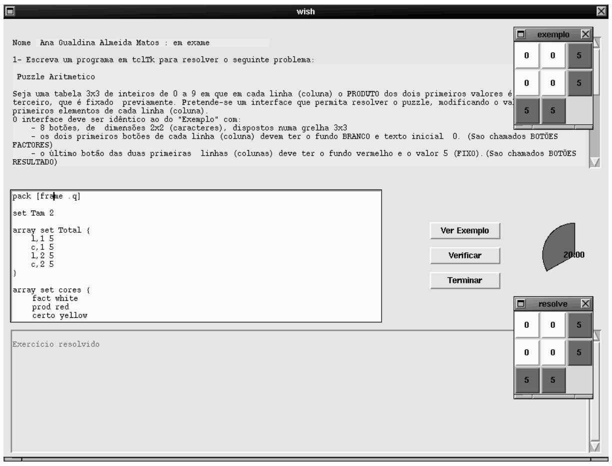

Figure 1: Program grading environment

The following section relates this work with other similar grading programs. Section 3 presents a brief description of the grading environment and section 4 evaluates the results achieved with this system. The advantages and drawbacks of this system, as felt by the teachers and students involved, are summarized discussed as basis for future work.

2 Grading Programs

There is a long history of research on programs that can grade. At least since 1961 such programs have been used, namely in the introductory programming and numerical analysis courses at Stanford University. The system described by George Forsythe and Niklaus Wirth in 1965 [FW65] already follows the fundamental structure of most modern systems: the procedures submitted by students were validated using a set of test data. Hence, it was not proved program correction, but the absence of known errors.

In these initial systems the procedure to be verified and the main program responsible for the grading, both written in BALGOL (a dialect of ALGOL 58), were compiled together in a new program. In general, the student's procedure used IO procedures supplied by the system in order to minimize interface problems.

With the evolution of computers, grading systems increased in complexity, giving more importance to communication with the operating system, diversifying the tests made to the subject programs, and introducing tools for monitoring the grading process. Systems available today, such as Ceilidh [ZF91], Kassandra [vM94], or CoBalT [JL95], take advantage from local networks and provide remote submission, although with some limitations regarding the supported operating systems and hardware. Meanwhile, several systems have been developed for grading other kinds of examinations using a computer, in particular multiple choice examinations [RH83].

Existing grading systems approach the problem by checking the student's program against a correct program relatively to a set of characteristics. This verification usually applies two different techniques of analysis: static analysis on the program's source code, and dynamic analysis on the object code. The final grade is a function of the results obtained by the two types of tests.

Static analysis uses software metrics to compare programs in their readability, programming style and maintainability, for instance. There are several methods for checking code complexity [Ver85, McC76], where the goal is to measure the program's quality. These metrics are based on statics of the syntactic elements of the source program.

Dynamic analysis is based on executing the program by an oracle that generates test data and verifies the results. Some systems check the program's efficiency by using an execution profiler. The generation of tests is a subject addressed by other research fields, such as program verification, and several algorithms have been proposed to determine sensitive data to pinpoint program faults[JF].

3 Design and implementation

Our goal with this grading system was to create a controlled environment allowing the evaluation of the student's programming skills. It was not intended as a learning environment to use during practical classes or for individual study.

The system is based on the notion of a **solving environment**, whose interface is presented in Figure 1, in which the students solves individually the examination. The system is complemented with a **monitoring environment**, whose interface is presented in Figure 3. The monitoring environment is used by the course teacher to oversee the examination and includes several tools for such tasks as listing the situation of each student, visualizing and printing examination sheets, visualizing student's solutions, or supporting manual correction of a student's grade.

Although it lacks the ambition of a true learning environment, this environment allows more than just writing the program and submitting it for automatic grading. Indeed, the student can also verify the program, and he or she can ask for a report with the errors detected so far, that he may then use to correct the program. The process ends when the student concludes the exercise without errors, or when the process runs out of time. In the end, the systems prints out an examination sheet similar to the one shown in Figure 2. This sheet contains the problem, the answer and the grade, and it will be signed by the student and stored for administrative purposes.

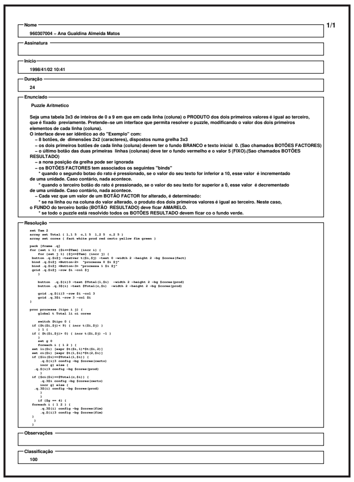

Figure 2: Example of an examination sheet

The main design goals for this grading environment were simplicity and robustness. The system is to be used by students, that frequently use it just a few times. Possibilities of interaction are reduced to facilitate immediate understanding of the GUI. The simplicity of the GUI itself increases the system's robustness, but nevertheless the system logs all modifications made by the students, hence allowing for recovering from error situations. As the design evolved, it became clear the advantages of organizing the system as two modules:

The solving module provides a GUI where the student can read the problem and write the answer. This module is responsible for controlling execution of the examination, namely the student's authentication, the examination's duration, recording all the examination data (problem, successive programs, error reports, grades, etc), and printing the examination sheet. This module is domain independent.

The grading module allows the execution of programs in a safe environment, compares the student's solution with a correct solution, produces an error report and gives a grade. This module depends on the domain of the exercise.

The grading module is thus the component of the system that is adapted to the domain characteristics. Hence, several grading modules were developed instead of a single generic module. We have found out that even for a given domain a substantial part of this module has to be adapted for each exercise.

The grading modules were implemented emphasizing execution security. Programs written by the students have denied access to the operating system, and the system detects most situations that could lead to non termination.

Exam verification in the grading modules has so far followed the standard approach in these kind of systems. Each exercise uses an **oracle**[1] – a component that generates test data and checks the results. The oracle is responsible for grading and producing an error report.

4 Evaluation

To evaluate the system, we asked the students of the Graphical Interfaces course to answer a small questionnaire about their preferences regarding the use of computers in practical examinations, automatic grading, and extension of this method to other courses. The students had performed 3 tests with this system. The questionnaire was anonymous, and it was handed out after the last

[1] In the literature, the term "oracle" is commonly used for the result checker

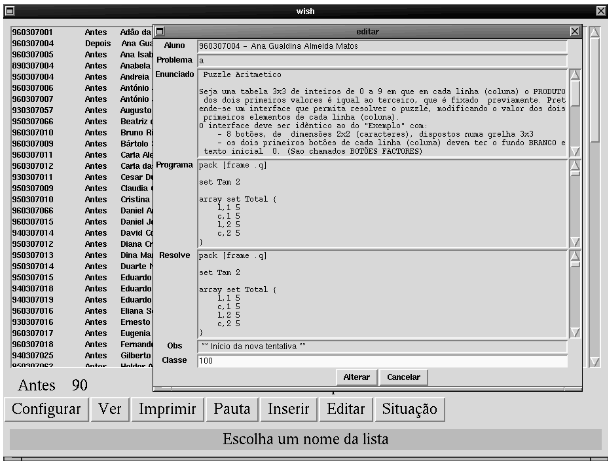

Figure 3: Examination monitor

examination and before publishing the final grades. The questionnaire was filled in by 47 students, corresponding to 52% of the 90 students that were actually tested.

I prefer examinations in the computer	65.96%
I prefer examinations on paper	12.77%
I am indifferent to how examinations are made	17.02%
Did not answer	4.26%
I prefer examinations to be graded automatically	40.43%
I prefer examinations to graded by the teacher	48.94%
I am indifferent to the grading method	6.38%
Did not answer	4.26%
I would like this kind of examination to be used in other courses	55.32%
I would like this kind of examination to stop	8.51%
I believe that this kind of examination only makes sense in this course	25.53%
Did not answer	10.64%

Table 1: Results of the questionnaire

The results of the questionnaire are presented on Table 1. They show that the majority of students (66%) approved this examination method, which was only rejected by a fairly small (13%) number of students.

Automatic grading divided the students more. We found that 49% of them stated that they would rather have a person grading the examinations, although 40% stated that they enjoyed automatic grading. It should be noted that more than 10% of students did not answer or were indifferent to the grading method. In the written comments some students expressed concern about re-grading, as they would want to guarantee that teachers could re-grade exams. Actually, several exams were in fact checked and in a few cases re-graded due to faults in the automatic grader.

A few students made other pertinent comments, namely about the amount of lines in error reports, the lack of support material during the examination (on-line manuals, for instance), and specific deficiencies of the grading system under unusual circumstances.

The information gathered from the questionnaire and the suggestions made by teachers motivated a reflection about the characteristics required by a grading system. The several proposed improvements can be cataloged in tree major groups.

Programming environment The programming environment emphasized simplicity at the cost of versatility. The programming environment should include more tools to help in program editing and debugging.

Debugging help The system should motivate the students to look for explanation for the detected errors, avoiding the systematic use of trial and error. Error reports should be more to accurate, and should point to on-line support material that, in tutorial mode, could include the courseware itself.

Oracle development This task is time consuming and error prone, and has to be repeated for each exercise. Ideally, one would want for the oracle to be automatically produced from a solution program.

Creating a more sophisticated grading environment requires that the students have become familiar with the environment before the actual examination. On the other hand, including more help within the system also points towards a learning environment, instead of just a simple grading environment.

5 Future Work

Arguably, the most challenging task in program graging research is the automatic creation of grading oracles with emphasis on dynamic rather than static analysis.

The information given by static analysis on quality and maintainability of programs is appropriate to large programs, but not relevant to small programming exercises. Moreover, students feel that the results of static analysis do not contribute to locate or explain errors when they are still trying to understand the basic principles of a programming language.

Figure 4: The "black box" model of the traditional oracle

From the point of view of traditional dynamic analysis the program to analyze is seen as a **"black box"** to which data is given and from which results are extracted, following the model described in Figure 4. In general, the data is chosen in order to verify some conditions specified in the problem. The grade is computed by analyzing the different results.

An oracle has to be created for each exercise. In existing systems, the oracle programs are implemented manually for each case. Ideally, we would like for the oracles to be produced automatically from a solution program, in order to simplify the integration of new exercises in the courseware. Moreover, an ideal oracle would locate errors precisely and give tips for their solution. Having these goals in mind, the following possibilities should be investigated.

Oracle generation In the traditional approach, the fact that oracles are produced manually for each exercise increases the chance of errors or unpredicted situations. Although the creation of generic oracles seems infeasible, it is expected that oracles of same domain to share common functionalities. For that purpose we intend to study the results and techniques of Computer Science fields such as program verification, program synthesis from specifications and semantics of programming languages. As a result of this study it is expected the development a language for oracle specification.

Location errors In situations where the results obtained from tests are different from those expected, it is important to identify the source of error and not simply to detect its existence. It is not expected to find a general error location algorithm. Most likely, different algorithms may be found depending on the domain and even on the class of problem in a given domain. In any event, there are some generic strategies that can be applied, such as splitting the "black box" into components, or the detection of usual semantic errors.

Explaining errors The errors found by oracles should be integrated with the information contained in the rest of the courseware for the given domain. For instance, the errors in the use of library functions/commands should be linked to the on-line documentation about them. Another example is the syntactic error messages that should be linked to to the pages of the courseware where the language's syntax is described. These two types of errors are fairly simple to detect and document, nevertheless it is expected to extend this feature to explain other types of errors.

In the following subsections the different domains that will be included in future projects are analyzed. These domains where chosen for their diversity and for the possibilities they open for automatic grading.

5.1 Formal computational models

Formal models of computation are studied in an introductory course which also cover basic notions of discrete mathematics. They appeal and help to develop formal reasoning skills, which often results in students facing some difficulties. The possibility of simulating these models in a learning system is highly desirable.

The formal computational models can be divided in two groups: total and universal ones. Using the former only a restricted number of computable functions can be calculated, or certain classes of recursive languages can be recognized. In the later, the Church-Turing conjecture claims that any computable function can be calculated. Examples are Turing Machines or λ-calculus.

Total models, in particular the finite automata, have several practical applications due to their equivalence to regular expressions. The use of regular expressions is essential in many string manipulation languages, operative system shells and database query languages. On the other hand, the analysis of properties such as string recognition, automata equivalence, minimization, relationship between automata, regular languages, and so forth, can be efficiently implemented and, with an adequate GUI, used in the construction of exercises for the study of these subjects. This domain module will make use of a general system for graph manipulation that is presently under development at LIACC.

In what regards the universal models, in particular the Turing Machines, although they raise the same problems as most of the programming languages, its simplicity should also be explored, at least for a small set of introductory examples.

5.2 Databases

Databases is another field highly suitable for the development of several application domains. An introductory course in databases [UW97] usually focus on two fundamental issues that are relevant in the learning process of the students undertaking the course, these are: conceptual design (or data modeling) and data manipulation languages.

A major difficulty with the students in this course is data modeling using the standard entity-relationship model. A data model in this formalism can be seen as a colored graph, thus it is a good target for the same graphic and analytic tools for graph manipulation described in section 5.1.

There are several algebras and calculus used for querying databases and SQL is a standard language in this area. SQL has the advantage of beeing a declarative language, therefore shall simplify the creation of oracles for this sub-domain. In fact, the oracles of this domain are expected to actually prove the equivalence of two queries and not simply to validate as in general purpose programming languages. It would also be interesting to extend this domain to other relational calculus and algebras.

5.3 Logic programming

The logic programming paradigm is based on the use of **logic** to represent knowledge, controlled and manipulated by and an **inference** mechanism. Prolog [CM84] is a logic programming language that is taught in Artificial Intelligence courses.

A Prolog program is a set of formula from a subset of the first order logic for which there exists a complete inference system. Nevertheless, although testing the correction of a Prolog program could be made using formal logic, it leads to a semi-decidable problem. Moreover, the non equivalence between denotational and operational semantics raises additional difficulties.

The Prolog grading system implemented in Aleph uses a meta-interpreter that limits the use of non-logic predicates (such as IO predicates) and detects infinite loops, controlling the depth of the search tree. For each program (to which an oracle predicate is associated) the set of solutions is compared with the set of solutions of a correct program.

Several improvements can be made to this domain module, such as the inference of argument types and modes [FD92, Zob87]. Another improvement could be the identification of classes of problems with denotational semantics that support methods for evaluating program equivalence, using techniques similar to those used for proving integrity in deductive databases [Lev91].

5.4 Assembly programming

Assembly programming languages can be seen as a low-level imperative programming language. Undergraduate students are often exposed to these languages in the early years of their formation in Computer Science, usually in the context of a course on Computer Architecture. Standard tools are processor simulators and matching graphical tools. Example includes `SPIM` [Lar97], used to teach MIPS assembly language, and `m68k` [Fil97], used to teach MC68000 assembly.

One can easily adapt the simulator to check for program assertions and impose timing limits on programs. On the other hand, the very low level nature of these programs implies that in this case it is particularly hard to directly connect the original specification and the actual program being executed, even if the specification is relatively simple.

A typical situation is where the teacher gives the student an algorithm specified imperatively. Usually, teachers would ask the student to declare how he or she associates variables in the algorithm with registers, and how he associates fragments of assembly code with lines in the original specification algorithm. This is a situation where the semantics are particularly well-defined, and one of the few where the system could actually try to validate the program.

5.5 Graphical interfaces

Graphical users interfaces (GUI) are an important component of nowadays software, simplifying its use with metaphorical graphic controls, and improving output with graphical visualization. This subject is addressed in the Graphical Interfaces course.

In the programs with a GUI it is usual to separate the **interface layer** from the **computation layer**, and this separation is reinforced by the use of graphical toolboxes for building and managing GUIs. This separation is present in several high-level systems such as tcl-tk [Ous94] or Visual Basic [McF97]. The interface layer is composed of a set of graphical objects (buttons, menus, etc) that, in response to user actions, invoke procedures in the computational layer. In turn, these procedures can modify the properties of the GUI.

As shown in Figure 4, the traditional oracle model sets the program under analysis in a sequential flow, receiving data, processing it and producing results. This approach results from the interaction with the operating system proposed by Unix, but it is not adjusted to programs with a GUI. In the black box approach the "interface" (formated data IO) is just a detail that makes verification difficult. In this domain the interface itself is under evaluation which requires new approaches to concept of oracle.

In the current implementation this problem is solved by separating the verification of the two layers. The verification of the interface layer is basically comparing the two structures that represent the objects, their attributes and the spatial relationships among them. The verification of the computational layer follows the basic principle of the traditional oracle model with a twist. Instead of a single program or procedure being tested as a whole, each procedure of the procedural layer, called back from the interface layer, is compared using in the black box approach.

5.6 Imperative Languages

Undergraduate students learn several imperative languages, such as C, C++ or Java, in different courses. At this stage it is not decided which languages should be addressed first in our research. In spite of being frequently the first paradigm that is taught to undergraduates, this is the domain where the generation of oracles is more troublesome.

An oracle for any of these languages will have the constraints described previously that lead to the use of the black box model. We expect that the experience with the other domains will suggest new ideas to simplify the generation of oracles for these languages. In particular we expect to find a strategy for identifying parts of the solution program with parts of the program being graded.

6 References

[CM84] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer Verlag, Berlin, 1984.

[FD92] M. Florido and L. Damas. Types as theories. In *Proc. of post-conference workshop on Proofs and Types, Joint International Conference and Symposium on Logic Programming*, 1992.

[Fil97] Miguel Filgueiras. *A Portable Environment for Programming in MC68000 Assembly*. LIACC, Universidade do Porto, R. do Campo Alegre 823, 4150 Porto, Portugal, January 1997.

[FW65] George E. Forsythe and Niklaus Wirth. Automatic grading programs. *Communication of the ACM*, (8):275–278, 1965.

[JF] Bingchiang Jeng and Istváan Forgács. *On the Automatic Generation of Domain test Data*. Dep. of Information management National Sun yat-sen University of Kaohsiung, Taiwan.

[JL95] M.S. Joy and M. Luck. On-line submission and testing of programming assignments. *Innovations in Computing Teaching*, pages 97–103, 1995.

[Lar97] James R. Larus. *Assemblers, linkers, and the SPIM simulator*, chapter Appendix A, pages A2–A78. Morgam Kaufmann, 1997.

[Lev91] Jonathan M. Lever. Program equivalence, program development and integrity checking. In *Logic Program Synthesis and Transformation*, pages 1–26, Manchester, 1991. Springer-Verlag.

[McC76] T. McCabe. A software complexity measure. *IEEE Trans. Software Engineering*, 2(12):308–320, Dec 1976.

[McF97] P. McFedries. *Visual Basic for Applications*. Sams, Indianapolis, 1997.

[Ous94] J.K. Ousterhout. *TCL and the TK Toolkit*. Addison Wesley, Reading, MA, 1994.

[RH83] R. Rottmann and H. Hudson. Computer grading as na instructional tool. *Journal of College Science Theaching*, 12:152–156, 1983.

[UW97] J.D. Ullman and J. Widom. *A First Course in Database Systems*. Prentice Hall, New Jersey, 1997.

[Ver85] P. B. Van Verth. A system for automatically grading program quality. Technical report, SUNY (Buffalo), 1985.

[vM94] Urs von Matt. Kassandra: the automatic grading system. *SIGCUE*, (22):26–40, 1994.

[ZF91] Abdullah Mohd Zin and Eric Foxley. Automatic program quality assessement system. In *Proceedings of the IFIP Conference on Software Quality*, Vidyanagar, India, March 1991.

[Zob87] J. Zobel. Derivation of polymorphic types for prolog programs. In *Proceedings 4th International Conference on Logic Programming*, 1987.