AUTOMATING EXERCISES VALIDATION IN E-LEARNING ENVIRONMENTS

Antonio Ortega, Rubén Álvarez

Abstract: E-learning is supposing an innovation in teaching, raising from the development of new technologies. It is based in a set of educational resources, including, among others, multimedia or interactive contents accessible through Internet or Intranet networks. A whole spectrum of tools and services support e-learning, some of them include auto-evaluation and automated correction of test-like exercises, however, this sort of exercises are very constrained because of its nature: fixed contents and correct answers suppose a limit in the way teachers may evaluation students. In this paper we propose a new engine that allows validating complex exercises in the area of Data Structures and Algorithms. Correct solutions to exercises do not rely only in how good the execution of the code is, or if the results are same as expected. A set of criteria on algorithm complexity or correctness in the use of the data structures are required. The engine presented in this work covers a wide set of exercises with these characteristics allowing teachers to establish the set of requirements for a solution, and students to obtain a measure on the quality of their solution in the same terms that are later required for exams.

Keywords: E-learning, Computer Science, Data Structure and Algorithms, Abstract Data Type, Test Case Generator.

ACM Classification Keywords: testing tools, mechanical verification, diagnostic, trees, list, stacks, queues.

Introduction

Nowadays e-learning systems allow, among other services, support students while carrying out the study of a subject. The mythology considers a wide spectrum of services which allow the communication between students and teachers, thanks to the existing tools, such as different utilities for content management and visualization, chats for on -line mentoring, or forums in which students and teachers share opinions on the matter.

Some enterprises provide e-learning services such as tools for supporting the definition of test-like exams so that students may solve these exercises, providing marks and lists of errors, some of them give also a brief explanation on the correct solution, based in links to theory contents. However, we suggest that tests are no the correct choice for some materials, and that a wider kind of exercise or exam type is a must in many environment. In the one hand theoretical exams may be supported through test-like exercises, in the other, practical knowledge and skills require a more complex service, complaining not only the final solution given by the student, but also the goodness of the solution itself depending on complex criteria.

We expect to demonstrate that it is feasible to design such a service, and present a specific validator in the field of data structures and algorithms (DSA), which include the concepts of Abstract Data Types, such as stacks or queues, dynamic structures, e.g. binary trees, linked lists of some types, etc.

This system could provide students a repository, accessible through Internet, containing exams which could be solved by students, and automatically validated in exactly the same terms that teachers do in real exams. Each student could get not only a mark, but a complete list of errors, including conceptual mistakes committed while developing the solution, and then acquire the knowledge on how the code shall treat all data structures properly and how the algorithm shall look like.

Teacher collaboration is a strict requirement for the system to provide best possible results. Teachers are encouraged to manage contents, administer exercises, specify solution constraints and basic test cases that provide a way to compare program outputs.

Test case automation

The engine, we propose within this paper, is really a test case generator that automates test case definition for each program. Then the test cases are to be executed obtaining the outputs for each solution provided. The novelty is that later the evaluation is also automated, based on academic criteria. Note the similarity to general software testing.

We must keep in mind that in the software testing stages of software development we do not really desire to verify if the program works correctly, furthermore we expect to find errors. A successful test is the one which finds errors. Software testing includes different kinds of tests that allow the evaluation of different aspects of the product. Depending on the goal of the product testing is to be more or less exhaustive, e.g. fault tolerance level. For instance, testing a video game shall be different from testing a flight navigation system or the controls of a nuclear power plant.

Glen Myers [1] establishes three rules as the goals for a test:

1. A test is a process involving the program execution with the intention to find errors.
2. A good test case is the one in which there is a probability of finding an error not yet discovered.
3. A test is successful if a new error has been discovered.

The ideal cycle in a test plan should be: discover a bug, repair the bug, and finally commit again with the tests prior to fault detection. This is called regression testing, and it is used to check that the changes made while correcting bugs do not generate new bugs.

Our work is oriented to validate exam solutions that usually are reducing to little size programs, with simple structure and not much subprogramming. In this sense, and considering a subprogram as a portion of a unit to test, or as maximum a unit itself, the research is focused in unit testing, validating design and behaviour of the result [2].

Regarding unit testing two different approaches may be found. The first allows the validation of the behaviour or interface of a unit (black box test) and the other allows evaluating the structure and testing all execution paths (white box).

In [3] we can find several paradigms for test automation:

- Random generation [4], in which inputs are generated until a useful one is found.
- Symbolic testing (static analysis) [5], in which symbolic values are assigned to variables in order to obtain an algebraic expression with the goal of representing the program processing.
- Dynamic test generation [6], in which a direct search for test cases is done through program execution.

[7] Enumerates and explains the ten most important challenges in test automation, still not solved. Another way to afford the problem is based in metaheuristic techniques. Among the works in the matter, we can find [3], where three types of algorithms are discussed: genetic, simulated annealing and tabu-search. More information on this issue is found in [8], where the authors consider not only metaheuristics focused on white box testing automation,

but also the automation of black box testing. It has been considered that all works in the matter are focused in reducing the costs and effort while developing industrial systems, the complexity of this kind of projects is out of the scope of our work..

JUnit [9] is a tool that provides facilities for testing java software. It does not automate test case generation, but executes provided tests automatically. The research work in [10] presents a framework for automated verification of object oriented programs. The kernel of this system is composed of a repository of classes in XML format. In this proposal a parser or syntax analyzer inspects the source code in the selected programming language and extracts relevant information which feeds the repository. XML is used as a metalanguage for creating abstract syntax trees which provide programming language independence. Test cases are created using this repository.

Exam validation engine

Basing the solution in all mentioned issues, we relay in a lexical analyzer generator. Flex [11] is used to extract, from the plain text sent by students, the structure of the program. With this structure we can define the test cases that are required as well as input data for generate the cases.

For testing the program we can execute it, providing the proper inputs. Checking that the internal processing of the solution is correct relays in the scope of black box testing. Finding automatically the proper input data for the program is in the scope of white box texting. For this purpose to be achieved we can use a lexical analyzer generator. This implies not only generating a lexical analyzer, but a tool which, together with the analyzer, provides the capability of recognizing each part of the source code (conditions, assignments, calls to other functions, etc.).

The latest tests to be executed on the solution are black box tests. This is the reason for analyzing the code, because once parameters for the functions are known, the code manipulating the parameters is located, and equivalence classes may be calculated for each input data.

Keeping in mind that we are interested in validating subprograms (functions or procedures), due to the kind of exams in the area, and thinking of reducing the complexity of the engine, only parameters in the main function of the program are to be taken into account for black box testing.

Flex is capable, as written in its specifications, of generating test cases automatically. The question then is: Are we automating test cases or we want to achieve a further goal? If so, which is that goal?

The answer is that we want not only to generate tests automatically, and obtain inputs and outputs from programs. Our goal is to determine how good a solution is, the goodness of the programming style, properness of data structures management.

In this sense, we found in Flex a set of constraints. For solutions that require ADTs, it is necessary to count on the declaration of the structures and operations for knowing which data type are contained in each structure. On the other hand, if the solution for the exercise requires of external libraries, information about them is needed. Specifically information about the expected results for each operation in the libraries, as long as the source code of the library is usually not available.

Validating an exercise about DSA implies also checking specific aspects of the implementation, such as number of times each data is accessed. These are aspects out of the scope of Flex.

Working with Flex only provides to create a hypothetic tool that generates test cases, to execute tests on a program looking for errors, but does not guarantee the correctness of the solution in academic terms.

By that time, we decided to abandon the support given by this tool, and started evaluating the possibility of creating the engine from the scratch.

In order to provide students a validation service, we have developed a system which receives, only, the solution requested in the exam. Note that in many exercises it is assumed that a set of libraries is available, but we do not demand this additional code, only the function(s) or procedure(s) required by the statement.

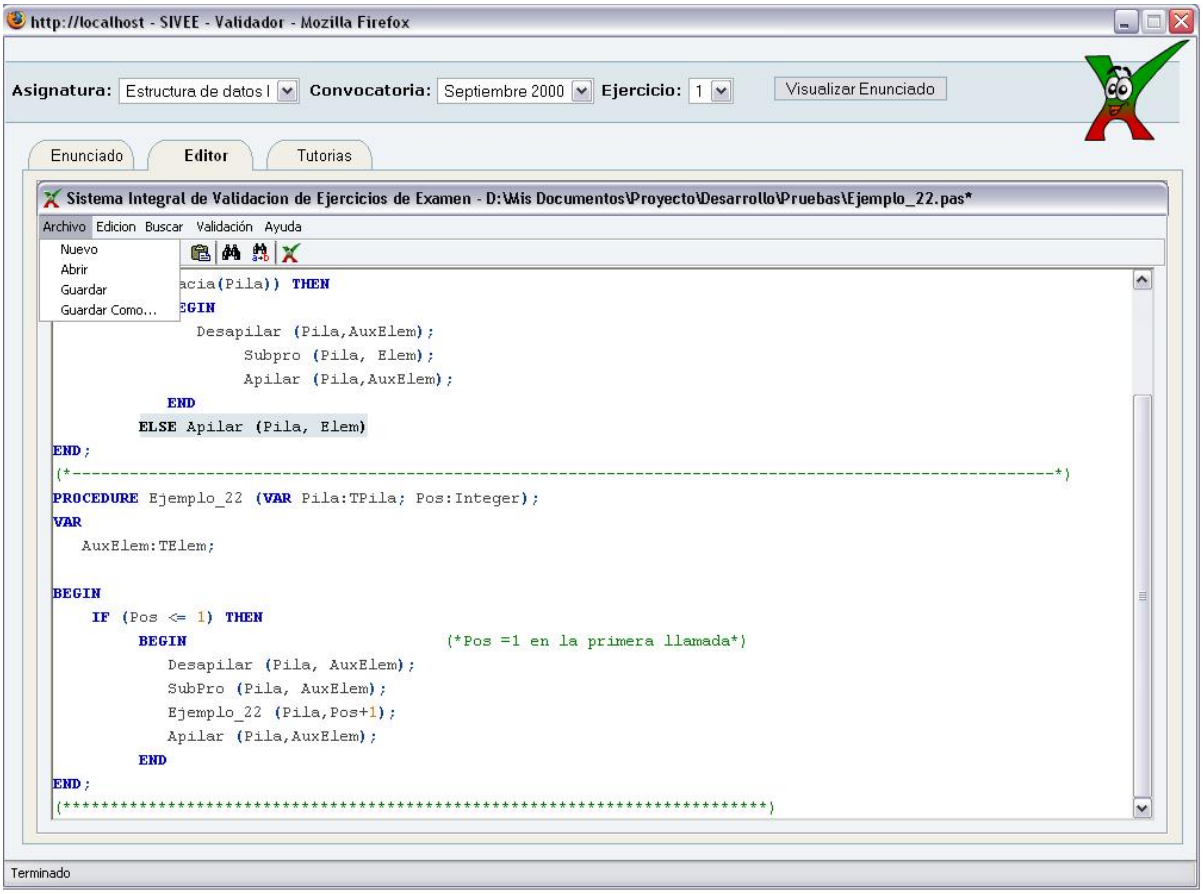

Figure 1. Interface provided to students

We needed a tool capable to generate a complete program, compliable and executable, so that we shall add to the student's code, a set of sentences, declarations of data types and structures, ADTs, initialization values, and functional verification support to obtain the outputs. Also we shall be capable to detect and check that the code provided by students is compliant with the constraints defined by teachers.

The tool is capable to generate dynamically complex data structures, while teachers are free to define any kind of list or tree, for instance, as a part of the scope of the software to be produced.

The basic architecture of the product is show below:

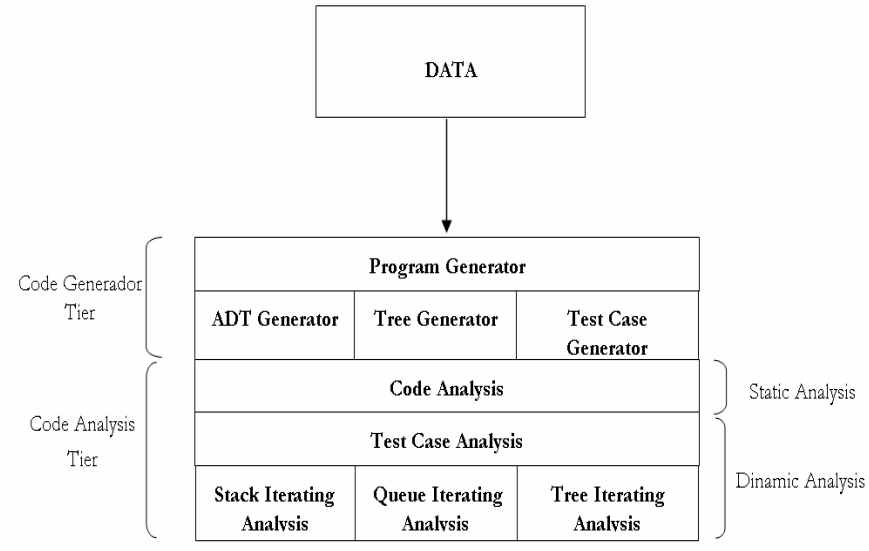

Figure 2. System architecture

The student only must complain about respecting the constraints defined for correct solutions:

- If the exam statement specifies that a subprogram is expected, and that the subprogram shall have same parameters specified by teachers, we expect students to provide a function or procedure which considers strictly the specification, in other case the exam is failed.
- The user has no limit respecting the amount of subprograms that can be written to get a proper solution, while the previous point has been respected this does not have consequences regarding the mark.
- The user must respect the names of the different data structures provided by the statement, otherwise the engine is to fail during validation, since the system is going to complete the program basing the generation in the given specification.
- The user shall provide, together with the solution, the set of initialization values for variables and parameters that make the solution work.

The engine performs also the generation of ADTs, automatically, for all simple data types and some complex homogeneous data types. It may be thought that instead generating the ADTs they could be provided by teachers, we suggest that generating them is also a base for further extensions of the system.

Static constraints to consider that a solution is valid and correct are as follows:

1. Check if the student uses auxiliary data structures that are not allowed.
2. Control the use of pointers when these are not required.
3. Validate the kind of subprogram, it shall meet the specifications (procedure, function, or any of both).
4. Guarantee that the main subprogram receives necessary parameters, in the correct form (by value, by reference), and that there are no more or less parameters.
5. If the problem request a function, ensure that the solution returns the correct data type.
6. Check the implementation, some problems request or forbid iterative or recursive solutions.

In order to verify all items mentioned above, our engine analyzes the code of the solution that shall be free of compilation errors. This reduces significantly the complexity of the system, with no lose of functionality, compilation may be carried out in local computer using some of the common compilers. After inspecting the code, relevant information is stored (subprograms used, their names and parameter names and types, whether the parameters are passed by value or by reference, main subprogram location, kind of algorithm…). This information is used to support the verification of the solution, and permits the detection of forbidden data structures and pointers (for instance looking the ^ symbol for Pascal programming language) or if the solution is based on a procedure or a function.

Dynamic constraints are the following:

1. Check the number of iterations on a data structure.
2. Track that the solution does what is expected to do while and after execution.

Code in the same language of the provided solution is generated to build a complete program, free of compilation errors and capable to be executed. Then the student's solution is aggregated to this code. All the information required for code generation is extracted from system data store. The data in this store are provided by the teachers before publishing an exercise.

The complete program is then instrumented, adding traces that are written to an output file containing the memory addresses of the nodes in the dynamic structures. Once these addresses are know, and after executing the program, the engine calculates the number of accesses per node in the structure. If some of the addresses have been accessed (non consecutive) more times than those allowed by teachers, or in an incorrect order, we can determine that the algorithm is not correct.

In the case of Queue ADT, the output of the traces points the number of *enqueue* and *dequeue* operations, and the number of times *size* service has been called. Also the operations requested from user code are tracked to check this.

The code generated by the system is in charge of requesting to the queues the number of elements at a given time. Once the file is written, a first algorithm is responsible for separating the traces depending on its kind.

Let the system know the number of queues available for an exercise, and the number of accesses to each node inside, then another algorithm is capable to verify the correctness in treating the structure.

This algorithm calculates the maximum number of iterations on each queue in the problem as follows: calculate the number of times that dequeue is requested per each structure, divide that amount by the number of elements in the queue. If the rest in the calculus is zero, then the number of iterations on the queue is the quotient of the operation, in other case the number of iterations is the quotient plus one.

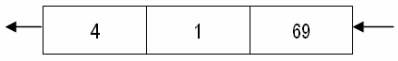

Figure 3. A queue

In the example above a queue containing three elements is shown. Following the described algorithm, if we dequeue three times we can observe that 3 dequeue operations divided by 3 elements equals 1, rest=0. The queue has been iterated once. Instead, if we dequeue and enqueue four times 4 dequeue operations divided by 3 elements equals 1, rest 1. The queue has been iterated more than one time, because one of the elements has been read twice (producing also a change in the order of the nodes of the structure, which may be also a

constraint). Note that if we unqueue 4 times, without enqueueing, then the forth operation fails, and does not compute in terms of a second iteration, because no element has been extracted.

The output of the traces introduce for the exercises using Stack ADT indicates the number of calls to *pop* and *push* operations. When the execution finishes, an algorithm extracts the traces for pop and push, in same terms that was done for enqueue and dequeue. In order to determine the number of iterations on the stack, the algorithm looks up for the number of consecutive times that pop or push operations are requested. The succession of maximum length is the base to calculate the number of iterations.

When validating that the program does what it is expected to do, the engine is supported by the information provided by teachers. The engine generates an executable program in which the declarative sentences and initialization values for parameters, from teachers, are added to create an execution of the student's program. Apart from these sentences, final values for parameters, especially those by reference are checked.

Conclusions

Once the work described has been finished, we consider that it is feasible to carry out the design and implementation of this kind of system, and that only a study for each kind of exercise and data structure is required. The basic engine, implemented during the research stage, demonstrated that automating the correction of exams and exercises, maintaining the constraints of each subject, are possible and provide a great help to students on data structures and algorithms.

As many other computing systems, of course, ours has limitations and constraints. The engine detects and reports on compilation errors or faults during execution. Regarding specific constraints, teacher dependent, we are capable to specify the kind of error committed by students, but we cannot specify which part in the code of the solution is producing the error. Either it proposes the student how to avoid this issue.

In spite of a number of types of exams on DSA cannot be solved by the engine, because of the use of ADTs not yet implemented, we consider that the engine is very useful, and supposes a great step in the matter, and that covering more ADTs is only a matter of time, once the kernel has been developed successfully.

Bibliography

[1] Myers, G., The Art of Software Testing, Wiley, 1979.

[2] Métrica 3. Ministerio de Administraciones Públicas. http://www.map.es/csi

[3] Eugenia Díaz, Raquel Blanco, Javier Tuya. Comparación de técnicas metaheurísticas para la generación automática de casos de prueba que obtengan una cobertura software. ADIS 2002

[4] Ntafos, S., On random and partition testing, Intl. Symp. On Software Testing and Analysis, 1998

[5] DeMillo, R.A., Offutt, A.J., Constraint-based automatic test data generation, IEEE Transactions on Software Engineering, 17(9).1991.

[6] Korel, B., Automated software test data generation, IEEE Transactions on Software Engineering, 16(8), 1990

[7] Rice R.W. "Surviving the top 10 challenges of software test automation." CrossTalk: The Journal of Defense Software Engineering (Mayo): 26-29. (2002).

[8] Dr. Macario Polo Usaola. Curso de doctorado sobre Proceso software y gestión del conocimiento. Pruebas del Software. Departamento de Tecnologías y Sistemas de Información. Febrero 2006.

[9] http://www.junit.org/

[10] Pedro Jesús Vázquez Escudero, María N. Moreno García, Francisco J. García Peñalvo. Verificación con XML. Departamento de Informática y Automática. Universidad de Salamanca. Noviembre 2001.

[11] Flex, version 2.5. A fase Scanner Generator. Editor 2.5, march 1995. Ver Paxson.

Authors' Information

Rubén Álvarez González – BSC in Computer Scence; e-mail: ruben.alvarez.gonzalez@gmail.com

Antonio Ortega Manchón – BSC in Computer Scence; e-mail: antonio.ortega.manchon@gmail.com