Partial Correctness and Continuous Integration in Computer Supported Education

Daniela Fonte[1], Ismael Vilas Boas[1], Nuno Oliveira[1], Daniela da Cruz[1],
Alda Lopes Gançarski[2] and Pedro Rangel Henriques[1]
[1]*Departmento de Informática, Universidade do Minho, Braga, Portugal*
[2]*Institute Telecom, Telecom Sud Paris, Paris, France*

Keywords: Computer Uses in Education, Problem-based Learning, Competitive Learning, Collaborative Learning, Automatic Grading System, Semantic Program Evaluation, Program Analysis.

Abstract: In this paper we support the idea that students and teachers will benefit from a computer-based system that assesses programming exercises and provide immediate and detailed feedback: students would be able to evolve in the right direction and teachers would follow and assess more fairly their students. This assessment should outperform the typical right/wrong evaluation returned by existing tools, allowing for a flexible partial evaluation. Moreover, we adopt a concept from Agile Development, the *Continuous Integration* (CI), to improve students' effectiveness. The applicability of CI reflects a better monitoring by the teams and their individual members, also providing the ability to improve the speed of the development.
Besides the description of the capabilities that we require from an *Automatic Grading System* (AGS), we discuss iQuimera, an improved AGS that we are working on, that implements our teaching/learning principles.

1 INTRODUCTION

When students start learning Computer Programming, they can experience difficulties to understand the basilar concepts due to a lack of adequate background (Milne and Rowe, 2002) or even adequate proficiency. Actually, Programming is a very complex task specially for beginners who need to have special skills including a high abstraction capability. Programmers need to be capable of understanding the problem statement, analyze it, sketch algorithms and implement them in a programming language. These tasks are equally important and all of them require a lot of effort to be dominated.

In particular, learning a programming language (to code the algorithm and implement the problem resolution) has a lot of complex details that need to be mastered. However, semantic issues involved in any programming language (besides the syntatic pecularities of each one) are the main obstacles to this learning process. We believe that *Problem-based Learning* approach is a good way to diminish this barrier. This is, we advocate that Programming courses should be highly practical; students should solve on the computer practical exercises since the first class, to practice the new language syntax and semantics. Following a problem-based approach, they should learn by practice to deal with all the phases of programming life cycle (since the problem analysis to the program testing). However, to assist all the students in the context of big laboratorial classes (with 20 or 30 students) is almost impossible for the teacher. In real situations, teachers can neither monitor individually all their students nor provide them an exhaustive feedback regarding the exercises they solve. This leads students to give up and fail to reach the course objectives.

From the teacher's perspective, this problem-based approach in Programming courses implies the obligation of manually analyzing and testing the code for each exercise. This time consuming task is neither simple nor mechanical: it is a complex and arduous process, prone to faults, that involves a lot of work. Different teachers may assign different evaluations to the same exercise, due to several factors like fatigue, favoritism or even inconsistency.

Summing up, we recognize that teachers are not able to fully support the students, giving feedback about the mistakes they made in every exercise. This strongly suggests that problem-based learning should be supported by powerful software tools — that is pre-

Fonte D., Vilas Boas I., Oliveira N., da Cruz D., Lopes Gançarski A. and Rangel Henriques P..
Partial Correctness and Continuous Integration in Computer Supported Education.
DOI: 10.5220/0004848802050212
In *Proceedings of the 6th International Conference on Computer Supported Education* (CSEDU-2014), pages 205-212
ISBN: 978-989-758-021-5

cisely the focus of the present paper.

Moreover, the changes introduced by *Bologna Process* (Tauch, 2004) have generalized the concept of *Continuous Evaluation* in Education. This technique fosters the work outside classes, providing space for an active and collaborative learning. It also aims at achieving a closer monitoring of each student's progress along a course, by issuing tests and individual or group works. Group-work is another relevant idea: in this way, students are able to share knowledge and take advantage of collective and collaborative activities, like brainstorming or social facilitation; they can also improve soft skills like scheduling, communication, distribution of tasks and leadership (Forsyth, 2009). On the other hand, grouping students in a team provides an efficient solution for teachers to assess multiple students through a singular assessment process, with a significant reduction of the amount of work required compared to individual evaluation (Guerrero et al., 2000).

This new trend also introduces some difficulties both for students (that need to comprehend how to follow it), and for teachers (that must find a just way to assess individually each student). These troubles can, once again, be overcame resorting to the support of intelligent software tools, like Automatic Grading Systems (AGS), as will be discussed along the following sections (this topic is the second motivation for the work here described).

After this motivation and before conclusion, the paper has 3 sections. Section 2 discusses in general the support that computers can bring to education. In Section 3 we introduce our proposal — the main characteristics that an effective AGS should have. Then, in Section 4 we show the architecture of iQuimera, an AGS that we are developing to realize our proposal.

2 COMPUTERS IN EDUCATION

Since a long time ago, computer scientists and software engineers are working on the development of tools to help teachers and students in the teaching/learning process; this gave rise, in the eighties, to the so called area CAE (Computer-aided education) that also involved people from the Education Science and Psychology fields. Two large classes of computer applications were then developed: *tutoring systems*, to help students learning (Natural and Exact Sciences); *learning management systems*, to help teachers to deal with students registry and follow-up.

The evolution of *tutoring systems* went in the direction of the development of tools to assess individually students. In a first phase, using tests (in the form of questionnaires, or similar devices) previously made by teachers; in a second phase, generating automatically those tests selecting questions from a repository or creating them from templates.

Later, with the development of computer communications and the birth of computer networks, people started the construction and use of the so called *e-Learning systems*, on the one hand combining both classes of tools, and on the other hand enabling the distance learning process.

As previously said, in this paper we are interested in a new generation of tools called *Automatic Grading Systems* (AGS) (Matt, 1994; Leal and Silva, 2003; Hukk et al., 2011) devoted to the support of Computer Programming courses. They can be distinguished according to the approach followed to evaluate the submitted program: *static* or *dynamic* analysis. *Dynamic* approaches depend on the output results, after running the submitted program with a set of predefined tests. The final grade depends on the comparison between the expected output and the output actually produced. *Static* approaches take profit from the technology developed for compilers and language-based tools to be able to gather information about the source code without executing it. The improvement of dynamic testing mechanism with static techniques (like metrics or style analysis), led to a new generation of hybrid systems such as CourseMaker, Web-CAT, eGrader, AutoLEP or BOSS that combine the best of both approaches. This symbiosis keeps providing immediate feedback to the users, but enriched by a quality analysis – which is obviously a relevant extra-value.

In the past few years, with the spreading of Agile Methodologies (Beck, 2001), some experiments (Hazzan and Dubinsky, 2003; Jovanovic et al., 2002) were conducted aimed at combining two topics: Agile Development and Education. In this context, the implementation of *Continuos Integration* (CI) (Kim et al., 2008) concept within classrooms seems to be perfectly acceptable as a way of tracking the real evolution of the students in what concerns their programming skills and knowledge about the essential programming concepts, as will be discussed in Section 4.4. With this concept, the evaluation becomes truly continuous, as the name and the original idea assumes. Nevertheless, this matter is not yet deeply explored; to the best of our knowledge there are no AGS which implement this concept.

As we discuss in Section 1, current educational methodologies invest on group work and social collaboration between students to improve their overall effectiveness in regard to learning. In fact, group work is already a reality in education, reducing the time spent by teachers on evaluation and improving

the students' effectiveness on learning. What is missing, however, is to take full advantage of its benefits. We will propose in the next section an improvement of an AGS that supports the adoption of CI practices in education, bringing to teachers valuable resources that allow them to follow and assess the students' evolution with more accuracy.

3 A PROPOSAL FOR AN EFFECTIVE AGS

According to the teaching/learning ideas exposed along Section 1, a software tool to support, on the one hand, the *problem-based learning* and, on the the other hand, the *continuous evaluation* (based on group-work) approaches should comply with the following requirements.

From the teacher's perspective, it should alleviate his tedious and error prone (manual) work of marking and grading the programming exercises submitted by students, allowing him to focus on the students' actual needs.

From the students' perpective, it should make possible to have their works automatically evaluated, finely measured and fairly graded in due time (this means, as fast as possible after each submission).

To do that, the envisaged AGS shall provide detailed reports on each submission. Those reports must say if the answer is totally or partially correct, but shall also include information on the code quality (sofwatre metrics) and on eventual code plagiarism. Student's position when compared with other students and evolution rate are other data that shall be considered in the report.

As discussed in Section 2, at this moment available AGS provide several of these basic features. However, these tools still lack some important features, mainly in what concerns the following: the acceptance of semantically correct answers, no matter its lexico-syntatical appearance; the evaluation of partially correct answers; and the ability to follow students' individual evolution in the context of a continous group work.

All of these considerations led us to build the system that will be introduced in the next section.

4 iQUIMERA

iQuimera is an improved version of our AGS first approach – the Quimera system (Fonte et al., 2012) – developed to support the features announced in Section 3. This system is a web-based application to assess and grade students' programming exercises written in the C programming language, either in *learning* or *programming contest* environments. It offers a complete management system, allowing to set up and manage contests; register students and associate them in groups; and follow up and monitor the different activities involved.

With iQuimera extension, we aim at improving our original approach in two important directions:

- *Flexibilization of the Grading process* – on the one hand, being able to accept as correct answers semantically similar to the expected one, independently of their syntactic differences[1]; on the other hand, allowing the definition of a set of partially correct answers that should also be accepted and graded[2].
- *Collaborative learning* – continuously assessing the group work, providing individual feedback about each group element.

The next subsections introduce iQuimera architecture and its main functionalities. First, we describe the basic features, already implemented in Quimera first version, and then we discuss the improvements (listed above) underdevelopment.

4.1 Quimera Initial Functionalities

Quimera is a system capable of evaluating and automatically ranking programming exercises in *learning* or *programming contest* environments, by combining a very complete static source code analysis with dynamic analysis. Thus, Quimera is able to ensure the grading of the submitted solution based not only on its capability of producing the expected output, but also considering the source code quality and accuracy. Our system guarantees that different programming styles will receive different grades even if they produce the correct output and satisfy all the requirements; this can be an advantage on learning environments, because the student is stimulated to find different correct solutions. Teachers will not waste time searching for problems that have only one solution; the approach adopted allows problems that can be solved in different manners. Moreover, they can settle more interesting problems whose solutions are a set of values, instead of a single value (this constraint, typical of the

[1] The structure of the output file produced by the student program should not be taken into consideration when comparing it to the expected output file.

[2] Using the same semantic similarity approach to compare the output produced by the student program with the partially correct outputs.

classic systems, heavily restrict the kind of statements that can be proposed).

Quimera completes its static analysis with a plagiarism detection tool, in order to prevent fraud among submitted solutions (a common issue in learning environments). To the best of our knowledge, this feature is not provided by the tools referenced above.

Quimera also allies a simple and intuitive user interface with several graphics exhibiting various statistics concerning the assessment process flow. These statistic graphics are useful for the students, as they illustrate their individual and overall evaluation and performance rates; but they are also useful for the teachers, as they provide fast views over each student, groups of students working on the same problem, or over groups working on different problems.

Notice that the feedback returned to the programmer immediately after each submission and the possibility to resubmit new solutions, encourages competitive learning. The student, after receiving the information that his submission is wrong, tends to locate the error, rewrite the program to fix it and submit it again.

4.1.1 Dynamic Analysis

Quimera *Dynamic Analyzer* module follows the traditional strict approach: the submitted (source) code is compiled and then executed with a set of predefined tests. Each test is composed of an input data vector and the correspondent output vector. Running the compiled code (submitted by the student) with one input data vector, the output produced is saved as a vector and is compared with the expected output. The student's submission passes that test if both output vectors are strictly equal (the difference between them is null). In this way, the submission under assessment is correct if and only if all the tests succeed.

After the assessment (running the submitted code with all the test vectors), the user can consult the percentage of tests successfully passed. However, he does not know how many tests there are and only has access to the tests included in the problem statement. This is useful to avoid situations of *trial and error* for *test set guessing*.

4.1.2 Static Analysis

Quimera *Source Code Analyzer* module produces a complete report about the quality of the source code submitted through a set of predefined metrics (currently 56 metrics), grouped by five classes, namely: *Size*, *Consistence*, *Legibility*, *Complexity* and *Originality* (as detailed in (Fonte et al., 2012), here ommited for the sake of space).

The different values computed automatically by this module using classic compiler technologies, when combined with the dynamic evaluation (as discussed in subsection 4.1.1), provide a fine grain assessment that is uttermost relevant for the student learning process. However to be useful in the grading process, the numerical values obtained along the metrics evaluation must be appropriately transformed. For the sake of simplicity and space we do not discuss here those transformations.

4.1.3 Automatic Grading Process

The *Grader* module takes into account the results delivered by the *Dynamic Analyzer* and by the *Source Code Analyzer* and combines them using a grading formula. This grading formula considers six different assessment categories and uses appropriate weights (that can be tuned by the teacher according to the learning objectives). Those assessment categories are the five metric classes introduced in the previous subsection (Size, Consistency, Legibility, Complexity, Cloning) plus one that results from the strict dynamic analysis and that mesures the execution success (the number of tests passed and the execution time). *Cloning* is obviously related with the Originality metric, but more pragmatically measures the percentage of duplicated code in the source code submitted.

At present we are using the following combination of weights: *Execution* (75% + 5%), *Size* (2%), *Consistence* (5%), *Legibility* (2%), *Complexity* (6%) and *Cloning* (5%). In this way, we enhance the dynamic analysis: the actual program ability to produce the correct output corresponds to 80%; the source code quality influences the final grading with just a factor of 20%.

The report issued by Quimera after the Grading phase, contains not only the final grade but all details concerned with the two previous marking phases, as depicted on both sides of Figure 1.

On the left side of Figure 1, we can see a report for an answer that only passes in 22% of the tests. This

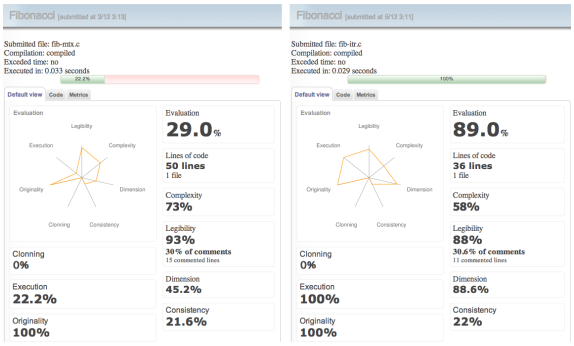

Figure 1: Two assessment report examples.

is a well documented answer (it has 15 commented lines in 50 lines of code) and its *Legibility* has a final score of 93%. The *Complexity* score is owed to the number of used variables and data structures. Since its 50 lines of code exceed the average size of the contest submissions for this problem, this penalizes the final grade on the *Dimension* category (only 45.2%). In the *Consistency* category, its 21.6% are owed to the use of several returns through the code (assuming a maximum of two returns per function as a reasonable limit to the source code consistency) and also to the use of pointers.

The right side of Figure 1 shows a report for a correct answer that passes all the tests. This answer is better documented (11 commented lines in 36 lines of code), but its *Legibility* has a lower final score (88%) once it did not used *defines*, as the other solution. The *Complexity* score of 58% is owed to the implemented *loop*. Its 36 lines of code improve its final grade on the *Dimension* category to 88.6%. In the *Consistency* category, its weak weight of 22% is also owed to the use of several returns through the code.

Finally, it is possible to see that both solutions have not been plagiarized (100% original) and students follow the good practice of not repeating their own code (0% of duplicated code). These assessments led to a final grade of 29% on the first case and a significant 89% on the second case. Quimera also completes this evaluation with radial charts to a quicker and easier comparison of the solution performance in the different grading categories.

4.2 Architecture

iQuimera architecture is organized in four levels, as depicted in Figure 2.

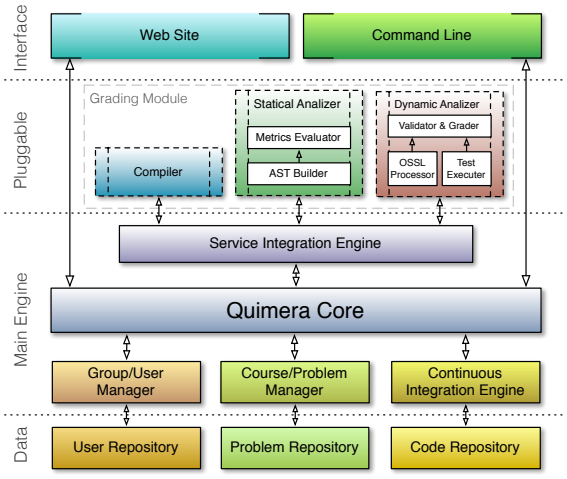

Figure 2: iQuimera Architecture.

- The *Data* level represents the data abstraction layer. It is composed of the modules related with the storage of the data concerned with contests, users, and problems in a relational Database. This level also includes the *Code Repository*, where all the students' submissions are stored.
- The *Main Engine* level works as the system core, linking and controlling the communication between the several components of the system. This level includes: the *Course/Problem* and the *Group/User* managers which are responsible for the interaction with the data associated with the problem and user management tasks; The *Continuous Integration Engine* that is responsible for keeping the *Code Repository* integrity, managing the tasks associated with *Code Repository* update and with the continuous assessment process; and finally, the *Service Integration Engine* that is the API for the communication between the plugins and the system.
- The *Pluggable* level encloses all the plugins currently available in the system. These plugins represent the different tools that can be used in the grading process. They are grouped in three types, namely: *Compilers*, *Static Analyzers* and *Dynamic Analyzers*.
- The *Interface* level is responsible for the interaction between the user and the system. Like the first version, iQuimera is a web-based system with a nice and modern web interface; however we also provide the possibility to active and work with the system via the command line.

4.3 Flexible Dynamic Analysis

As said in Sections 2 and 3, one of the main drawbacks of the existing AGS is their incapability of detecting and correctly grading alternative answers. For instance, they do not support situations where small differences in the order of the output elements do not mean a bad answer (which would be considered correct in a manual assessment).

To overcome this gap we propose a generic model to obtain a more flexible and rigorous grading process, closer to a manual one. More specifically, an extension of the traditional Dynamic Analysis concept, by performing a comparison of the output produced by the program under assessment with the expected one at a semantic level. We aim at allowing not only the specification of which parts of the generated output can differ from the expected one, but also to define how to mark partially correct answers (Fonte et al., 2013).

To implement this model, we perform a semantic comparison between both outputs based on a Domain Specific Language (DSL), specially designed to specify the output structure and semantics – the *Output Semantic-Similarity Language (OSSL)*. The OSSL design also supports the mark of partially correct answers.

For this, we are extending the grading module of Quimera with a *Flexible Dynamic Analyzer* (*FDA*). Figure 3 depicts the architecture of that extension, which is composed of three modules: the *OSSL Processor*, the *Flexible Evaluator* and the *Grader*.

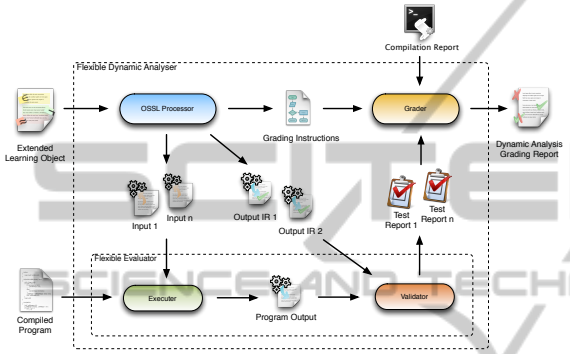

Figure 3: Flexible Dynamic Analyzer Architecture.

The *OSSL Processor* is responsible for producing the set of resources required to execute, validate and grade the submission under assessment. It receives an *Extended Learning Object* containing the problem description, the associated metadata and the *OSSL* specification, and generates the set of *Inputs*, the set of the *Expected Outputs* (through an intermediate representation, IR) and the *Grading Instructions*.

The *Flexible Evaluator* is responsible for executing and validating the submissions. It receives the set of *Inputs*, extracted from the *OSSL* specification by the *OSSL Processor*, and executes the *Compiled Program*. If an execution is successful, the *Flexible Evaluator* module produces a *Program Output* file that is validated against the respective *Output IR* – the intermediate representation of the *OSSL* specification of the expected output, generated by the *OSSL Processor*. This output IR allows to compare (at the semantics level) the meaning of the expected output with the output actually produced. This validation process produces a *Test Report* for each test performed, containing the details about time and memory consumptions and the test results.

The *Grader* module produces a *Grading Report* resulting from the dynamic evaluation performed, concerning the set of *Test Reports* produced by the *Flexible Evaluator* and the *Grading Instructions* provided by the *OSSL Processor*. This *Grading Report* is composed of the details about each individual test report and the submission assessment, which is calculated regarding time and memory consumptions, the weight and score for each test and the number of successful tests. Moreover, if the submission under assessment fails the compilation phase, this grading process is based on the *Compilation Report* provided by the compiler, in order to give feedback about the program under assessment.

Using the *FDA* only requires from the teacher to describe the test set by providing the *OSSL* definition of input and expected output for each problem. This description has only to be written once for each different problem, since this module automatically generates all the data necessary for the student's answers validation.

As an example of usage, consider an exercise where it is asked to write a program that tests which points of a giving set of cartesian coordinates can define a square. As input, the program receives a file with two integers per line, giving the cartesian coordinates of each point. As output, is expected that the student's answer produces the set of four coordinates that defines each square. Considering as input the following set of coordinates: (0,0), (0,10), (10,10), (10,0), (20,0), (20,10), (25,10), (25,0), it is possible to define two squares as depicted in Figure 4.

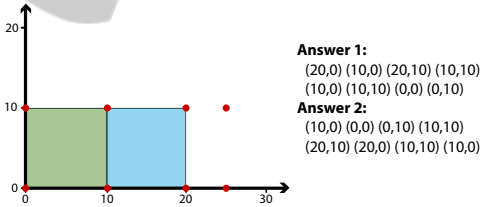

Figure 4: Cartesian representation of the exercise input.

Traditional *AGS* only accept answers containing the groups of four coordinates in a certain order. Also, answers that output only one of the two possible squares are considered incorrect. With the proper *OSSL* definition (as introduced in (Fonte et al., 2013), here ommited for the sake of space), the *FDA* module accepts answers with the correct coordinates for each square in any order or with the squares also listed in any order. Moreover, it can also accept as partially correct answers solutions that only output one of the two possible squares. On the right side of Figure 4, we can see an example of two answers accepted by the *FDA* module, that differ not only on the coordinates order inside each square, but also in the squares order.

We have confidence that the proposed approach is easy to use (*OSSL* allows to specify the output meaning in a simple way) and not difficult to implement.

We also argue that it effectively improves the role of *AGS* as Learning Support Tools, ensuring the interoperability with existent programming exercise evaluation systems that support *Learning Objects* (Hodgins et al., 2002).

4.4 Continuous Integration to Track Students' Evolution

Continuous Integration (CI) is one of the main keys for success within *Extreme Programming*. It grants the quality of the software, by controlling the development process on each small step.

CI takes advantage of the automation of a set of pipelined tasks done in order to improve the productivity in the development (Duvall et al., 2007). It starts by merging the code from multiple team members in an integration machine. This machine stores the latest version of the project and is able to integrate new changes with the existing code, following the synchronous model for team programming (Babich, 1986). This operation may produce collisions in the merging server when two or more files were changed by different team members.

Despite integration may take advantage of usual repository version control tools (like *SVN*[3] or *Git*[4]) to easily handle colisions, CI goes further in the sense that the code is always built and tested upon each commit. A valuable coproduct is the generation of success statistics and reports for every team member who performs an integration (Miller, 2008).

We are enriching Quimera evaluation process by taking advantage of the techniques used in CI, to improve students' effectiveness (Vilas Boas et al., 2013). To the best of our knowledge, CI methodology was never attempted as assessment tool within the educational environment, more specifically, in teaching programming languages to beginners.

Figure 5 depicts the Quimera system extension to include the CI methodology. This process requires the implementation of a *Code Integration Engine* (CIE) with capability to work with some of the common Version Control Systems. New code integration will be automated as much as possible, using the automerge mode of those tools to minimize collisions.

After a successful Integration, CIE builds the code, with the tools already available in the AGS. Along this process, the warnings and errors found are reported. Then, some metrics are evaluated directly from the source code to enrich the final report. As an optimization, files without changes are not reevaluated, speeding up the process.

[3]http://subversion.apache.org

[4]http://git-scm.com

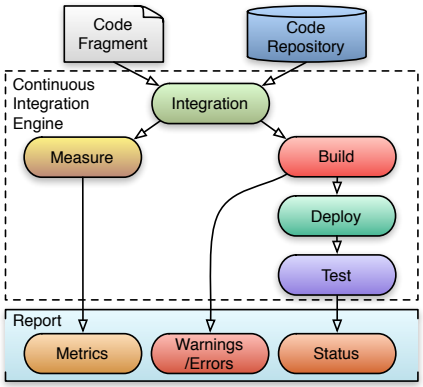

Figure 5: Continuous Integration Engine activity.

Recurring to the AGS, the executable code produced is deployed to a *sandbox* running on protected environment, in order to be tested. This allows to emulate the real environment where the code should be tested and prevents system failure that can come from the execution of malicious code. After deployment, the set of test units is run and a report about the code correctness is generated. This report is the main result of the integration. Finally, another report is generated with details about the status of each step of the process described. This report, available in several formats, is sent to the students and teacher according to their preferences.

iQuimera requires a new set of configurations. To administrators, these new settings are applied at system's initial setup. A configuration routine and documentation will be available to simplify this process. To teachers, the continuous integration process is automatic, so does not require any extra effort. To students, the use of a *CVS* requires machine specific configurations according to their development environment. This may be the most difficult step to the success of its adoption.

As briefly referred in Section 2, there are some good results that arose from the application of *Agile Development* (mainly *Extreme Programming*) methods in educational environments. However, they are applied mostly on advanced years of programming courses due to the programming knowledge level required to follow the complex rules of Extreme Programming. With novices, these will hardly bring positive results, which led us to design this approach to be more suitable for beginners.

We have in mind that this proposal will not soften the steep curve for programming concepts acquaintance, once the basilar concepts will remain hard to grasp. However, with the feedback on continuous evolution on this matter will allow to focus where the problem really exists. Moreover, it also provides a fair

assessment, since students are followed closely and measured for their evolution and implicit effort. All these ingredients contribute to augment the students motivation and enthusiasm.

5 CONCLUSION

We discussed the difficulties inbred in teaching and learning Programming Languages and proposed iQuimera, a system developed in the scope of computer aided education to help making this process more successful through *problem-solving paradigm*. iQuimera extends Quimera, an hybrid (static and dynamic) *Automatic Grading System* (AGS) in what concerns students evaluation approaches.

Our contribution is two fold. On one hand, we propose a more flexible assessment model accepting answers that have syntactic differences (although semantically correct), and also partially correct answers. On the other hand, we propose to apply *Continuous Integration* (CI) principles for a continuous assessment of group elements.

Although iQuimera requires more configuration effort from the three actors enrolled than the traditional *AGS*—more time to specify the problem, test vectors, and grading mechanisms (teacher), to manage all the participants and processes (admin), and to use the system in collaborative mode (students), as well as, more sensibility to tune appropriately the system (teacher)—, it is not an effort necessary each time someone wants to use the system, but it shall be done only once per course or exercise. Thus, we believe that this configuration effort is surpassed by the advantages iQuimera provides with CI and *Flexible Dynamic Analysis* adoption.

We plan (1) to finish the implementation of the *Flexible Dynamic Analyser* and the *CI* engine; (2) to develop new front ends in order for iQuimera to accept other programming languages than C; (3) to extend iQuimera such that it performs with minimal students intervention. Afterwards, we plan to test iQuimera with real users by means of a carefully sketched real case study, in order to verify its usefulness and benefits.

ACKNOWLEDGEMENTS

This work is funded by National Funds through the FCT - Fundação para a Ciência e Tecnologia (Portuguese Foundation for Science and Technology) within project "Projeto Estratégico - UI 752 - 2011-2012 - Ref. PEst-OE/EEI/UI0752/2011".

REFERENCES

Babich, W. A. (1986). *Software configuration management: coordination for team productivity*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Beck, K. (2001). Manifesto for Agile Software Development. *http://agilemanifesto.org*.

Duvall, P. M., Matyas, S., and Glover, A. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional.

Fonte, D., Boas, I. V., da Cruz, D., Ganarski, A. L., and Henriques, P. R. (2012). Program analysis and evaluation using quimera. In *ICEIS'12*, pages 209–219.

Fonte, D., Cruz, D. d., Gançarski, A. L., and Henriques, P. R. (2013). A Flexible Dynamic System for Automatic Grading of Programming Exercises. In *SLATE'13*, volume 2, pages 129–144.

Forsyth, D. R. (2009). *Group dynamics*. Wadsworth Publishing Company, Belmont, 5 edition.

Guerrero, L. A., Alarcon, R., Collazos, C., Pino, J. A., and Fuller, D. A. (2000). Evaluating cooperation in group work. In *Groupware CRIWG 2000*, pages 28–35.

Hazzan, O. and Dubinsky (2003). Teaching a software development methodology: the case of extreme programming. In *CSEE&T 2003*, pages 176–184.

Hodgins, W. et al. (2002). Draft Standard for Learning Object Metadata. *IEEE 1484.12.1-2002*, pages i–44.

Hukk, M., Powell, D., and Klein, E. (2011). Infandango: Automated Grading for Student Programming. In *ITiCSE '11*, page 330. ACM.

Jovanovic, V., Murphy, T., and Greca (2002). Use of extreme programming (XP) in teaching introductory programming. *FIE'02*, 2:F1G–23.

Kim, S., Park, S., Yun, J., and Lee, Y. (2008). Automated Continuous Integration of Component-Based Software: An Industrial Experience. In *Automated Software Engineering*, pages 423–426.

Leal, J. P. and Silva, F. (2003). Mooshak: a Web-based multi-site programming contest system. *Software: Practice and Experience*, 33(6):567–581.

Matt, U. v. (1994). Kassandra: the automatic grading system. *SIGCUE Outlook*, 22(1):26–40.

Miller, A. (2008). A Hundred Days of Continuous Integration. In *AGILE '08*, pages 289–293.

Milne, I. and Rowe, G. (2002). Difficulties in learning and teaching programming—views of students and tutors. *Education and Information technologies*, 7(1):55–66.

Tauch, C. (2004). Almost Half-time in the Bologna Process - Where Do We Stand? *European Journal of Education*, 39(3):275–288.

Vilas Boas, I., Oliveira, N., and Rangel Henriques, P. (2013). Agile Development for Education effectiveness improvement. In *SIIE'13*, Viseu, Portugal.