Automated Assessment of Complex Programming Tasks Using SIETTE

Ricardo Conejo, Beatriz Barros, and Manuel F. Bertoa

***Abstract*—This article presents an innovative method to tackle the automatic evaluation of programming assignments with an approach based on well-founded assessment theories (Classical Test Theory (CTT) and Item Response Theory (IRT)) instead of heuristic assessment as in other systems. CTT and/or IRT are used to grade the results of different items of evidence obtained from students' results. The methodology consists in considering program proofs as items, calibrating them and obtaining the score using CTT and/or IRT procedures. These procedures measure overall validity reliability as well as diagnose the quality of each proof (item). The evidence is obtained through program proofs. The SIETTE system collects and processes all data to calculate the student knowledge level. This innovative method for programming task evaluation makes it possible to deploy the whole artillery developed in this research field over the last few decades. To the best of our knowledge, this is a new and original contribution in the area of programming assessment.**

***Index Terms*—Automated Grading, Automatic Programming Assessment, Evidence-Centered Design, Item Response Theory, Computer Supported Education**

I. Introduction

Programming assignments are an important part of practical tasks in engineering degree programs. Their evaluation and grading can be done either manually by teachers, semi-automatically (some tasks are supported by a computer), or automatically. In any of those cases, the task is tedious and time-consuming. The automation of the evaluation of assignment tasks benefits both teachers and students [1]. Automatic assessment is better than human in some aspects and worse in others. Humans understand the program better and more deeply than an automatic system and easily prevent or ignore minor, insignificant errors. On the other hand, consistency is higher in automatic assessment because the computer always applies the same rules without subjective biases. However, immediate results and feedback are the key factors in favour of automatic assessment.

We are aware that *"Testing can only prove the presence of bugs, not their absence"*, as Professor Edsger W. Dijkstra said, and that the method we are proposing here does not completely replace the need for human teaching. We support current trends for blended learning, and our idea is not to substitute teacher feedback but rather to facilitate the grading of student programming exercises and provide a framework to support well-founded assessment.

Ricardo Conejo and Beatriz Barros are with E.T.S. Ingenieria Informatica, University of Malaga, Spain. Manuel F. Bertoa is with E.T.S. Ingenieria Telecomunicacion, University of Malaga, Spain. E-mail: conejo,bbarros,bertoa@lcc.uma.es

The automation approach is time-consuming in the preparation and definition of the task and obliges the teacher to define a precise and computable criterion to be applied to a student's solution. However, in its favour, it results in a more precise and objective evaluation in addition to allowing the teacher to document and reuse this work in the future.

There are many systems that carry out automatic assessment of practical assignments, summarized in surveys such as [2], [3], [4], and recently, [5]. Some of the systems referred to in the surveys just accept or reject the solution; others use a set of proofs that accumulate points and calculate a final grade using just the arithmetic mean or the arithmetic mean of a weighted sum, where weight is defined heuristically by the teacher.

The evaluation of programming assignments is a complex task that requires several layers of knowledge and skill. Therefore, it is not enough to get a pass/no-pass or a heuristic value as the outcome of a set of proofs applied to students work. It is a challenge to get the fairest and most accurate value. In our research, we focus on the grading problem, that is, *How the final grade is calculated from the different proofs applied to the students program?* There are assessment theories such as Classical Test Theory (CTT) and Item Response Theory (IRT), that provides accurate measures and quality indicators for test-based assessment, that is, assessment based on a set of questions, commonly called items, but they are not currently applied to programming assignment evaluation as far as we know.

The main contribution of this work is the proposal of a technique to allow automatic assessment of programming assignments by using these well-founded assessment theories (CTT and IRT) instead of heuristic assessment as other systems do. To the best of our knowledge, this is a new and original contribution. The advantages of this approach are derived from the advantages of using a well-founded theoretical frameworks and all their measures, quality indicators, and calibration processes that can be applied. To sum up, the article tries to illustrate two issues: (1) that it is feasible to conceive the evaluation of a programming assignment as a CTT or IRT assessment problem; and (2) that these techniques provide richer information and control over the assessment of programming assignments.

In order to support our contribution, this technique has been implemented and tested as a flexible and configurable extension of a web-based assessment system called SIETTE, described in the following sections. This extension enables the evaluation of programming assignments with different types of proofs, and their integration with the CTT and IRT features of SIETTE. The system is independent of the programming language, and the proofs can be of different types. SIETTE also implements several textual and graphical forms of output of the different measures of these theories.

The underlying idea is inspired by Evidence-Centered Design (ECD) methodology, which is a guideline for designing, producing, and delivering educational assessments [6], summarized in Section II-A. A second source of inspiration is the Constraint Based Modelling (CBM) field, which is summarized in Section II-B. In a nutshell, different program proofs are viewed as constraints, or test items that can be satisfied or not, and the results are integrated into a well-founded statistical assessment theory.

In this paper we are going to present the system's extension and the different experiments carried out to show the advantages of using this approach. This article is organised as follows. First, background concepts and some related work are provided. Then, a short description of SIETTE and some features related to the research are presented. Next, the method for the assessment of programming tasks is described in detail and illustrated with a toy case. After that, some representative experiments are described, and illustrated by applying the method to three different subjects. We have selected the subjects from the various subjects available to cover the different types of analyzes and studies available. The system is compared with previous works under different perspectives, and finally discussion, conclusions, and future lines of research work are presented.

II. Related Work

There are many systems for evaluating practical assignments, some of them cited in the introduction. In this section, we focus on three research fields closely related to our proposal: evidence-centered design, constraint-based modelling, and the automatic scoring of programming assignments.

A. Evidence-Centered Design (ECD)

ECD is a theoretical and methodological framework introduced by [6]. It is defined as a *framework that makes explicit the structures of assessment arguments, the elements and processes through which they are instantiated, and the interrelationships among them"* - [7]. ECD treats assessment as a process of reasoning from the necessarily limited evidence of what students do in a test environment to claims about what they know and can do in the real world [8].

ECD considers a task model (in our case, the task model is the programming assignment) as well as an evidence model made up of two submodels: the evaluation model (defines the observable elements of each task) and the statistical model (defines how the evidence is turned into knowledge-level measures).

In our case, the evidence is obtained through proofs (evaluation programs, measurements, etc.) gathered by external tools, all of them launched from the SIETTE platform. The use of external tools to carry out the proofs offers the advantage of being able to achieve different types of evaluations: black-box, code structure, performance, similarity, etc., with specialised programs on quality measurements. The evaluation model is implemented in SIETTE by a dichotomous or a polytomous item model, and the statistical method is accomplished using CTT or IRT.

B. Constraint-Based Modelling (CBM)

The CBM paradigm for building Intelligent Tutorial Systems (ITS) is based on Ohlsson's theory of learning from performance errors [9], according to which, incomplete or incorrect student knowledge can be used within an ITS to provide guidance. The application of CBM is very simple: once a student has finished solving a problem, constraints are checked against the student's solution. The satisfaction condition of a constraint specifies properties that the solution must fulfil to be correct. The performance of a student with respect to the constraints, i.e., the list of violated and satisfied constraints in each solution, form a part of his student model.

In our approach, our evaluation model defines a set of proofs to be applied to the student program. This plus ECD gives the *assessment model for problem solving environments* described in detail in [10]. This technique has been successfully used to evaluate disciplines such as project investment [11], Object-Oriented Programming and Simplex algorithms [12].

From the CBM point of view, each proof is a kind of constraint. We have extended SIETTE with functions to declare evaluation scripts for program assignments as a set of proofs and features for communication with external tools and translate proof results into a dichotomous or polytomous evaluation model.

C. Automatic Scoring

The systems referred to in surveys on automatic evaluation or program assignments such as [2], [3], [4] or [5] use black-box methods with proofs (input, expected output) to evaluate students code. These proofs yield a list of successes or failures used as input to calculate a grade for the practical work. There are different proofs to be applied to practical work in various areas: about the submission process, code style, compilation process, program execution (with two phases to be judged: correctness and program efficiency), and the similarity between the student code and a model provided by the teacher. [13] identify three main approaches to classify the existing automatic grading of student programs: dynamic analysis, software metrics and source code analysis. [14] classifying the automated marking of programs into black-box testing and source code assessment.

Positive proofs give points. These points are the input of a heuristic mode to calculate the final score. Usually, the heuristics consists in an average of points or a weighted sum of points, depending on the proof. Other proposals, such as [15], combine these black-box methods with a study of the quality of code, both combined in a weighted sum of values. They use linear regression methods to calculate the values of a weighted sum, using a model built with a dataset assessed manually by a teacher. [16] describe some types of tests used for the most recent systems to assess student programs. They propose metrics implemented in ProgTest tool, for the C and Java languages, and consider three testing tools: unit testing tools, coverage testing tools, and mutation testing tools. They used Java and C specific testing tools. These proofs are combined using a variety of heuristics to calculate the final score. As far as we know, none of the existing systems in the literature exploit the advantages of IRT to assess practical assignments (see [17] for a description of the IRT approach).

III. The SIETTE Assessment Environment

SIETTE is a web-based tool for managing and administering computerised assessments (see [18] and [19]). The system incorporates item banking, test building, delivery, and result presentation and analysis. It supports Classical Test Theory (CTT), Item Response Theory (IRT) and Computer Adaptive Testing (CAT). Currently, it has more than 40,000 users from various universities in Spain, Ecuador, and Chile. There is a plug-in to integrate SIETTE with Moodle, the Learning Management System (LMS), used in the Virtual Campus of the University of Malaga (VC-UMA). Since 2009, all students registered in the VC-UMA have had direct access to SIETTE, which is viewed as another Moodle activity. Scores obtained in SIETTE are automatically transferred to the VC-UMA.

A. Assessment Modes and Item Types

SIETTE supports multiple item and assessment modes beyond the classical multiple-choice test. Items are stored in an item bank labeled with metadata. Basically, an assessment poses a set of items to collect evidence about the student knowledge level. Assessments are configured defining different item selection, evaluation and finalization criteria, posing modes, assessment mode, etc. According to the conditions defined, the assessment can be taken once or multiple times. In the case of a programming assessment, the items are the test cases that evaluate the correctness and quality of the program. In this section, we describe SIETTE's internal items and how a programming test case can be viewed as an item in itself.

Assessment modes are the set of rules that are used to obtain the student score. There are several pre-defined modes, but they can be grouped in three types: (1) *Percentage of correct answers*. In this case, each item is correct or incorrect, and the model just finds the ratio. (2) *Heuristic scoring*. In this case some points are assigned to each response option and the score is calculated in terms of the relationship between the points obtained by the student and the maximum points available. (3) *Item Response Theory*, that will be explained later.

Internal items in SIETTE are of three types: (1) Multiple-choice, single answer; (2) multiple-choice, multiple answer, and (3) short-answer. Each of these internal types has its own evaluation associated procedure for each assessment mode i.e. the correctness, the points, or the item characteristic curve (ICC) associated with each response option. Short-answer items allow students to enter a short text as an answer, which is recognized by a set of patterns provided by the teacher and assigned to the corresponding answer option. SIETTE provides different types of patterns for different uses. The most commonly used are regular expression patterns. For instance, a question could ask: What is the value of π? The pattern $* < 3.14 \mid 3.15 > *$ will accept a line of text that includes a numerical value of between 3.14 and 3.15 as the correct answer, and any other as incorrect. For instance, the answer 3.1416 approx. would be considered correct.

In addition to the basic internal items, SIETTE supports composed items, which are a set of internal items that are always posed together. A composed item could be, for example, a small physics problem where the stem asks for the acceleration, the velocity, and the space of a solid, given certain conditions. Composed items can be evaluated as a single item or by considering each of its components as an item. This second method is commonly used because it provides richer evidence about student knowledge.

SIETTE also supports rich interactive questions that are played by Java applets or Javascript in the web browser. The student interaction is finally transformed into an option selection or into a short text, which is recognized by a multiple-choice or by a short-answer internal item (see [20] for a detailed description of this mechanism). Moreover, SIETTE has defined its own high-level protocol to call an external web application, which plays an item. SIETTE passes the item data to the external application, including the stem and expected results, and the external application returns to SIETTE the student answer turned into a multiple-choice option or a short text to be recognized by an internal short-answer item pattern. This mechanism also supports composed items.

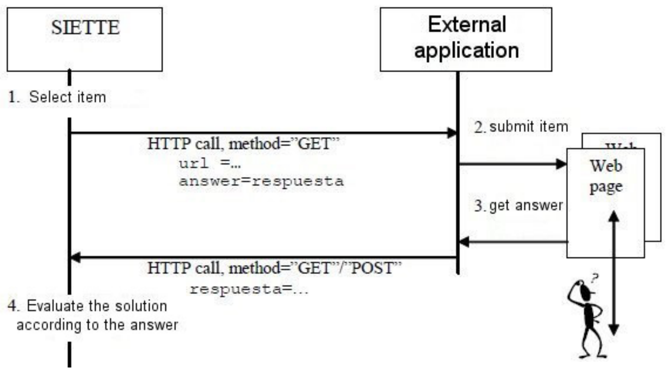


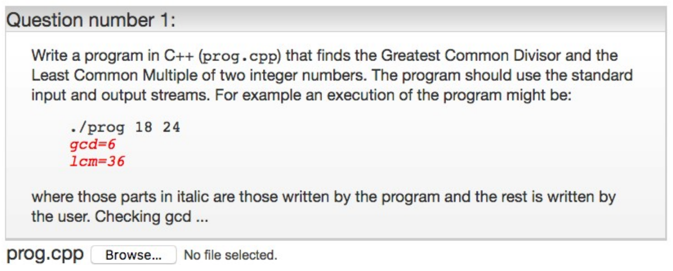
Fig. 1. Overview of the protocol to play external items.

When an external item is selected, SIETTE generates an *HTTP* call (Figure 1) (using get or post methods) to a given url with some given parameters, including at least two predefined ones. The first is the returning *url* parameter, that is, the *url* that should be called back by the external application once the presentation of the item has been completed. This returning *url* includes the item, student and session identifiers, so that SIETTE can process the answer after returning. The second is the answer parameter, which tells the application the name of the parameter that should be returned to SIETTE containing a response string. SIETTE and the external application communicate solely through *HTTP* calls, and so they do not have to be on the same machine. However, the external application is displayed in the same frame of the browser used by SIETTE, so the student perceives it as the same session.

B. The File-Item Player

Special kinds of external items have been defined for those items that require the student to send a file or a set of files as his answer. This is the feature used to evaluate programming tasks. In this case, the external application is just a SIETTE component, called SFIP (witch stands for SIETTE File-Item Player) that is able to execute a high-level script language, called SPS (witch stands for SIETTE Processing Script), developed specifically for this extension of SIETTE. In addition to passing the stem and the expected results to any external item player, in this case, SIETTE passes over to the SFIP the script, the file name, and optionally, a directory containing auxiliary files and receives back the file or the set of files submitted by the student and the evaluation results.

We now explain the whole process with a programming toy example. We ask the student to compute the greatest common divisor (gcd) and the least common multiple (lcm). Figure 2 shows how the question is posed. The question asks the student to submit a file that will be named prog.cpp and which will be compiled and executed passing arguments through the command line.


Question number 1:

Write a program in C++ (`prog.cpp`) that finds the Greatest Common Divisor and the Least Common Multiple of two integer numbers. The program should use the standard input and output streams. For example an execution of the program might be:

```
./prog 18 24
gcd=6
lcm=36
```

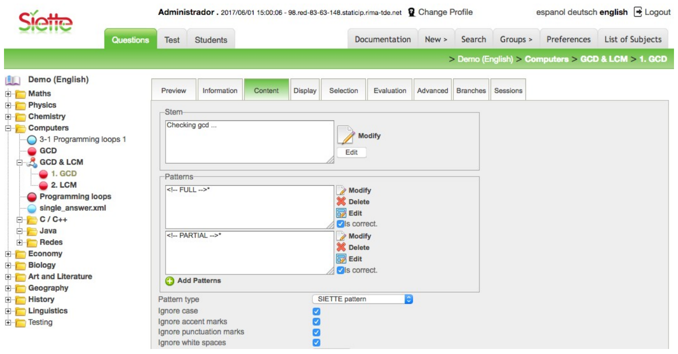
where those parts in italic are those written by the program and the rest is written by the user. Checking gcd ...

prog.cpp Browse... No file selected.

Fig. 2. Programming task item example (student view).

Suppose that the teacher wants to check whether the program works for some test cases, but wishes to check ***gcd*** and ***lcm*** independently. Additionally, for those programs that do work, a complexity check is run. In this simple case, the teacher will define a composed item in SIETTE with three short-answer sub-item components, one for each function, plus a complexity measure (Figure 3). For each component, three assessment options are defined: `FAIL`, `PARTIAL`, and `FULL`. For this example, a simple heuristic scoring approach for the assessment mode will be used, so the teacher will assign 0 points to `FAIL`, 0.5 points to `PARTIAL`, and 1 point to `FULL`. [1] The teacher should also include in this sub-item a section of the entire SPS (Figure 4), which is entered in the Advanced tab.

The processing script declares the set of commands that should be executed to evaluate the submitted file. It first compiles the code and checks that the result gives no error. If successful, it continues; otherwise it stops (`@OnError stop`). Then, the script executes the program proofs with the given examples and checks that the results match the expressions given using the (`@SIETTEPattern`). If the result matches, it continues processing; if it fails, it skips to



Stem: Checking gcd ...

Patterns: <!-- FULL -->* ✓ Is correct. <!-- PARTIAL -->* ✓ Is correct.

Pattern type: SIETTE pattern
Ignore case ✓
Ignore accent marks ✓
Ignore punctuation marks ✓
Ignore white spaces ✓

Fig. 3. Programming task item example (teacher view).

the next sub-item (`@OnError skip`). The third sub-item has a different proof. It is only executed if the first and second have been completed, and it tests the program code complexity using the score provided by the GNU complexity command.[2]

```
@SIETTEPattern: true, true, true, true
@OnError stop
@Eval max correct
g++ -w -o prog prog.cpp <> *error*
chmod +x prog == *
@OnError skip
@Answer 1
./prog 18 24    | grep gcd == *<6>*    -> PARTIAL
./prog 192 224  | grep gcd == *<32>*   -> PARTIAL
./prog 110 154  | grep gcd == *<22>*   -> FULL
@Answer 2
@OnError stop
./prog 18 24    | grep lcm == *<72>*    -> FAIL
./prog 24 18    | grep lcm == *<72>*    -> PARTIAL
./prog 192 224  | grep lcm == *<1344>* -> PARTIAL
./prog 110 154  | grep lcm == *<770>*  -> FULL
@Answer 3
complexity -h prog.cpp | grep Highest == *<1>* -> SIMPLE : COMPLEX
```

Fig. 4. Example of a file-item processing script

The returned value of this process is the label associated with the last command successfully executed for each sub-item (that is, either `FAIL`, `PARTIAL`, or `FULL`) for the first two sub-items and `SIMPLE` or `COMPLEX` for the third. In addition, the source file submitted is saved for subsequent use.

When the control is returned to SIETTE, the behavior is similar to a student answering the given question with the text `FULL` or `PARTIAL`. From this point, the item is treated as any other internal item; the heuristic score assessment mode is applied, the appropriate feedback message could be triggered, [3] the next item selection procedure is following, etc.

To sum up, each sub-item can be evaluated with a simple true or false model (dichotomous item) or a graded model (polytomous item). Each sub-item may contain one or more proofs: regarding the quality of code, its performance, the outcomes of a given input, the similarity of the solution with a specified one, etc. Commands could compile and execute the code, but they could also include calls to static code analysis, software measuring tools, test-case generators, etc. The important issue is that after executing a set of proofs, a label is returned to SIETTE and that label is recognized as a response option. Using composed items, the teacher can define

[1] Notice that there is no need to include the `FAIL` option, because anything not matching the given patterns is considered to be wrong.

[2] See http://www.gnu.org/software/complexity/

[3] Note that different labels can be used to recognises common student errors and provide adaptive feedback.

as many sets of proofs as needed, and they can be executed independently.

According to the Evidence Centered Design (ECD) terminology, the file-item player presents a programming exercise to the student, provides a form to submit a set of files as the result (the *Presentation Process*), and applies a given set of rules (the *Evidence Rules*) to obtain a string that represents the evaluation of the exercise (the *Work Product*). That string is passed to an underlying open answer internal item in SIETTE, which assigns a score to that evaluation string (the *Response Scoring Process*).

C. Grading Procedures and Analysis Tools

Once the file items corresponding to programming tasks have been completed, they are treated in SIETTE as any other item and, therefore, CTT and IRT can be used for grading and to analyze test and item performance and quality.

A major advantage of using SIETTE to evaluate programming tasks is the possibility of using SIETTE's built-in grading procedures and test and item analysis tools. This is a key feature that, to the best of our knowledge, differentiates this proposal from other systems that automatically correct programming tasks.

1) Test Analysis: In general, an assessment (also called a test in this section), is comprised of a set of items. In the case of programming tasks, there might be a single composed item with multiple sub-item components or multiple independent and differentiated small programming tasks or even a combination of both. Each time a student takes an assessment, it is called a *student session*. The teacher can configure the assessment to allow or disallow, multiple sessions of the same student (retries), and to apply date and access constraints.

SIETTE includes tools to analyze the assessment results. They include descriptive statistical data such as the number of sessions, number of items per session, final-score distribution histogram, time-spent histogram, etc.

Moreover, SIETTE calculates three of the most commonly used test reliability indicators in the CTT: Cronbach's α (raw and standard); Guttman's λ_4 and Spearman-Brown's coefficient calculated using random test halves. If IRT is used as the grading procedure, SIETTE also provides the Test Information Function (TIF).

2) Item Analysis: SIETTE provides a set of tools to analyze the performance and quality of the items. They include descriptive statistical data such as the number of sessions in witch items have appeared, a time-spent histogram, absolute and relative frequency of response; and CTT and IRT indicators.

Classical Test Theory indicators include the difficulty index, the discrimination index, and the point-biserial correlation between item and test results; the tetrachoric correlation matrix and Cohen's kappa are among the items. SIETTE also calculates the indicators for each item response option. If the IRT is used as the grading model, SIETTE shows the Item Characteristic Curve (ICC) of the dichotomous classical three-parameter model (3PL); the Item Information Function (IIF); and the curves associated with each response option in the polytomous Graded Response Model (GRM).

D. Validity and Reliability

It is not common practice in the field of automatic programming assessment to refer to the psychometric concepts of validity and reliability of the scoring procedure. *Validity* refers to how well a test measures what it is purports to measure. Lets assume that the validity of the human grading of programming tasks is the ground truth, that is, the criteria imposed by the teacher to measure a student's programming ability. The only source of error in this case would be the interpretation of the student's submission, that is, the assessment reliability, which could be measured by the correlation between two different human graders, but is usually not considered. In the case of automatic assessment, validity is ensured in the same way because the teacher explicitly declares the criteria. However, human raters cannot always elicit their criteria, and sometimes they make a holistic assessment of the student's submission. Another source of problems is that the system applies the rules strictly, and human raters are, in general, more flexible. For instance, an automatic programming assessment system will apply a null score to programs that do not strictly follow the naming instruction, while human assessment will penalize that error but assign some credit to the rest. Thus the validity problem remains. In the early stages of the system's development a study was conducted to measure validity as the correlation between machine and human scores in a particular case. The results are presented in Section IV-A1.

Reliability is the degree to which an assessment tool produces stable and consistent results. That is, an assessment is reliable if it produces the same results each time it is employed. The system proposed in this article allows the application of techniques based in standard measurement theories techniques to assess the reliability. The reliability indicators are shown for all cases and listed in Section IV-A2.

Of course, validity and reliability are closely linked to the specific assessment case, and conclusions cannot be extrapolated universally. However, one of the advantages of the system proposed in this article is that reliability is automatically measured and can be improved by standard psychometric procedures (i.e., discarding some items, calibrating item curves, choosing the most appropriated assessment model, etc.).

In Section IV, different use cases are presented. They are used to show different features of the system, demonstrate their usefulness, and provide information for good practices in automatic-programming tasks assessment.

IV. Use Cases

The framework was first developed in 2009, although some features were not added until later versions after we had resolved the problems of previous ones. It has been used for formative and summative assessment in different subjects at the School of Computer Engineering, the School of Electronic Engineering and the School of Mathematical Science at the University of Malaga. The exercises proposed consisted in student exercises that took between ten minutes to six hours of programming. At the time of writing of this paper, the system has been used in 9 different subjects, which means that 96 different assessments have been taken by 1,748 different students and around 40,000 sessions have been completed. Table I shows the number of sessions by year, showing increasing use of the system.

TABLE I
NUMBER OF SUBJECTS, ASSESSMENTS, STUDENTS AND SESSIONS BY YEAR

Year	Subjects	Assessments	Students	Sessions
2009	1	6	130	244
2010	1	3	88	163
2011	1	4	86	216
2012	3	13	188	938
2013	2	8	170	1,589
2014	4	15	212	2,481
2015	5	24	371	3,382
2016	6	32	644	8,956
2017	9	68	826	14,515

In the following sections, we have selected some of the subjects to illustrate the different features of the system and system usage.

A. Compiler Construction

The SIETTE system has been used since 2001 in the Compiler Construction course at the School of Computer Science at the University of Malaga. The application of the module to evaluate programming tasks was earlier introduced in this subject back in 2009. The automatic assessment of programming exercises was a component of the final score of computer tests and other classical written exercises. Most of the content has been reused at that time; firstly, some programming tasks were used in summative assessments and later were left open to students to practice. Table 3 shows the usage data. The number of files submitted by the students indicates the number of different tasks involved in the assessment. A single submission might be a zipped file containing several files.

The programming assignments, required the development of a program written in C, LEX, and YACC, (currently in JAVA, JFLEX, and CUP), [4] which implements the compiler or the interpreter of a high-level language called PLX, which is a simplified version of a C/JAVA-like programming language. The language and the basic implementation are explained during the course. In the final exam, students have to extend a previous version of their own compiler code to cover some new PLX language features. The code is around 2,000 lines. For example, they are asked to include language functionality for a new expression operator, or a new control sentence, or support for a new data type, etc. Students have around six hours to complete the exam in a controlled environment.

1) Validity studies: Validity is an important concern, especially in the early application of automatic assessment of these programming tasks. In order to guarantee that the automatic assessment produces results equivalent to human assessment, we have undertaken various experiments, shown in Table II.

[4]LEX and YACC are classical scanner and parsing generator tools used as part of the UNIX environment. JFLEX and CUP are equivalent tools for JAVA.

A programming task was proposed and was defined in the same way as an automatic assessment, with a clearly defined stem, a clearly defined expected output, and a clearly defined scoring schema. The students worked on this assignment and submitted the files to the teacher. Another teacher marked the exercise manually (following the scoring schema strictly). After the score was obtained, an automatic assessment was defined, and we fed the system with the files provided by the students previously (ID=13887). That produced a new score which was compared to the human score. A total of 34 exercises were submitted. The average score was 28.4 (std. dev. = 33.5) points in the human score and 13.2 (std. dev. = 24.1) in the automatic score. The correlation coefficient between the two distributions was just 0.55. However, average statistical data did not provide good information in this case. A careful review of the data, showed two important facts: (1) the human score was always higher than the computer score; and (2) there were seven cases that were marked by the computer as null, which explained almost all the differences. Removing those cases, we found a new correlation coefficient that we named the *underlying correlation coefficient*, which is very high (0.98).

We repeated the experiment in September with a subset of the same students (ID=14027). In this case, we did the opposite; we first allowed only a single submission to the computer assessment tool, and then a teacher manually inspected the code, evaluated it, and resubmitted it to the computer assessment tool after minor errors were resolved. In five cases, it was found that a human evaluator would have not considered the initial computer assessment to be correct. The error detected in these cases varies: the students did not follow the guidelines for file naming; they leave extra debugging messages on the output; the output was redirected to a fixed file, not to the standard output; etc. Discarding these cases, the *underlying correlation coefficient* was 1.0, which is perfect match.

We concluded that the differences in scoring between the human and automatic assessment were explained by the fact that the human was able to consider some errors as simple mistakes without major importance, while the computer applied the rules strictly and consider the same errors as major and marked them null. Thanks to this experiment, we learned that we have to give more opportunities to students to correct their minor mistakes in order to improve the validity of automatic assessment.

TABLE II
CORRELATION BETWEEN AUTOMATIC AND HUMAN SCORING

ID	Students	Human Score Mean (s.d.)	Computer Score Mean (s.d.)	Correlation Coefficient	Incorrect Cases	Underlying Correlation Coefficient
13887	34	28.4 (33.5)	13.2 (24.1)	0.55	7	0.98
14027	23	57.8 (17.2)	42.6 (32.3)	0.71	5	1.00
14747	52	49.26 (32.53)	45.25 (35.86)	0.92	7	1.00

A similar study was undertaken the following year with a new group of students (ID=14747). In this case, three submissions were allowed. Students were told that the final score would be the average of the three submissions. If we compare the first submission to the last submission, we find a correlation coefficient of 0.92 and that 14 students improved their scores. We then proceeded in the same way as explained in the previous paragraph but only considered the final submission. The goal was to detect if a fourth submission of the students' code, after removing minor errors (according to the teacher criteria), would obtain a higher score. In this case, we found just seven unfair scores (out of 52), and the differences were much fewer. In addition to the problems detected previously, we noted another source of potential problems for automatic assessment: there were minor differences in the server and the local environment. Some (unnecessary) C libraries were not present in the server, and that produced some unexpected compilation errors if the students had referred to them in their code. These experiments led us to improve the system by adding some facilities to improve validity that are discussed in Section VI.

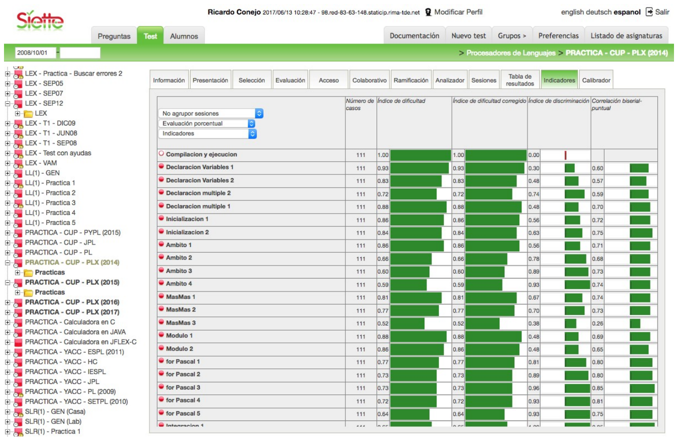
2) Reliability studies: Whenever more than a minimum of students takes an assessment, SIETTE automatically computes Cronbach's α; Guttman's λ_4; and Spearman-Brown's reliability. At first glance, the reliability of the automatic assessment is extremely high. In most cases, Cronbach's α is above 0.9, which is a very high value, and it is similar with the other indicators. As expected, reliability increases with the number of items. When a single program is submitted, which is quite common in the Compiler Construction assessment, the number of items is the number of components of the composed item, that is, the number of different proofs.

The same students can submit the task several times, and that could affect the measure of the reliability. However, assessment reliability does not change very much if we take just one session per student, i.e., the last or the highest scoring submission. SIETTE allows filtering those data and recalculating the indicators. Table III shows the reliability indicators for the last 10 assessments of this subject. Here, the indicators are calculated using just the highest scoring session of each student. Reliability slightly increases in all cases.

TABLE III
RELIABILITY INDICATORS FOR THE LAST 10 ASSESSMENTS

ID	#students	#sessions	All Sessions			#Sessions	Highest Score Session		
			α	λ_4	SB		α	λ_4	SB
18347	131	836	0.99	1.00	1.00	131	0.98	1.00	0.99
18527	123	1133	0.97	0.98	0.99	123	0.96	0.98	0.98
19727	103	277	-	-	-	103	-	-	-
20167	96	127	0.86	0.71	0.76	96	0.99	0.84	0.89
20207	36	54	0.61	0.78	0.70	36	0.65	0.77	0.77
20487	89	724	0.96	0.98	0.99	89	0.98	0.99	0.99
22048	25	98	0.97	1.00	0.99	25	0.97	0.99	0.98
22287	62	432	0.92	0.98	0.96	62	0.91	0.97	0.92
26768	33	61	0.86	0.93	0.92	33	0.92	0.94	0.92
27948	35	290	0.96	0.99	0.99	35	0.97	1.0	1.0

3) Item, test, and student result analyzes: SIETTE provides an interface that shows the classical test theory indicators for each item, including item difficulty, discrimination, and point biserial correlation between item and assessment result. The item difficulty is simply defined as the percentage of cases that have solved the item, while the item discrimination indicates the difference between the percentage of success of high-scoring and low-scoring students. In this case, the data differ depending on whether we consider all sessions, or just the highest scoring session submitted by the same student. Figure 5 shows a screenshot from the SIETTE interface for the test with ID=18527, considering just the highest scoring session of each student. In this table, rows are the items (proofs), and columns show item identifier, number of valid cases, difficulty index, corrected difficulty index (not applicable in this case), discrimination index, and point biserial correlation. The first row corresponds to the compilation proof, which is accomplished by 100% of the valid submissions. This item has no discrimination and it has been marked as *not evaluable*. The rest of the items are correct, with more or less difficulty but positive and high correlation with the test results. A low or negative value in these columns will appear in red indicating a potential problem with that proof. Using this interface, the teacher can detect those problems at a glance.

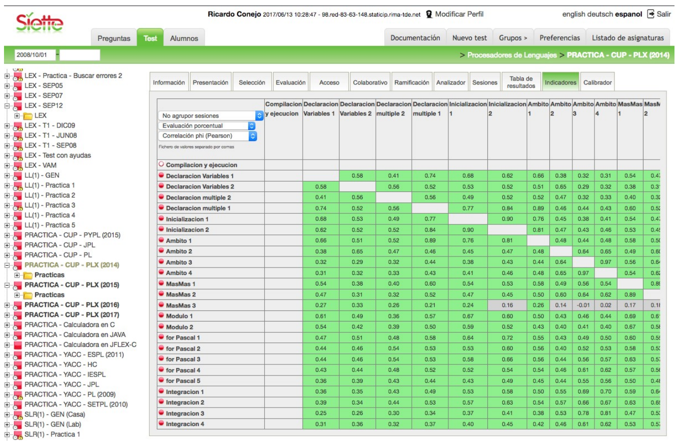


	Número de casos	Índice de dificultad	Índice de dificultad corregido	Índice de discriminación	Correlación biserial-puntual
Compilacion y ejecucion	111	1.00	1.00	0.00	
Declaracion Variables 1	111	0.93	0.93	0.30	0.60
Declaracion Variables 2	111	0.83	0.83	0.48	0.57
Declaracion multiple 2	111	0.72	0.72	0.74	0.59
Declaracion multiple 1	111	0.88	0.88	0.48	0.70
Inicializacion 1	111	0.86	0.86	0.56	0.72
Inicializacion 2	111	0.84	0.84	0.63	0.75
Ambito 1	111	0.86	0.86	0.56	0.71
Ambito 2	111	0.66	0.66	0.78	0.68
Ambito 3	111	0.60	0.60	0.89	0.73
Ambito 4	111	0.59	0.59	0.93	0.74
MasMas 1	111	0.81	0.81	0.67	0.74
MasMas 2	111	0.77	0.77	0.70	0.73
MasMas 3	111	0.52	0.52	0.38	0.26
Modulo 1	111	0.88	0.88	0.48	0.69
Modulo 2	111	0.86	0.86	0.48	0.65
for Pascal 1	111	0.77	0.77	0.81	0.80
for Pascal 2	111	0.73	0.73	0.89	0.80
for Pascal 3	111	0.73	0.73	0.96	0.85
for Pascal 4	111	0.72	0.72	0.93	0.81
for Pascal 5	111	0.64	0.64	0.93	0.75

Fig. 5. Example of item analysis. Classical test theory item indexes.

It is also important to analyze the internal correlation between assessment items. A low correlation between one item and the rest of the items might indicate that item is not working in accordance with the rest of the items and suggest that item should be removed to increase reliability. However, a very high correlation close to 1.0 indicates that those items (proofs) are continually returning the same information for the final assessment, and therefore they are redundant. Figure 6 shows a partial view of the item correlation matrix for the same assessment shown in Figure 5. The green cells indicate positive meaningful correlation; the gray cells indicate low but not significant correlation, and red cells (there are none in this example) indicates negative correlation.

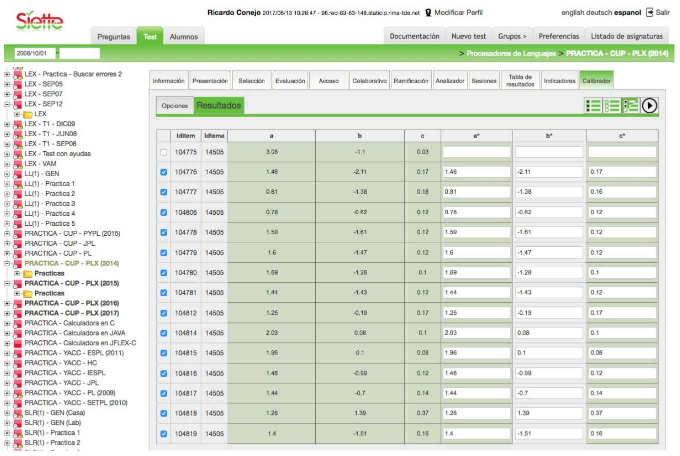
Items can be calibrated using IRT models. The calibration is done using an external calibration service. Currently SIETTE can call JICS (Java Item Calibration Service, which has been implemented by our team) or the classical MULTILOG [21]. A web service wrapper has been constructed in order to call MULTILOG and receive the results. To accomplish that, SIETTE incorporates an interface that arranges data and generates the input file in MULTILOG format. It receives and reads the returned file. Item parameters are presented to the teacher who should approve the modification of previous data if any exists (see Figure 8). The table in figure 7 shows the current ICC parameters a, b, and c for each item. a*, b*, and c* are the new proposed values. Information about model fitness



	Compilacion y ejecucion	Declaracion Variables 1	Declaracion Variables 2	Declaracion multiple 2	Declaracion multiple 1	Inicializacion 1	Inicializacion 2	Ambito 1	Ambito 2	Ambito 3	Ambito 4	MasMas 1
Compilacion y ejecucion												
Declaracion Variables 1			0.58	0.41	0.74	0.68	0.62	0.66	0.38	0.32	0.31	0.54
Declaracion Variables 2		0.58		0.56	0.52	0.53	0.52	0.51	0.65	0.29	0.32	0.38
Declaracion multiple 2		0.41	0.56		0.56	0.49	0.52	0.52	0.47	0.32	0.33	0.40
Declaracion multiple 1		0.74	0.52	0.56		0.77	0.84	0.89	0.46	0.44	0.43	0.60
Inicializacion 1		0.68	0.53	0.49	0.77		0.90	0.76	0.45	0.38	0.41	0.54
Inicializacion 2		0.62	0.52	0.52	0.84	0.90		0.81	0.47	0.43	0.46	0.53
Ambito 1		0.66	0.51	0.52	0.89	0.76	0.81		0.48	0.44	0.48	0.58
Ambito 2		0.38	0.65	0.47	0.46	0.45	0.47	0.48		0.64	0.65	0.49
Ambito 3		0.32	0.29	0.32	0.44	0.38	0.43	0.44	0.64		0.97	0.56
Ambito 4		0.31	0.32	0.33	0.43	0.41	0.46	0.48	0.65	0.97		0.54
MasMas 1		0.54	0.38	0.40	0.60	0.54	0.53	0.58	0.49	0.56	0.54	
MasMas 2		0.47	0.31	0.32	0.52	0.47	0.45	0.50	0.60	0.64	0.62	0.89
MasMas 3		0.27	0.33	0.26	0.21	0.24	0.16	0.26	0.14	-0.01	0.02	0.17
Modulo 1		0.61	0.49	0.36	0.57	0.67	0.60	0.50	0.43	0.46	0.44	0.69
Modulo 2		0.54	0.42	0.39	0.50	0.59	0.52	0.43	0.40	0.41	0.40	0.67
for Pascal 1		0.47	0.51	0.48	0.58	0.64	0.72	0.55	0.43	0.49	0.50	0.60
for Pascal 2		0.44	0.46	0.54	0.53	0.53	0.60	0.56	0.40	0.52	0.53	0.58
for Pascal 3		0.44	0.46	0.54	0.53	0.58	0.66	0.56	0.44	0.56	0.57	0.63
for Pascal 4		0.43	0.44	0.48	0.52	0.52	0.54	0.54	0.46	0.61	0.62	0.57
for Pascal 5		0.36	0.39	0.43	0.44	0.43	0.49	0.45	0.44	0.55	0.56	0.50
Integracion 1		0.36	0.35	0.43	0.49	0.53	0.58	0.50	0.55	0.69	0.70	0.59
Integracion 2		0.39	0.34	0.44	0.53	0.57	0.63	0.54	0.57	0.66	0.67	0.63
Integracion 3		0.25	0.26	0.30	0.34	0.37	0.41	0.38	0.53	0.78	0.81	0.47
Integracion 4		0.31	0.36	0.32	0.37	0.40	0.45	0.42	0.46	0.61	0.62	0.53

Fig. 6. Example of item analysis. Classical test theory item indexes. Pearson correlation matrix among items.

is shown separately. The teacher can try filtering sessions to increase the goodness of fit of the IRT models.

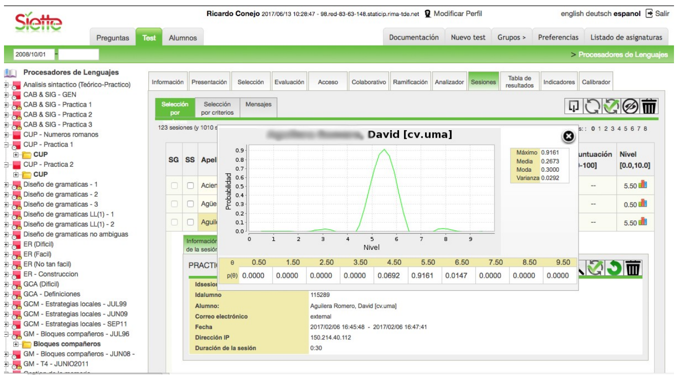


	IdItem	Idtema	a	b	c	a*	b*	c*
☐	104775	14505	3.08	-1.1	0.03			
☑	104776	14505	1.46	-2.11	0.17	1.46	-2.11	0.17
☑	104777	14505	0.81	-1.38	0.16	0.81	-1.38	0.16
☑	104806	14505	0.78	-0.62	0.12	0.78	-0.62	0.12
☑	104778	14505	1.59	-1.61	0.12	1.59	-1.61	0.12
☑	104779	14505	1.6	-1.47	0.12	1.6	-1.47	0.12
☑	104780	14505	1.69	-1.28	0.1	1.69	-1.28	0.1
☑	104781	14505	1.44	-1.43	0.12	1.44	-1.43	0.12
☑	104812	14505	1.25	-0.19	0.17	1.25	-0.19	0.17
☑	104814	14505	2.03	0.08	0.1	2.03	0.08	0.1
☑	104815	14505	1.96	0.1	0.08	1.96	0.1	0.08
☑	104816	14505	1.46	-0.99	0.12	1.46	-0.99	0.12
☑	104817	14505	1.44	-0.7	0.14	1.44	-0.7	0.14
☑	104818	14505	1.26	1.39	0.37	1.26	1.39	0.37
☑	104819	14505	1.4	-1.51	0.16	1.4	-1.51	0.16

Fig. 7. IRT Calibration.

When the calibration is complete and the IRT parameters have been defined for each item, the tasks can be reassessed using IRT Bayesian or maximum likelihood criteria. This is a data-driven technique so the scoring does not depend on heuristic but rather on statistical behaviour of the different proofs. Moreover, the IRT assessment can provide the distribution of probability of the student having a given knowledge level (in the scale defined by the teacher) as a result. That distribution can be inspected for each student and can help with decision-making in high-stakes assessment for those students closer to the pass-no-pass border. Figure 8 shows the distribution of probability of a given student having a certain knowledge level on the scale 0 to 10 (commonly used in Spain). For instance, the student shown in this figure has a probability 0.91 of having knowledge between 4 and 5,and just 0.089 of having knowledge less than 5, which is the common pass/no-pass threshold.

SIETTE also includes graphical representation of the Items Characteristic Curves (ICC) as well as the information function for each item and the test information function (IIF/TIF),



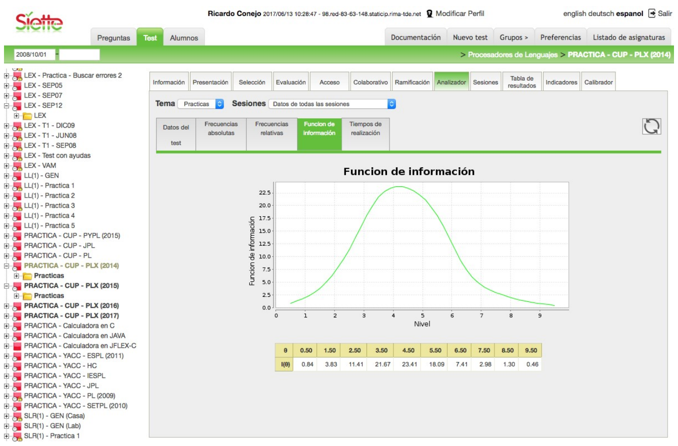
Máximo	0.9161
Media	0.2673
Moda	0.3000
Varianza	0.0292

θ	0.50	1.50	2.50	3.50	4.50	5.50	6.50	7.50	8.50	9.50
p(θ)	0.0000	0.0000	0.0000	0.0000	0.0692	0.9161	0.0147	0.0000	0.0000	0.0000

Idsesion	
Idalumno	115289
Alumno:	Aguilera Romero, David [cv.uma]
Correo electrónico	external
Fecha	2017/02/06 16:45:48 - 2017/02/06 16:47:41
Dirección IP	150.214.40.112
Duración de la sesión	0:30

Fig. 8. Student result analysis.

as shown in Figure 9. Roughly speaking, the ICC gives the conditional probability of success for an average student with a given knowledge level in this proof, while the IIF, witch is closely related to the ICC, provides information about the suitability of this proof for a given student knowledge level. The TIF is the summary of all the IIFs involved in the test. For instance, the TIF shown in figure 9 indicates that the test fairly discriminates among students with a knowledge level between 1 and 7, but not students with a knowledge level above 8. That is, if it is necessary to discriminate among students with knowledge levels of 8, 9, or 10, the test should include more difficult questions (see [17] for a better description of meaning of these curves).



θ	0.50	1.50	2.50	3.50	4.50	5.50	6.50	7.50	8.50	9.50
I(θ)	0.84	3.83	11.41	21.67	23.41	18.09	7.41	2.98	1.30	0.46

Fig. 9. IRT test information function.

B. *Design & Analysis of Algorithms*

The SIETTE system has been used since 2012 in the Design & Analysis of Algorithms course at the School of Computer Science of the University of Malaga. The course includes four or five practice tasks that should be implemented in the Java programming language. The proposed problems are classical in a subject like this in which it is important to implement efficient algorithms and to evaluate that the student code is original. The system is used to automatically correct home programming assignments.

1) Embedding External Program Verifiers: Before using the SIETTE environment to correct student programs, correction in some cases was undertaken semi-automatically using different methods. In some instances JUnit test cases were developed to help with program verification. For instance, for the N-Queens problem, the teacher developed a skeleton of the program, which had to be completed by the student. He also implemented different JUnit test cases to test whether or not the student program found one or all the possible solutions and check that the important methods were correctly implemented. The SIETTE implementation is very straightforward. SIETTE simply calls JUnit through the command line and checks that the result is correct. A single line included in the SIETTE script does the trick:

```
java -cp .:n-queens.jar:/usr/share/java/junit4-4.11.jar \
org.junit.runner.JUnitCore NQueensTest1 | head -2 | tail -1 == ... -> 10
```

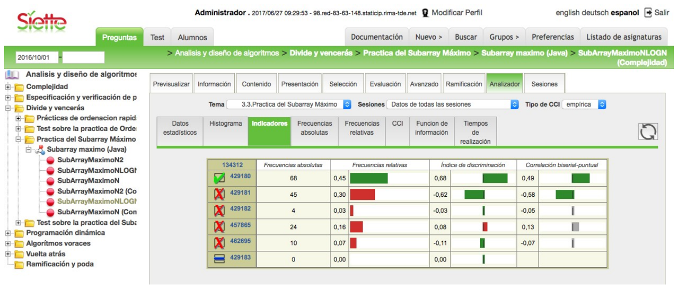
The program complexity is another important issue to take into account in this subject. To illustrate this issue, let us focus on the *maximum subarray problem*, [5] which can be implemented with algorithms of differing complexities. The brute force solution is $O(n2)$, and the divide and conquer is $O(nlog(n))$. However, there is a better solution of $O(n)$.

Analysing the present Java program complexity is difficult and not always accurate. A library (called aeca.jar) for this task was implemented in the last course and included experimentally in the assessment. The program just tests the code dynamically with different values of a parameter *n*, and checks the execution time. The output is a table of $n * execTime(n)$. The program decides, from among a given set of complexity functions ($O(n)$, $O(nlog(n))$, $O(n^2)$, $O(n^3)$, etc..), which is closer in the asymptote to the table values. This proof is implemented in SIETTE by adding the following line to the processing script:

```
java -Xmx1g -cp aeca.jar:. TestMaximumSubArray Complexity
```

That is, a test program that makes use of the aeca.jar API is executed. It returns an estimation of the program complexity as a label that could be the following: 1, N, $NLOGN$, $N2$, $N3$, etc. The correct answer for the divide and conquer solution should be $NLOGN$. This proof is included as part of the set that the program should pass. It is beyond the scope of this article to describe the details of this evaluation, which is an interesting problem in itself. However, this is a good example to show how the features of SIETTE item analysis can be used to detect potential assessment problems.

In this particular case, Figure 10 shows the graphical information for the item analysis based on the 151 selected test sessions with ID=26828. A single session with the highest score has been selected from each student. This is almost the equivalent of selecting just the last submission of each student. The relative frequency of responses indicates that this proof has been passed 45% of the cases, (corresponding to a complexity of $O(nlog(n))$ in the first line of the table) while an unknown incorrect answer has been detected 30% of the cases (this case also includes incorrect programs, shown in the second line of the table). An incorrect answer corresponding to a complexity of $O(n2)$ is detected in 4% of the cases (third line of the table). An incorrect answer corresponding to a complexity of $O(n)$ is detected in 16% of the cases (fourth line of the table), and in 7% of the cases, the complexity is evaluated as constant (fifth line of the table). The discrimination index is shown in the fourth column and the point biserial correlation with the test result in the fifth column. The graphical information represents positive values to the right and negative to the left. If they are correct they are shown in green; otherwise, they are shown in red to make them more easily seen. Grey indicates that there are too few data to be fully conclusive at 95% confidence. Some item option (i.e., the ones corresponding to complexity of $O(n)$) have a positive correlation with a good assessment result where negative values are expected, because it is a wrong option. Therefore, this item (proof) is not working properly. Maybe the test program is unable to correctly detect the complexity in some cases, or there may be another explanation. In either case, this is evidence that this proof should be not be used for assessment and should be modified. For this reason, SIETTE includes a feature to deactivate an item indicating, that it is not available for grading.



Administrador . 2017/06/27 09:29:53 - 98.red-83-63-148.staticip.rima-tde.net

2016/10/01 -

> Analisis y diseño de algoritmos > Divide y vencerás > Practica del Subarray Máximo > Subarray maximo (Java) > SubArrayMaximoNLOGN (Complejidad)

Tema 3.3.Practica del Subarray Máximo Sesiones Datos de todas las sesiones Tipo de CCI empírica

134312	Frecuencias absolutas	Frecuencias relativas		Índice de discriminación		Correlación biserial-puntual	
✓ 429180	68	0,45	🟩	0,68	🟩	0,49	🟩
✗ 429181	45	0,30	🟥	-0,62	🟩	-0,58	🟩
✗ 429182	4	0,03	🟥	-0,03	🟩	-0,05	▨
✗ 457865	24	0,16	🟥	0,08	🟥	0,13	▨
✗ 462695	10	0,07	🟥	-0,11	🟩	-0,07	▨
429183	0	0,00		0,00	🟩		

Fig. 10. A potential problem with an item detected by SIETTE.

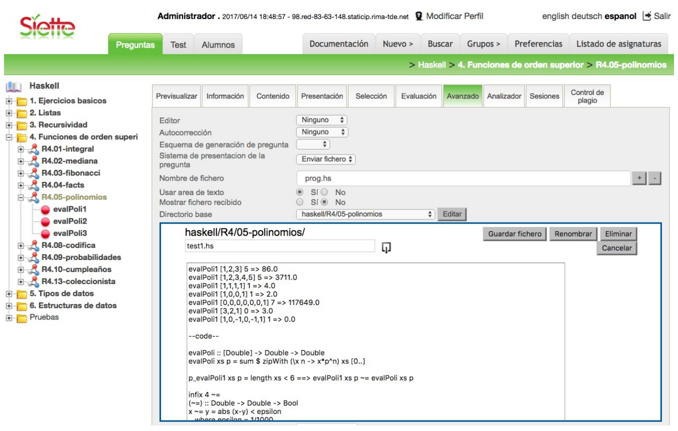
C. Haskell

The SIETTE system has been in use in the Introduction to Programming course at the School of Mathematical Science of the University of Malaga since 2015. The automatic assessment of programming tasks has been principally used to evaluate the students' home programming assignments. These assignments consist in the development of relatively small functions (2 to 50 lines of code) in the Haskell programming language. The score obtained in SIETTE represents 20% of the final score. There are two main differences between this subject and the subject presented in the previous section: (1) the students are younger and have less experience with programming; and (2) a single assessment is composed of 2 to 10 short exercises (tasks), and each exercise is evaluated with 1 to 4 different proofs.

1) Authoring domain-specific evaluation: The SIETTE environment can support different program verification tools. The Haskell language includes the module *QuickCheck* that can be used to test a Haskell function. Some of the proofs at Haskell are based on this tool, while others are based

[5]This problem is briefly described in https://en.wikipedia.org/wiki/Maximum_subarray_problem

on a comparison of the student solution with an alternative implementation provided by the teacher. To speed up the process of creating exercises and generating proofs, Haskell teachers have developed their own specification file that is parsed and merged with the student code. That parser has been added as a plug-in to SIETTE. It allows teachers to enter the specification file within the SIETTE authoring tool. Figure 11 shows the teacher interface, where they can directly define the Haskell script and auxiliary functions that are going to be used as the first proof of a task that requires implementing some functions for polynomials. The teacher introduces his/her own script that is going to be used to test the student code.

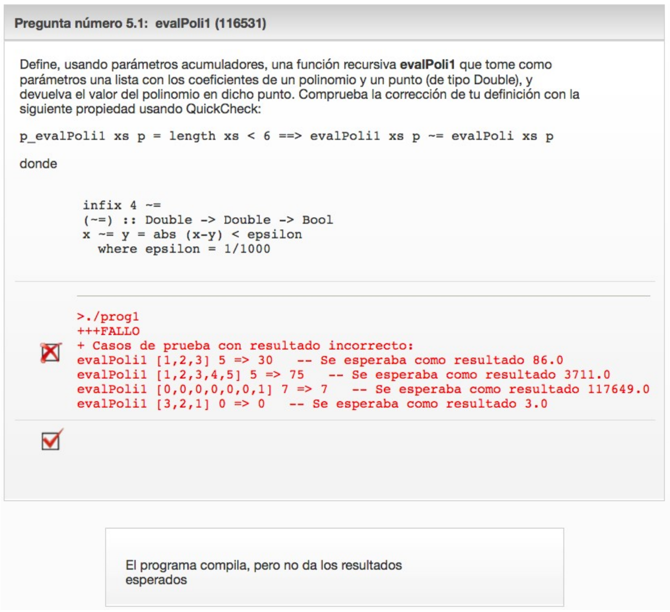


Editor	Ninguno
Autocorrección	Ninguno
Esquema de generación de pregunta	
Sistema de presentacion de la pregunta	Enviar fichero
Nombre de fichero	prog.hs
Usar area de texto	[✓ Sí] No
Mostrar fichero recibido	Sí [✓ No]
Directorio base	haskell/R4/05-polinomios

haskell/R4/05-polinomios/
test1.hs

```
evalPoli1 [1,2,3] 5 => 86.0
evalPoli1 [1,2,3,4,5] 5 => 3711.0
evalPoli1 [1,1,1,1] 1 => 4.0
evalPoli1 [1,0,0,1] 1 => 2.0
evalPoli1 [0,0,0,0,0,0,1] 7 => 117649.0
evalPoli1 [3,2,1] 0 => 3.0
evalPoli1 [1,0,-1,0,-1,1] 1 => 0.0

--code--

evalPoli :: [Double] -> Double -> Double
evalPoli xs p = sum $ zipWith (\x n -> x*p^n) xs [0..]

p_evalPoli1 xs p = length xs < 6 ==> evalPoli1 xs p ~= evalPoli xs p

infix 4 ~=
(~=) :: Double -> Double -> Bool
x ~= y = abs (x-y) < epsilon
   where epsilon = 1/1000
```

Fig. 11. SIETTE authoring tool.


Pregunta número 5.1: evalPoli1 (116531)

Define, usando parámetros acumuladores, una función recursiva **evalPoli1** que tome como parámetros una lista con los coeficientes de un polinomio y un punto (de tipo Double), y devuelva el valor del polinomio en dicho punto. Comprueba la corrección de tu definición con la siguiente propiedad usando QuickCheck:

```
p_evalPoli1 xs p = length xs < 6 ==> evalPoli1 xs p ~= evalPoli xs p
```

donde

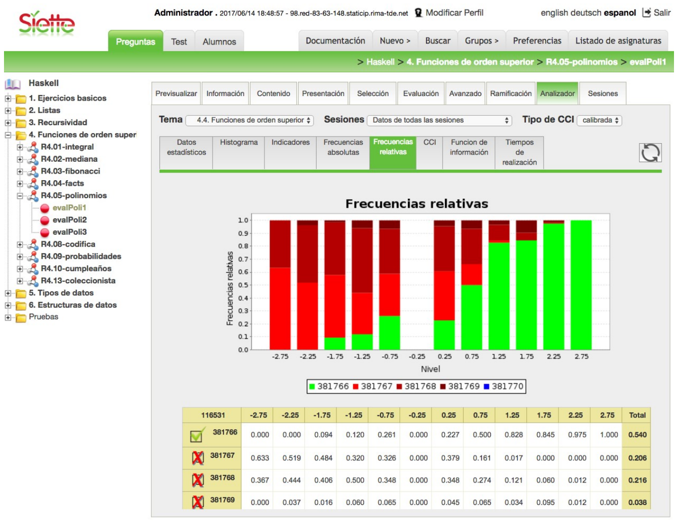
```
infix 4 ~=
(~=) :: Double -> Double -> Bool
x ~= y = abs (x-y) < epsilon
  where epsilon = 1/1000
```

```
>./prog1
+++FALLO
+ Casos de prueba con resultado incorrecto:
evalPoli1 [1,2,3] 5 => 30   -- Se esperaba como resultado 86.0
evalPoli1 [1,2,3,4,5] 5 => 75   -- Se esperaba como resultado 3711.0
evalPoli1 [0,0,0,0,0,0,1] 7 => 7   -- Se esperaba como resultado 117649.0
evalPoli1 [3,2,1] 0 => 0   -- Se esperaba como resultado 3.0
```

El programa compila, pero no da los resultados esperados

Fig. 12. SIETTE item correction feedback.

Figure 12 shows the result after the evaluation has been completed. The execution of the student program fails. The script provided by the teacher is used to generate the feedback indicating the cases that do not satisfy the stem specifications. The student receives the feedback indicating the proofs that have not passed, indicating the expected result and the result



116531	-2.75	-2.25	-1.75	-1.25	-0.75	-0.25	0.25	0.75	1.25	1.75	2.25	2.75	Total
✓ 381766	0.000	0.000	0.094	0.120	0.261	0.000	0.227	0.500	0.828	0.845	0.975	1.000	**0.540**
✗ 381767	0.633	0.519	0.484	0.320	0.326	0.000	0.379	0.161	0.017	0.000	0.000	0.000	**0.206**
✗ 381768	0.367	0.444	0.406	0.500	0.348	0.000	0.348	0.274	0.121	0.060	0.012	0.000	**0.216**
✗ 381769	0.000	0.037	0.016	0.060	0.065	0.000	0.045	0.065	0.034	0.095	0.012	0.000	**0.038**

Fig. 13. The histogram shows the relative frequencies of each item option depending on the final assessment score.

given by the student code (Message: *The program compiles but does not give the expected results*).

2) Item performance: In this subject, no errors have been detected in the assessment proofs. However, SIETTE also provides useful information about the main problem faced by the students.

The assessment results are divided into 12 discrete categories within the range $[-3, +3]$. Figure 13 shows a histogram with relative frequencies of the item options that correspond to the different results of the first proof of a polynomials exercise. The four options are: (1) Correct (represented in green with ID=381766); (2) Incorrect submission, missing files (represented in red with ID=381767); (3) The program does not compile this proof, (represented in dark red with ID=381768); and (4) The program compiles but does not gives the expected results in all cases, (represented in dark red with ID=381769). The X-axis represents the final score obtained in the assessment that was included this task. It is clear that the relative frequency of the correct option increases with the final assessment score, while the others decrease.

V. Comparison with Other Systems

The main goal of automatic assessment systems is to improve students' programming skills, primarily in undergraduate basic programming courses. Students can complete a large number of exercises, which are evaluated quickly and get immediate feedback, that does not depend on the personal opinion of the academic staff. Since automatic assessment first appeared, a large number of tools and systems have been proposed. These systems have evolved from tools that use verified the output of a source code to complex systems that are integrated in Learning Management Systems (LMS) and allow the tracking of students throughout the course. As indicated in [5], new challenges have arisen in this field, which assessment systems have been incorporating: integration into LMS, obtaining detailed information on the process of improving students' skills, or the use of mobile applications to increase student interest, among others.

There are currently several on line systems for automatic assessment. These systems are used for automatic grading of student homework assignments and/or as automatic judges of programming competitions. They usually have an exercise repository that, once a solution has been delivered, gives a verdict on its correctness and functionality. Examples of these systems are Web-CAT [22], Gradescope [23], UVa Online Judge [24], or Judge.org [25]. The systems reviewed usually have different features and are intended for different languages and different students' and teachers' needs. None of these systems use assessment theories in their grading mechanisms.

SIETTE shares many of the features present in these systems, such as techniques or strategies for evaluating programs, scoring them automatically or integrating them into an LMS such as Moodle. The most significantly different feature of SIETTE is the way in which it deals with the final scoring and analysis results.

Tables IV and V show information on various automatic assessment systems using the comparison features proposed in the reviews of [3] and [5].

The columns in Table IV refer to the following characteristics of automatic systems.

- Authoring tools: Specifies if the system allows content to be developed by filling forms and uploading proof files, and/or assessment scripts.
- Supported languages: Programming languages for which the system is able to evaluate tasks. In the case that the system is not restricted to a given programming language, it is labeled as *Multilanguage*.
- Self-assessment: The student takes an evaluation to know his/her level, but the evaluation is not used as a grade by the teacher. This is also called *formative assessment* [32]. The main system goal in this case is to provide detailed feedback for the student to improve.
- Full-assessment: This is an evaluation that allows the teacher to decide whether or not the student has passed a certain level. Also referred to as *summative* assessment [32]. The main system goal in this case is to obtain an accurate score reflecting student knowledge.
- Work mode: The system can be a stand-alone application (*Standalone*) or integrated in an LMS (*Plug-in*). The two ways are not incompatible.
- Source program assessment procedures: Specifies which characteristics the system might evaluate in a task such as, compilation success; static code analysis; dynamic code analysis; code style checking; software metrics; comparison with a model solution, etc. If multiple source methods can be applied this column is labeled *Multiple*.
- Sand-boxing: Indicates whether the system executes tasks in a safe and independent environment, avoiding security problems.

Table V on revised automatic evaluation systems shows the principal characteristics related to the evaluation of the tasks. The columns have the following meanings:

- Categorical grading: The result of the evaluation is expressed as a value belonging to a set of categories. Various systems use the categories of the ACM International Collegiate Programming Contest and are shown in the table as ACM ICPC.
- Numerical grading: The evaluation result is a number within a range of numerical values.
- Aggregated score procedure: This column refers to the way in which the different features analyzed by the *source program assessment procedures* are aggregated to obtain a single numerical score. The options in this field are: (1) *Percentage*, that is, the number of successful proofs divided by the total number of proofs; (2) *Heuristic weighted* score, that is, each proof is assigned a weight in the final score according to the teacher's criteria; and (3) *IRT-based criteria*, based on a calibration process of proof results.
- Feedback: The minimum feedback is just to inform the student whether or not the program submitted compiles and passes the set of proofs and provides the correct solution. The next level is to indicate the score obtained by different proofs. The system can provide a detailed error report that, in some cases, can be reviewed manually by the teacher. Finally, the feedback can even include remedial hints or an animated visualization of the program's execution.
- Result analysis: Specifies whether the system performs any kind of analysis of the tests and tasks proposed in order to find the best (and worst) of them from the point of view of evaluating the student's knowledge in the subject.
- Manual assessment: Indicates whether it allows manual evaluation in addition to automatic evaluation.
- Plagiarism detection: Indicates if the system has a way to detect task plagiarism.

The automatic assessment system Jutge.org [25] exposes the problem of selecting the most relevant test case when evaluating a submission. The novelty of its approach is to analyze incorrect submissions and test cases using data mining techniques to discover the most relevant tests to find failures and then use them to evaluate future submissions. This raises the issue of finding the most relevant test cases or, in a broader sense, answering the question of what the most relevant tasks or questions to evaluate a student are. One way to answer this question is by the main contribution presented in our proposal. To the best our knowledge, what is not usually seen in other automatic assessment systems, is the use of psychometric techniques (such as IRT) to analyze and discover the questions or tasks that most efficiently discern a student's knowledge of the subject matter we wish to judge and, ultimately, score.

Another principal contribution of our system to the field is therefore represented by its systematization of the scoring procedure. Previous systems have addressed scoring using heuristic approaches only, or by leaving the scoring task to the teacher. For this reason, in most cases the automatic assessment of programing tasks is used just for computer home assignments or formative assignment, where the score does not play a relevant role. Scoring a programming task is difficult per se, even for manual assessment. There are two issues: (1)

TABLE IV
COMPARISON WITH OTHER SYSTEMS (MAIN FEATURES)

System	Authoring Tool	Supported Language	Self-Assessment	Full-Assessment	Source program Assessment Procedures	Work Mode	Sandboxing
Web-CAT [22]	Yes	Multilanguage	No	No	Static analysis Dynamic analysis Student test	Standalone	Yes
JAssess [26]	No	Java	No	No	Compilation success	Plug-in	No
CTpracticals [27]	Yes	VDHL	No	Yes	Comparison with behavior model	Plug-in	Yes
Mooshak [28]	Yes	Multilanguage	Yes(*)	No	Dynamic analysis	Standalone	Yes
ViLLe [29]	Yes	Multilanguage	No	Yes	Dynamic analysis, Comparison with model	Plug-in	No
KATTIS [30]	Yes	Multilanguage	Yes	Yes	Dynamic analysis	Standalone	Yes
BOSS [31]	Yes	Multilanguage	No	Yes	Multiple	Standalone	Yes
Jutge.org [25]	Yes	Multilanguage	No	Yes	Dynamic analysis; Syntactic analysis	Standalone	Yes
ProgTest [16]	Yes	Java, C	No	No	Dynamic Analysis	Standalone	No
SIETTE	Yes	Multilanguage	Yes	Yes	Multiple	Standalone Plug-in	Yes

TABLE V
COMPARISON WITH OTHER SYSTEMS (SCORING FEATURES)

System	Categorical Grading	Numerical Grading	Aggregated score Procedure	Feedback (Detailed Errors)	Result Analysis	Manual Assessment	Plagiarism Detection
Web-CAT [22]	ACM ICPC	No	N/A	Score with comments	Available dataset for research	Yes	No
JAssess [26]	Yes	No	N/A	No	No	Mandatory	No
CTpracticals [27]	Yes	Yes	Percentage	Error report Correct design	No	Optional	No
Mooshak [28]	ACM ICPC	No	N/A	Error report (revised)	No	Optional	No
ViLLe [29]	Yes	Yes	Percentage	Correctness, Visualization	No	No	No
KATTIS [30]	ACM ICPC	No	N/A	Correctness, CPU time used	No	No	No
BOSS [31]	No	Yes	Heuristic weighted	Manual	No	Optional	Yes
Jutge.org [25]	Yes	No	N/A	Error report	Data-mining	No	No
ProgTest [16]	No	Yes	Heuristic weighted	Error report	No	No	No
Siette [18]	Yes	Yes	Percentage Heuristic weighted IRT	Error report	Psychometric analyzes	Optional	Yes

setting the appropriate criteria for the correction, that is, decide what is going to be scored and what is the contribution of each part to the final score; and (2) determining if the criteria are satisfied by the student submission. Of course, both issues interact: a fuzzy criteria is harder to test.

VI. DISCUSSION

The experience using this framework has been positive. The continuously increasing number of users indicates that the system has proven useful. However, over the years certain problems have arisen. In this section, we will try to summarize our experience, the lessons learned, and the solutions we have applied.

The stem that contains the exercise specifications should be defined very carefully. The task to be done can be complex, but the output format of the exercise should be simple enough to be captured by a regular expression. The stem should describe the output clearly, even in the failed cases. One good idea is to provide a compiled version of the solution so that the students can check different test cases. The test cases can be divided into two groups:

1) public test cases, those given as examples with the stem.
2) private test cases.

There should always be private test cases in order to avoid cheating. If no private cases are used, there is always a strategy to produce the result simply by writing it to the output, without processing. Private test cases should always be kept secret if the exercise is repeated; otherwise, the evaluation reliability could be compromised.

Students sometimes commit minor errors due to carelessness or due to misunderstanding of the stem. Even though they can use different tools to validate their exercise before submitting (compilers, code checkers, and even a compiled version of the exercise proposed), they sometimes fail to notice some minor errors (like typos) that will produce an incorrect result in the automatic assessment tool. To solve this problem we have adopted different strategies:

1) Allow more than one submission of the same exercise; in fact, in our experience, the best choice is to allow unlimited submissions and take the highest score.
2) Include detailed feedback that points out the error so that the student can fix it and resubmit.
3) Optionally, it is possible to reassess submitted exercises. This feature has developed as a key feature of SIETTE.

Teachers sometimes commit errors when preparing the assessment, errors that are not noticed until after the test has been taken. Modifying the test cases and reassessing would therefore sometimes be useful. It may even be occasionally necessary to manually correct minor typos in the student code. To be able to do so, it is important to record all the source files submitted, not just the assessment results. Automatic evaluation is especially useful for the formative evaluation. It motivates the student to ”beat the machine,” and so they apply more time and effort to exercises, achieving higher scores. For high-stakes exams, students are always told that the score obtained at the end of the exercise is provisional. Some teachers even decide not to show the score at the end of the session. It is important to supervise the automatic assessment process, and to be able to automatically re-assess, or even manually, the code upon student demand.

Item independence is a desired feature to accomplish the assumptions of IRT. If an exercise is evaluated according to different perspectives, more independent items imply more information. This is a goal that cannot always be accomplished. In those cases, it is desirable that easiest items are evaluated first. If there is a dependency between items, and they are sorted in ascending difficulty order, then a partial credit model is a good solution. Otherwise, it is better to define multiple components underlying items. The two approaches are not incompatible. The first approach implies to use simple underlying items, with one component each, while the current trend is to use a single and longer exercise with independent test cases. The number of components also depends on the accuracy needed for the evaluation. It is not the same to take a simple ”pass/no pass” decision that assigns a qualitative note or a quantitative score for a ranking of students.

There is also a compromise between the amount of work required by human evaluation of exercises and the construction of automatically assessed exercises. Clearly if the number of students is small, the amount of work required from the teacher is much less. However, even if the student groups are small, one advantage of the automatic assessment is that the work can be re-used at any time. The creation of a bank of automatically assessed exercises is a future investment.

Using IRT as a grading mechanism has proved to have several advantages and a few drawbacks. Most of them are listed in the literature and are common to other types of assessments. In particular, for programming tasks, one advantage of using IRT is that there is no need to use heuristics to assign points to the different proofs. For instance, in the Compiler Construction programming tasks, where 30 or 40 proofs are triggered, the teacher does not have to define how many ”points” are assigned to each one. The system finds the item characteristic curves and applies those data for grading that increase reliability. On the other hand the calibration process is not fully automated, and requires initial data, so the first application of this assessment cannot provide results based on this technique. In this case, the methodology we follow is to provide an heuristic estimation based on the percentage of proofs passed by the first application, and then calibrate the items and change the scoring procedure to IRT. However, IRT is not applicable when the number of proofs (items) in the same assessment is small.

The system also offers the possibility to manually assess the tasks, fully or partially, using SIETTE just to deal with the submissions, and maybe to carry out some preliminary proofs. In this case, the teacher should only define the assessment criteria (called the rubric) but not the processing script. A teacher should review the submissions and manually assign them to one of the defined values in the rubric. The advantages in this case are that the evaluation criteria of programming exercises are clear but the previous amount of work needed to develop proofs is reduced to a minimum. On the other hand the teacher has to manually assess each exercise according to the rubric, and so the students cannot have immediate feedback. Although this feature is available, it has not yet been used.

VII. Conclusion

This article has presented a new technique to evaluate programming assignments. The key idea is to conceive the assessment of a programming assignment as a set of proofs that are treated as items of Classical Test Theory (CTT) or Item Response Theory (IRT) models. We have proven that this approach is feasible by implementing an extension of SIETTE to automatically assess and score programming assignments. We have constructed upon SIETTE a system that is able to deal with the whole process of authoring, submission, plagiarism detection, and grading of student assignments.

The system includes most of the features that other systems with the same objective have, and it has been designed for programming language independence (see Section V). The system supports everything from simple programming tasks, such as problems posed to beginners, to complex tasks, where the diagnosis of the program quality is based on several proofs. In fact, the system is just a framework that can support different types of proofs, from static to dynamic, and quality testing. It allows external general purpose software engineering tools to be used and/or the development of specific extensions to deal with the code analysis of a given language or a set of problems, like *JUnit* for Java or *QuickCheck* for Haskell. Sections IV-B1 and IV-C1 show the integration of these tools.

One of the main contributions of this new implementation is the use of well-founded assessment theories (CTT and IRT) for grading and result analysis. The SIETTE system core is based on these data-driven techniques (see Section III). This extension conceptualizes programming task evaluation based on different proofs, such as an evidence-based assessment, which makes it possible to apply the entire arsenal that has been developed in this field over the last few decades to improve the information and quality of the assessment of programming assignments (see Section VI). To the best of our knowledge, this is a new and original contribution in the area of programming assessment.

By applying these theories, validity and reliability of assessments can be measured. The aim of this article has not been to demonstrate that almost all the assessments developed within this platform are valid and reliable, although this is the case, but to show that it is something that can and should be included in the assessment of programming tasks. This is especially important for high-stakes assessment. See the validity studies in Section IV-A1.

Another important opportunity offered by these techniques, and implemented in SIETTE, is the possibility to carry out the analysis of assessment results and item behavior (see Sections IV-A3; IV-B1 and IV-C1). These analyzes are important to not only gain an overall idea of the population distribution, that is, if the students are passing certain proofs or not, but can also indicate potential problems with certain proofs that should be removed.

The experience over these last few years has been very positive, which is proven by the continuously increasing number of users. During this time many technical and usability issues have been improved in response to users' opinions, both teachers and students. The system is available at https://www.siette.org, but most of the content of the programming tasks is currently restricted to the Virtual Campus of the University of Malaga (VC-UMA). We are working to develop content that can be shared worldwide and integrated with other systems.

References

[1] V. Pieterse, "Automated assessment of programming assignments," in *Proc. CSERC 2013*. Open Universiteit, Heerlen, 2013, pp. 4:45–4:56.

[2] C. Douce, D. Livingstone, and J. Orwell, "Automatic test-based assessment of programming: A review," *J. Educ. Resour. Comput.*, vol. 5, no. 3, Sep. 2005.

[3] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in *Proc. 10th Koli Calling International Conference on Computing Education Research*, ser. Koli Calling '10. New York, NY, USA: ACM, 2010, pp. 86–93.

[4] R. Romli, S. Sulaiman, and K. Z. Z amli, "Automatic programming assessment and test data generation a review on its approaches," in *Proc. Information Technology (ITSim)*, vol. 3. IEEE, 2010, pp. 1186–1192.

[5] J. C. Caiza and J. M. Del Alamo, "Programming assignments automatic grading: review of tools and implementations," in *INTED2013*, 2013, pp. 5691–5700.

[6] R. J. Mislevy, L. S. Steinberg, and R. G. Almond, "Focus article: On the structure of educational assessments," *Measurement: Interdisciplinary research and perspectives*, vol. 1, no. 1, pp. 3–62, 2003.

[7] R. J. Mislevy and M. M. Riconscente, "Evidence-centered assessment design," *Handbook of test development*, pp. 61–90, 2006.

[8] M. J. Zieky, "An introduction to the use of evidence-centered design in test development," *Psicologia Educativa*, vol. 20, no. 2, pp. 79–87, 2014.

[9] S. Ohlsson, *Constraint-Based Student Modeling*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 167–189.

[10] J. Galvez, E. Guzman, R. Conejo, A. Mitrovic, and M. Mathews, "Data calibration for statistical-based assessment in constraint-based tutors," *Knowledge-Based Systems*, vol. 97, no. Supplement C, pp. 11 – 23, 2016.

[11] J. Galvez, E. Guzman, and R. Conejo, *Exploring Quality of Constraints for Assessment in Problem Solving Environments*. Berlin: Springer, 2012, pp. 310–319.

[12] J. Galvez, E. Guzman, R. Conejo, and E. Millan, "Student knowledge diagnosis using item response theory and constraint-based modeling," in *Proc. AIED 2009*. The Netherlands: IOS Press, 2009, pp. 291–298.

[13] T. Wang, X. Su, Y. Wang, and P. Ma, "Semantic similarity-based grading of student programs," *Inf. Softw. Technol.*, vol. 49, no. 2, pp. 99–107, Feb. 2007.

[14] K. A. Naude, J. H. Greyling, and D. Vogts, "Marking student programs using graph similarity," *Computers & Education*, vol. 54, no. 2, pp. 545 – 561, 2010.

[15] M. Vujošević-Janičić, M. Nikolić, D. Tošić, and V. Kuncak, "Software verification and graph similarity for automated evaluation of students' assignments," *Inf. Softw. Technol.*, vol. 55, no. 6, pp. 1004–1016, Jun. 2013.

[16] D. M. d. Souza, S. Isotani, and E. F. Barbosa, "Teaching novice programmers using progtest," *IJKL*, vol. 10, no. 1, pp. 60–77, 2015.

[17] R. K. Hambleton and H. Swaminathan, *Item Response Theory*. Dordrecht: Springer Netherlands, 1985.

[18] R. Conejo, E. Guzmán, E. Millán, M. Trella, J. L. Pérez-De-La-Cruz, and A. Ríos, "Siette: A web-based tool for adaptive testing," *International Journal of Artificial Intelligence in Education*, vol. 14, no. 1, pp. 29–61, 2004.

[19] R. Conejo, E. Guzmán, and M. Trella, "The SIETTE automatic assessment environment," *I. J. Artificial Intelligence in Education*, vol. 26, no. 1, pp. 270–292, 2016.

[20] I. Arroyo, R. Conejo, E. Guzman, and B. P. Woolf, "An adaptive web-based component for cognitive ability estimation," in *Proc. AI-ED*, 2001, pp. 456–466.

[21] D. Thiessen, "Multilog user's guide, version 6," *Chicago: Scientific Software International*, 1991.

[22] "Web-CAT – the web based center for automated testing," 2003, accessed: 2013-08-11. [Online]. Available: http://www.web-cat.org

[23] A. Singh, S. Karayev, K. Gutowski, and P. Abbeel, "Gradescope: A fast, flexible, and fair system for scalable assessment of handwritten work," in *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, ser. L@S '17. New York, NY, USA: ACM, 2017, pp. 81–88. [Online]. Available: http://doi.acm.org/10.1145/3051457.3051466

[24] M. A. Revilla, S. Manzoor, and R. Liu, "Competitive learning in informatics: The uva online judge experience," *Olympiads in Informatics*, vol. 2, pp. 131–148, 2008.

[25] J. Petit, S. Roura, J. Carmona, J. Cortadella, A. Duch, O. Gimenez, A. Mani, J. Mas, E. Rodriguez-Carbonella, A. Rubio, J. de San Pedro, and V. Divya, "Jutge.org: Characteristics and experiences," *IEEE Transactions on Learning Technologies*, no. 99, pp. 1–1, 2017.

[26] N. Yusof, N. A. M. Zin, and N. S. Adnan, "Java programming assessment tool for assignment module in moodle e-learning system," *Procedia - Social and Behavioral Sciences*, vol. 56, no. Supplement C, pp. 767 – 773, 2012.

[27] E. Gutiérrez, M. A. Trenas, J. Ramos, F. Corbera, and S. Romero, "A new moodle module supporting automatic verification of vhdl-based assignments," *Comput. Educ.*, vol. 54, no. 2, pp. 562–577, Feb. 2010.

[28] J. Leal and F. Silva, "Mooshak: A web-based multi-site programming contest system," vol. 33, pp. 567–581, 05 2003.

[29] V. Karavirta, R. Haavisto, E. Kaila, M.-J. Laakso, T. Rajala, and T. Salakoski, "Interactive learning content for introductory computer science course using the ville exercise framework," in *2015 Int. Conf. on Learning and*

Teaching in Computing and Engineering, April 2015, pp. 9–16.

[30] E. Enstrom, G. Kreitz, F. Niemela, P. Soderman, and V. Kann, "Five years with kattis – using an automated assessment system in teaching," in *FIE 2011*, Oct 2011, pp. T3J–1–T3J–6.

[31] M. Joy, N. Griffiths, and R. Boyatt, "The boss online submission and assessment system," *J. Educ. Resour. Comput.*, vol. 5, no. 3, Sep. 2005.

[32] W. Harlen and M. James, "Assessment and learning: differences and relationships between formative and summative assessment," *Assessment in Education: Principles, Policy & practice*, vol. 4, no. 3, pp. 365–379, 1997.

VIII. ANNEX. List of Abbreviation

CBM - Constraint Based Modeling.
CAT - Computer Adaptive Testing.
CTT - Classical Text Theory
ECD - Evidence-Centered Design
GRM - Graded Response Model.
ICC - Item Characteristic Curve.
IIF - Item Information Function.
IRT - Item response Theory.
ITS - Intelligent Tutoring Systems
JICS - Java Item Calibration System.
LMS - Learning Management Systems.
SFIP - SIETTE File-Item Player.
SPS - SIETTE Processing Script.
TIF - Test Information Function.
VC-UMA - Virtual Campus of the University of Malaga

Ricardo Conejo (MS & PhD in Ingeniero de Caminos, Canales y Puertos, UPM, Madrid) holds the position of Professor (Full) in the Languages and Computer Science Department of the University of Malaga, Spain, where he has worked since 1986. He has taught Compilers and Programming for more than 30 years. His research interests currently focus on adaptive testing, student knowledge diagnosis, and intelligent tutoring systems. He has also worked on fuzzy logic, model-based diagnosis, multi-agent systems, and artificial intelligence applied to civil engineering. Prof. Conejo is a regular member of program committees of international conferences, such as *User Modeling Adaptation, and Personalization* (UMAP), *Intelligent Tutoring Systems* (ITS), and *Artificial Intelligence in Education* (AIED) and is an associate editor of *IEEE Transaction on Learning Technologies*.

Beatriz Barros (MS, Computer Science, PhD, Artificial Intelligence, UPM, Madrid) holds the position of Professor (Full) in the University of Malaga. She teaches Programming in several courses in Computer and Engineering degree programs. Her research interests include the design of adaptive and interactive learning environments, and collaborative learning systems.

Manuel F. Bertoa holds a degree in Telecommunications Engineering from UPM, Madrid and a PhD from the University of Malaga. For 16 years, his professional career has been mainly in the private sector in several telecommunications and computer companies but also includes work in public administration. His research activity focuses on software quality and, currently, LMS.