View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Dagstuhl Research Online Publication Server

A Flexible Dynamic System for Automatic Grading of Programming Exercises

Daniela Fonte[1], Daniela da Cruz[1], Alda Lopes Gançarski[2], and Pedro Rangel Henriques[1]

1 Department of Informatics, University of Minho
Braga, Portugal
{danielamoraisfonte,danieladacruz,pedrorangelhenriques}@gmail.com

2 Institute Telecom, Telecom SudParis
Paris, France
alda.gancarski@telecom-sudparis.eu

Abstract

The research on programs capable to automatically grade source code has been a subject of great interest to many researchers. Automatic Grading Systems (AGS) were born to support programming courses and gained popularity due to their ability to assess, evaluate, grade and manage the students' programming exercises, saving teachers from this manual task.

This paper discusses semantic analysis techniques, and how they can be applied to improve the validation and assessment process of an AGS. We believe that the more flexible is the results assessment, the more precise is the source code grading, and better feedback is provided (improving the students learning process).

In this paper, we introduce a generic model to obtain a more flexible and fair 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 a program under assessment with the expected output at a semantic level. To implement our model, we propose a Flexible Dynamic Analyzer, able to perform a semantic-similarity analysis based on our Output Semantic-Similarity Language (OSSL) that, besides specifying the output structure, allows to define how to mark partially correct answers. Our proposal is compliant with the Learning Objects standard.

1998 ACM Subject Classification K.3.2 Computer and Information Science Education, D.3.1 Programming Languages - Formal Definitions and Theory

Keywords and phrases Automatic Grading Systems, Domain Specific Languages, DSL, Dynamic Analysis

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.129

1 Introduction

When learning a new programming language, students need to solve a large number of programming exercises to practice the new language syntax and semantics. Teacher's feedback about the mistakes that they made on those exercises is crucial to improve their knowledge. However, it is hard for teachers to manually manage all the students' solutions.

The manual grading of programming exercises can involve a lot of work and be a time consuming task, since each program must be tested and its source code must be analyzed by a teacher. This task is neither simple nor mechanical: it is often a complex and arduous process, prone to faults. Different human graders may assign different evaluations to the same exercise, due to several factors like fatigue, favoritism or even inconsistency [51].

© Daniela Fonte, Daniela da Cruz, Alda Lopes Gançarski and Pedro Rangel Henriques;
licensed under Creative Commons License CC-BY

2^{nd} Symposium on Languages, Applications and Technologies (SLATE'13).
Editors: José Paulo Leal, Ricardo Rocha, Alberto Simões; pp. 129–144
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

To minimize such problems, the research on tools capable to reduce the amount of work for instructors and improve the students learning experience led to the development of several *Automatic Grading Systems* (AGS), specialized on grading student's programs; they gained popularity in the field of teaching and learning programming languages [2] as Learning Support tools.

Aside their educational role, AGS are also used by the programming communities for *programming contests*. These contests can vary slightly in rules, but all of them are intended to assess the competitor skills concerning the ability to solve problems using a computer.

In a typical programming contest competitors participate in teams to solve a set of problems. For each problem, the team submits the source code of the program developed to solve the problem. Many well known programming contests in the world — such as ACM-ICPC[1] — are based on the automatic grading of the proposed solutions. This means that the submitted code will be immediately evaluated by an AGS. This process normally involves tasks like running the program over a set of predefined tests (actually a set of input data vectors), and comparing each result (the actual output produced by the submitted code) against the expected output value. Time and memory space consumptions are also usually measured during the program execution and taken into account in the final grade. This evaluation is typically complemented by the action of a human judge, who takes the final grade decision according to the specific rules for each contest.

1.1 Automatic Grading Systems as Competitive Learning Tools

Programming contests gained popularity in programming courses as a competitive learning tool in the form of exercises to stimulate the students' ability to solve practical problems in a competitive environment. An example of this was born with Mooshak[2], a system originally developed for managing programming contests [24]. Mooshak is at moment used as an e-learning tool in several universities in programming courses and is a reference tool for competitive learning in Portugal. In [26], the authors present an overview of this experience, evidencing the characteristics of competitive learning that stimulates students to work harder on problem solving using the subjects taught in each course. Students participate in several "contests" where they have to solve one or more problems, receive immediate feedback on their attempts and are able to compare their own progress with the progress of their colleagues. By providing immediate feedback to students, they are encouraged to improve their skills and to submit a new solution. The challenge associated with these competitive environments provides a meaningful way to learn and easily acquire practical skills on programming.

1.2 Assessment methodologies: Static versus Dynamic Analysis

AGS are classified concerning the methodology followed to evaluate the submitted program. This assessment can be done using two different techniques: *static* or *dynamic*.

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. Unlike dynamic analysis, this method is able to gather information about the source code without executing it.

[1] International Collegiate Programming Contest: `http://cm.baylor.edu/welcome.icpc`

[2] `http://mooshak.dcc.fc.up.pt/`

In Section 2 we detail each approach and present an overview of the existing systems that fall in each type.

1.3 Automatic Grading of Partially Correct Answers

As discussed in Section 1.2, existing dynamic-based AGS run the submitted program using a set of predefined tests and compare its output with the expected output. However, this comparison is sometimes done using a simple string comparison between the outputs, which does not allow formatting differences between them. A simple example of this situation may be a program that lists all the possible subsets of a giving set. If a specific order is not explicitly asked, different valid listing options can appear; and, of course, the formatting of the subsets can vary slightly on spacing and punctuation. However, such AGS do not support any variant of the expected output vector.

A possible solution, followed by some AGS (such as Mooshak [24]), is to allow the manual codification of a script to validate output differences. This option consumes time and effort; moreover the insertion of a new test in the test set may forces a change in that script, if it does not include all the valid options. This enables a flexibilization of the comparison process, but it is can be restricted to the programming languages supported by the AGS or by the host environment.

Consider now a situation where a student is asked to codify a program that outputs the divisors list of a given number (in ascending order). A submitted program may actually output the correct divisors, but not respect the asked order or even not output the complete list. In classrooms, a manual grading of these answers should consider them partially correct, allowing a score based on the severity level of the errors found. However, traditional AGS only grade these answers if they respect the structure defined on the comparison script — they do not support the individual evaluation of the partially correct answers.

The notion of grading partially correct answers is explored in [52], but only focused on submitted programs with *syntax* errors. In this work, the authors propose an automatic grading algorithm that combines dynamic and static marking, based on compiler theory and matching of knowledge points [52], capable to grade programs with syntax errors. However, based on the surveyed research work, there is no grading system capable of assessing programs whose produced output has *semantic* errors, regarding the expected output.

1.4 Our Contribution

We propose an output semantic-similarity based analysis that allows the comparison between the meaning of the actually produced output and the meaning of the expected output. Moreover, we aim to allow not only the specification of which parts of the generated output can differ from the expected output, but also to define how to mark partially correct answers.

More specifically, we intend to extend the traditional dynamic analysis concept by exploiting the use of a Domain Specific Language (DSL) for an output structure specification. This leads to a more flexible and fair grading process, closer to a manual one, by not restricting the output comparison process. To this end, we explore how to enrich similarity-based techniques with semantic annotations, in order to specify rules about how the outputs should be given and compared.

1.5 Article Structure

This document is organized as follows. Section 2 surveys the related work in AGS and presents the state of the art, describing their evolution in terms of the techniques applied to assess the submitted program. Section 3 is devoted to the exposition of our proposal. Section 4 closes the document with some conclusions and directions for future work.

2 Automatic Grading Systems for Program Evaluation

The earliest report about systems capable to automatically grade programs was published in 1960 in CACM by Hollingsworth [15], describing a "grader program" used to assess students in machine language at Rensselaer. This grader was completely automatic and did not require user special intervention or knowledge.

In 1965, Forsythe [11] introduced a system that follows the fundamental principle of the modern grading systems by validating the submitted solutions with a set of tests.

In 1969, BAGS [14], a system developed at University of Sydney, was used to test the submitted programs with two data sets. The system gives points for each of five activities: successful compile, complete run, data set 1 correct, data set 2 correct, and time sufficiently short. The program penalizes each extra submission after the first attempt.

Later, in 1988, Ceilidh [4] was the first computer-based assessment coursework system. Its first release only supports C language but, in 1992, a major release that supports C and C++ languages became available to all educational institutions. This version could be accessed either via a command line interface or a text based terminal menu interface. From its implementation in 1988, it had an important impact on the research and implementation of related grading systems, including CourseMarker [12], which is its direct descendant.

Kassandra [47] was developed in 1994 at ETH Zurich. It was designed to automatically mark Maple and Matlab exercises, implemented as a network service. After students submit their programs, Kassandra tests them according to two test cases and gives credit if both answers are correct. It also provides students with a complete assessment report.

With the evolution of computers, AGS increased in complexity, diversifying the tests made to the subject programs and introducing tools for monitoring the grading process. As referred, they can be distinguished according to the approach followed to evaluate the submitted program. This assessment can be done employing two major different techniques: *static* and *dynamic*. Next subsections are devoted to the analysis of this two approaches; also a more recent hybrid technique is introduced.

2.1 Dynamic Assessment

Dynamic approaches focus on the execution of the program through a set of predefined tests, comparing the generated output with the expected output (provided in the set of tests). It is the most obvious approach to verify the program correctness.

There are in the literature many systems that adopt this approach, such as Ceilidh [4], BAGS [14], TRY [40], Kassandra [47], PSGE [22], HoGG [33], Mooshak [25], JEWL [9], Quiver [8], Infandango [17], and the tools referred in [11, 15], among many others.

Some of the dynamic-based systems, such as Better Programmer[3], were developed as Web-based submission tools, where users can exercise and evaluate their programming skills by picking-up a problem from their repository, coding a solution and submitting

[3] http://www.betterprogrammer.com

it for assessment. This concept is also largely explored by several universities over the world to support automatic grading of their programming courses, encouraging the so called Competitive Learning (CL). Examples of CL projects are Practice-It[4], Marmoset[5], CodingBat[6], UVa Online Judge[7], CodeLab[8] and CodeWrite[9], among others.

Dynamic approaches are usually based in a simple string comparison between the expected output and the output actually produced to determine if both values are equal. Thus, the submitted program is considered correct if and only if this condition is true, which can be a limitation. Besides this, another important drawback of this approach is the incapability to produce an assessment when a program does not successfully compile, does not produce an output or does not end its execution (infinite loop).

2.2 Static Assessment

Unlike *Dynamic approaches*, which are incapable of analyzing the way source code is written, *Static approaches* take benefit from the technology developed for compilers and language-based tools, to gather information about the programming code without executing it. They are supported by source code analysis, which allows to detect situations where the submitted solution does not comply with the exercise rules.

As an example, consider a typical C programming exercise that asks the student to implement a Graph using *adjacency lists* to print the shortest-path between two given nodes. If the final output is equal to the expected one, Dynamic AGS will consider correct a solution implemented with an *adjacency matrix*. However, this solution is not acceptable because it does not satisfy all the assignment requirements – a static approach can be useful to help detecting the used data types. Or, even more dramatic, if the user computes by hand the shortest-path and the submitted program only prints it, the solution is also accepted using a dynamic approach, because it is not able detect such implementation faults.

The most popular method used in this approach is based on software metrics analysis. Metrics, such as lines of code, number of variables, statements and expressions or even the code complexity are used as the program grading base. They are easy to calculate, though the semantics of a program can not be analyzed. Examples of such systems are STYLE [23], Knots [18], CAP [43], Style Checker [32] and Verilog Logiscope [30].

Besides software metrics, there are other techniques that fit on static analysis approach such as the programming style assessment [1], syntax and semantics errors detection [45, 16], structural similarity analysis [3, 46, 34, 49, 51], non-structural similarity analysis [50], keyword detection [42] or even plagiarism detection [35], allowing static analysis to be a powerful approach to evaluate how well source code is written.

2.3 Hybrid Assessment

Static approaches can not be used for testing the correctness of programs with input and output operations. However, traditional dynamic grading systems leave aside one important aspect when assessing programming skills: the source code quality. These assumptions

[4] http://Webster.cs.washington.edu:8080/practiceit/
[5] http://marmoset.cs.umd.edu/
[6] http://codingbat.com/
[7] http://uva.onlinejudge.org
[8] http://turingscraft.com/
[9] http://codewrite.cs.auckland.ac.nz/

led to the construction of systems such as Web-CAT [44, 7], WebBot [5], Scheme-Robe [42] or BOSS[10], that combine the best of both approaches, by improving the dynamic testing mechanism with static techniques like metrics or style analysis. This symbiosis keeps providing immediate feedback to the users (students/competitors or instructors/judges), but enriched by a quality analysis – which is obviously a relevant extra-value.

The first system combining both approaches was Ceilidh [13] by introducing semantic error detection. It statically detects suspicious never-ending loops, a crucial feature to avoid breaks during the dynamic evaluation process. This system completes the dynamic analysis with a static verification of the program layout and structure – its indentation, identifiers, comments; it also measures readability and complexity metrics.

Another example is ASSYST [20], used to automate some aspects of grading in introductory Ada classes, as well as a second-year C-programming course. It gives weighted grades to students, based on the correctness (output), efficiency (run time), style and complexity of their answers, and also based on the adequacy of the submitted tests (student self-test data).

Used in Java introductory programming courses, eGrader [3] produces detailed feedback reports, showing to students the model solution provided by the teacher. Additionally, specific comments on syntax and semantic errors (if any) are also provided. Its static analysis process consists of two parts: the structural similarity, which is based on the graph representation of the program; and the quality analysis, which is measured by software metrics.

A more recent system, AutoLEP [48], improves the traditional static grading mechanisms with dynamic code testing, by enriching source code static analysis with a comparison of the similarity degree. Summing up, it evaluates the program construction and how close the source code is from the correct solution.

Another example of hybrid assessment is Quimera [10], a Web-based application able to evaluate source code written in C language, which provides a full management system for programming contests. Quimera allows to create and manage programming exercises both in competitive learning and programming contest environments. Besides the traditional dynamic approach, this system provides a static analysis of the program by measuring the source code quality. Thus, the final grade is based not only in the source code capability of producing the expected output, but also on its quality and accuracy.

3 Flexible Dynamic Analysis

Our proposal, a Flexible Dynamic Analyzer (FDA), is based on the traditional dynamic analysis which is, as referred, supported by the execution of the submitted answer (the program under assessment) over a set of predefined tests. We aim at extending this concept to allow a more flexible comparison between the output produced by the program under assessment and the expected output.

We propose to compare the meaning of both outputs performing a semantic-similarity analysis to achieve a more flexible grading process. A Domain Specific Language (DSL), specially designed to specify the output structure and semantics, will be used as the basis for the desired semantic comparison. The DSL's design will also support the mark of partially correct answers.

Next subsections are devoted to detail this proposal, a flexible system able to interpret the output meaning, concerning a predefined structure. This system produces a complete grading report, designed to be easily integrated with a traditional dynamic analyzer. Section 3.1

[10] `http://www.dcs.warwick.ac.uk/boss/about.php`

presents the proposed architecture. Section 3.2 introduces the Output Semantic-Similarity Language (OSSL), the proposed DSL used by the FDA. OSSL grammar is also presented and discussed; the subsection ends with illustrative examples.

3.1 Architecture

Our FDA was designed as an independent module that can be easily integrated with any AGS that uses an external evaluator. We just require that the AGS compiles the submitted source code, and provides a compilation report and (if the compilation is successful) the compiled program.

Moreover, to follow the trend of the evolution of systems that perform the automatic assessment of programming exercises [6], as well as to ensure the interoperability with other systems [36], the FDA uses the concept of Learning Object (LO) to describe a programming exercise, its assessment instructions and the associated resources. LOs are content components, organized in context independent and reusable digital pieces of information – a standard in the learning domain [37]. Since the standard LO cannot be used for complex evaluation domains such as programming exercise evaluation [28], we propose an extension of the Programming Exercises Interoperability Language (PExIL) [38] to include the OSSL description of the set of tests. PExIL aims at consolidating all the data required to cover the programming exercise life-cycle, since it is created until it is graded. The associated PexilUtils generator produces a IMS CC[11] LO package, allowing the definition of Specialized Learning Objects. The use of LOs components also ensures compatibility with specialized LOs Repositories such as CrimsonHex [27], allowing the reuse of programming exercises among different systems.

Therefore, the FDA architecture is composed of three modules: the OSSL Processor, the Flexible Evaluator and the Grader, as depicted in Figure 1.

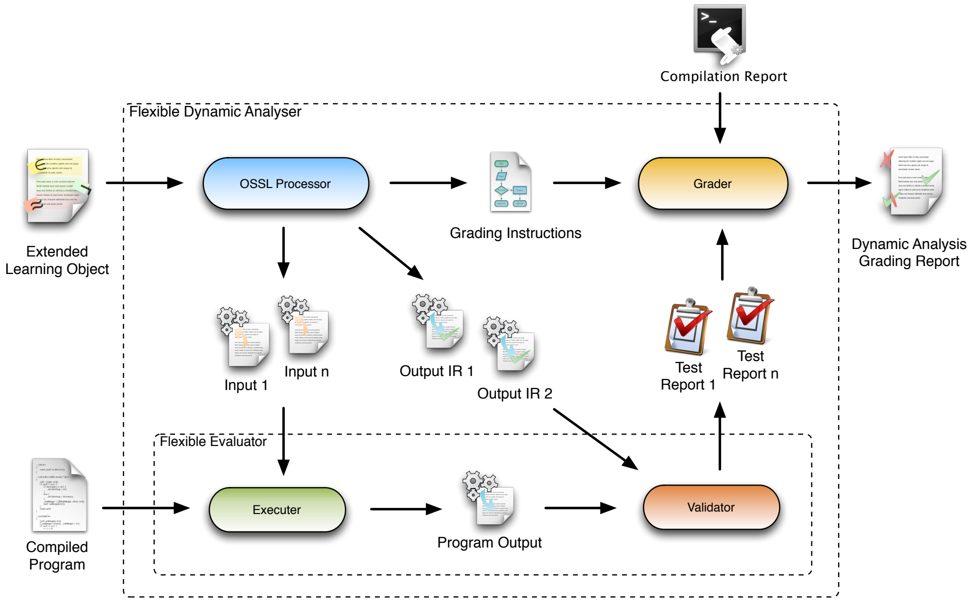

Figure 1 Flexible Dynamic Analyzer Architecture.

[11] A package standard that assembles educational resources and publishes them as reusable packages.

The OSSL Processor is the central piece of the FDA, being the 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) and the Grading Instructions.

The Flexible Evaluator is responsible for the execution and validation of 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 intermediate representation 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.

3.2 OSSL: Output Semantic-Similarity Language

We believe that the development of a DSL is the most flexible approach for: (i) the extension of an AGS dynamic analyzer to interpret different output values with the same meaning; and (ii) to support partial grading. DSLs are programming languages adapted to a specific application domain, which offer substantial gains in expressiveness and ease of use, when compared with general-purpose programming languages [31]. Rather than being for a general purpose, a DSL captures the *semantics* of its domain. Examples of DSLs include *lex* [29] and *yacc* [21], used for program lexical analysis and parsing, HTML [39], used for document markup, or even VHDL [19], used for hardware descriptions. Concerning the DSLs scope and our goals, we propose a DSL that, given a programming exercise, allows to define:

- The program output meaning;
- Partially correct solutions and their penalties;
- Mandatory and optional output components;
- The support of case sensitive text;
- The delimiter and punctuation characters used to produce the output;
- Output patterns.

As discussed along Section 1, the grade of programing exercises is a complex task that usually involves executing a program to assess its ability to produce the expected output concerning the given input. The OSSL language aims at supporting the output structure specification, in order to allow an easy and clear way of describing the instructions for the automatic grading of the output produced by the program under evaluation. This allows the interpretation of the output meaning and perform its grading, based on a semantic-similarity specification strategy. This description follows the traditional manual assessment process, determining partially correct answers and their respective grade.

To be able to automatically interpret the meaning of a program output, it is essential to categorize in a simple but formal way the format used to model that output. To represent the abstraction of the possible output values, we divide them into two main types: the *atomic* and the *compound* values. The *Atomic* category includes numeric, character, identifier and string values. The other category – *Compound values* – represents tuples, unions, sets, sequences, trees, graphs and mappings. Concerning this categorization, we propose the grammar in Listing 1.

Listing 1 OSSL Grammar Core.

```
Output   -> Value
Value    -> Atomic   |   Compound
Atomic   -> number | character | identifier | string
Compound -> Tuple | Union | Set | Seq | Mapping | Tree | Graph
```

An *Output* is defined by its *Value*, which can be an *Atomic* or a *Compound* value. As referred, *Atomic* values can represent *numbers*, *characters*, *identifier* or *strings*. *Tuples*, *Unions*, *Sets* and *Sequences* are represented by lists of elements that are *Values*. A *Mapping* is composed of a *Key* and an associated *Value*. *Trees* are represented by their *Root*, composed of an *Atomic* value and its *Descendants*. *Graphs* are represented by a list of arcs, being each arc a triple composed of source and destination *Nodes* and its *Weight*. Since this is a recursive definition, the elements of a *Compound* value can be atomic or compound.

In order to allow the specification of partially correct answers and also the association of a grade, the grammar axiom *Output* was redefined as can be seen in Listing 2.

Listing 2 OSSL Grammar extension for support partially correct answers.

```
Output -> Correct PartiallyCorrect
Correct -> Value Grade
PartiallyCorrect -> (Value Percent) *
```

An *Output* is composed of a *Correct* answer and a list of *Partially Correct* answers (if any). *Correct* answers have an associated *Grade*. This grade is the base of the *Partially Correct* answers assessment, which is a *Percent* of the correct answer grade.

To enable the automatization of each test, a new axiom was defined, including the specification of the input value, as can be seen in Listing 3.

Listing 3 OSSL Grammar support for test automatization.

```
Test -> Input Output
Input -> Atomic +
```

In order to permit the definition of all the tests in the same specification, the grammar is extended again with a new axiom and a new production, shown in Listing 4.

Listing 4 OSSL Grammar support for a set of tests.

```
TestSet -> Test +
```

To define completely OSSL, the abstract grammar presented above shall be transform into a concrete one, adding some syntactic sugar. Listing 5 shows our final choice. In the final version of the grammar, a new axiom was introduced, *Ossl*, composed of two elements. The *Header*, that allows to identify the problem, define the number of tests included and the total grading of the set of tests. The *TestSet* represents all the tests (the pairs of input/output descriptions). Notice that the sum of the grade corresponding to each output must be equal to the *Total* value defined in the *Header*.

Listing 5 OSSL Grammar.

```
Ossl -> Header TestSet
Header -> PROBLEM ":" identifier TESTS ":" number TOTAL ":" number
TestSet  -> Test +
Test     -> Input Output
Input    -> INPUT ":" Value
Output   -> OUTPUT ":" Correct PartiallyCorrect
Correct  -> Value "(" Grade ")"
PartiallyCorrect -> ( ALSO Value "(" Percent ")" )*
Value    -> Atomic | Compound
Atomic   -> number | character | identifier | string
Compound -> TUPLE Elems | Pair
            | UNION Elems | MAP Entries
            | SET Elems | SEQ  Elems
            | TREE Root | GRAPH Arc +
Elems    -> "<" Values ">"
Entries  -> Entry +
Entry    -> Key "->" Value
Values   -> Value ("," Value)*
Key      -> Atomic
Root     -> Node Descs
Node     -> Atomic
Descs    -> Root *
Percent  -> number
Grade    -> number
Arc      -> "(" Node "," Node Weight ")"
Weight   -> & | "," Atomic
Pair     -> "(" Value "," Value ")"
```

Let us now introduce some examples of OSSL grammar usage, presenting the OSSL specification for two simple programming exercises.

Example 1

Consider the following problem statement: *Given a positive integer, compute:*

a) *The sequence of its divisors, in ascending order;*
b) *The set of its divisors.*

Consider now that the set of tests defined for the proposed problem statement is composed of two tests, the first one with the number 10 as input, and the second one with the number 18. The correct divisors are 1, 2, 5 and 10 for the first test, and 1, 2, 3, 6, 9 and 18 for the second test. Concerning question a), the correct answer would be the sequence of the respective divisors. Using OSSL language, the set of tests and their corresponding grades would be defined as described in Listing 6.

When comparing the expected output with the effectively produced one, the FDA compares each value according to the defined order. Thereby, this specification ensures that only a sequence of the correspondent divisors will be considered a correct answer. For instance, the sequence <1, 2, 10, 5> is not accepted as a correct answer.

Consider now the question b). The set of tests is defined in OSSL language as described in Listing 7.

Listing 6 OSSL definition for question 1b).

```
PROBLEM: SeqDivisors TESTS: 2 TOTAL:3

INPUT: 10
OUTPUT: SEQ <1,2,5,10>   (1)

INPUT: 18
OUTPUT: SEQ <1,2,3,6,9,18> (2)
```

Listing 7 OSSL definition for the concerning question.

```
PROBLEM: SetDivisors TESTS: 2 TOTAL:3

INPUT: 10
OUTPUT: SET <1,2,5,10>   (1)

INPUT: 18
OUTPUT: SET <1,2,3,6,9,18> (2)
```

This definition ensures that any combination of the specified numbers is considered a correct answer – the FDA will verify, for each value of the produced output, if it is a member of the accepted set. So, <3, 18, 9, 2, 6, 1> is one of the possible correct answers for the second test (input 18).

Consider again question a). Listing 8 illustrates how to specify that incomplete sequences, missing their extreme values, should be accept as partially correct answers.

Listing 8 OSSL definition with partially correct answers.

```
PROBLEM: SetDivisors TESTS: 2 TOTAL:3

INPUT: 10
OUTPUT: SEQ <1,2,5,10>   (1)
        ALSO SEQ <2,5,10> (0.5)
        ALSO SEQ <1,2,5> (0.5)

INPUT: 18
OUTPUT: SEQ <1,2,3,6,9,18> (2)
        ALSO SEQ <2,3,6,9,18> (0.5)
        ALSO SEQ <1,2,3,6,9> (0.5)
```

The OSSL specification in Listing 8 allows to accept answers where the first or the last value of the correct sequence is not outputted. In such situations, the final grade will be 50% of the total grade.

Example 2

Consider now the following problem statement: *Write a program that allows to find all the possible paths to solve a given maze. A maze is represented through a 6 x 5 matrix, where the Lines are numbered from 1 to 6 and the Columns are identified from A to E, as depicted in Figure 2. Each cell represents a Position in the maze, expressed by a pair (Line, Column) (e.g., (1,A) represents the first cell of the maze, on its left top corner).*

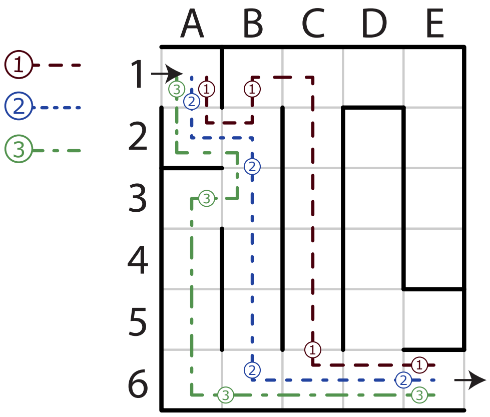

Figure 2 The three possible paths that solve the maze.

The maze Walls are represented by a list of Coordinates indicating the positions where each wall is in the maze. Each Coordinate is represented by a tuple (Position, Limit), where the Limit represents the side of the cell where the wall is located. The different positions are represented according to the following notation convention: Left (L), Top (T), Right (R) and Bottom (B) (e.g., ((1,A),R) represents a wall on the right side of cell (1,A)).

The program receives three parameters: the start Position, the end Position, and the Walls list, and will output the set of the correspondent possible paths.

Listing 9 represents the OSSL specification for the given problem, concerning the maze definitions depicted in Figure 2. The input is defined by a tuple with the start position and the end position, followed by the wall list of the maze. The output is composed of a set of Position sequences, representing each of the three possible paths. It is also considered that, if the program produces two of the three possible paths, it will receive 50% of the total input grade. Moreover, if the program only outputs one of the three possible paths, it will receive 25% of the total input grade.

4 Conclusion

Along this document, the problem of automatically grading the solutions submitted by students to programming exercises was introduced and characterized. This contextualization gave the motivation for the research topic of this work: improve a traditional dynamic grading system with the ability to interpret the meaning of the output, instead of a strict syntactic comparison. Moreover, the capability of marking partially corrected solutions was also considered. The deep study of the state of the art on AGS has shown that there is no other system supporting both requirements that we consider crucial for the successful use of such systems in learning environments.

We proposed an architecture for the Flexible Dynamic Analyzer (FDA) module to achieve the identified objectives. Also, we proposed a DSL, named OSSL, to support the output semantic specification. OSSL grammar was presented in the paper, and its use illustrated.

We strongly believe that the proposed approach is user-friendly (OSSL allows to specify the output meaning in a simple way) and is easy 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.

As this is an undergoing project, obviously the future work is concerned with the proposal

Listing 9 OSSL definition for the maze exercise.

```
PROBLEM: FindPaths TESTS: 1 TOTAL: 4

INPUT: TUPLE < (1 ,A) , (6 ,E) ,
               SET  < ((1 ,A) ,R) ,((2 ,A) ,B) ,((2 ,B) ,R) ,((2 ,C) ,R) ,
                      ((2 ,D) ,T) ,((2 ,E) ,R) ,((3 ,B) ,R) ,((3 ,C) ,R) ,
                      ((3 ,D) ,R) ,((4 ,A) ,R) ,((4 ,B) ,R) ,((4 ,C) ,R) ,
                      ((4 ,D) ,R) ,((4 ,E) ,B) ,((5 ,A) ,R) ,((5 ,B) ,R) ,
                      ((5 ,C) ,R) ,((5 ,E) ,R) > >

OUTPUT: SET < SEQ < (1 ,A) ,(2 ,A) ,(2 ,B) ,(1 ,B) ,(1 ,C) ,(2 ,C) ,
                    (3 ,C) ,(4 ,C) ,(5 ,C) ,(6 ,C) ,(6 ,D) ,(6 ,E) >,
              SEQ < (1 ,A) ,(2 ,A) ,(2 ,B) ,(3 ,B) ,(4 ,B) ,(5 ,B) ,
                    (6 ,B) ,(6 ,C) ,(5 ,C) ,(6 ,C) ,(6 ,D) ,(6 ,E) >,
              SEQ < (1 ,A) ,(2 ,A) ,(2 ,B) ,(3 ,B) ,(3 ,A) ,(4 ,A) ,
                    (5 ,A) ,(6 ,A) ,(6 ,B) ,(6 ,C) ,(6 ,D) ,(6 ,E) > > (4)
       ALSO SET < SEQ < (1 ,A) ,(2 ,A) ,(2 ,B) ,(1 ,B) ,(1 ,C) ,(2 ,C) ,
                        (3 ,C) ,(4 ,C) ,(5 ,C) ,(6 ,C) ,(6 ,D) ,(6 ,E) >,
                  SEQ < (1 ,A) ,(2 ,A) ,(2 ,B) ,(3 ,B) ,(4 ,B) ,(5 ,B) ,
                        (6 ,B) ,(6 ,C) ,(5 ,C) ,(6 ,C) ,(6 ,D) ,(6 ,E) > > (0.5)
       ALSO SET < SEQ < (1 ,A) ,(2 ,A) ,(2 ,B) ,(1 ,B) ,(1 ,C) ,(2 ,C) ,
                        (3 ,C) ,(4 ,C) ,(5 ,C) ,(6 ,C) ,(6 ,D) ,(6 ,E) >,
                  SEQ < (1 ,A) ,(2 ,A) ,(2 ,B) ,(3 ,B) ,(3 ,A) ,(4 ,A) ,
                        (5 ,A) ,(6 ,A) ,(6 ,B) ,(6 ,C) ,(6 ,D) ,(6 ,E) > > (0.5)
       ALSO SET < SEQ < (1 ,A) ,(2 ,A) ,(2 ,B) ,(3 ,B) ,(4 ,B) ,(5 ,B) ,
                        (6 ,B) ,(6 ,C) ,(5 ,C) ,(6 ,C) ,(6 ,D) ,(6 ,E)>,
                  SEQ < (1 ,A) ,(2 ,A) ,(2 ,B) ,(3 ,B) ,(3 ,A) ,(4 ,A) ,
                        (5 ,A) ,(6 ,A) ,(6 ,B) ,(6 ,C) ,(6 ,D) ,(6 ,E) > > (0.5)
       ALSO SEQ < (1 ,A) ,(2 ,A) ,(2 ,B) ,(1 ,B) ,(1 ,C) ,(2 ,C) ,
                  (3 ,C) ,(4 ,C) ,(5 ,C) ,(6 ,C) ,(6 ,D) ,(6 ,E) > (0.25)
       ALSO SEQ < (1 ,A) ,(2 ,A) ,(2 ,B) ,(3 ,B) ,(4 ,B) ,(5 ,B) ,
                  (6 ,B) ,(6 ,C) ,(5 ,C) ,(6 ,C) ,(6 ,D) ,(6 ,E) > (0.25)
       ALSO SEQ < (1 ,A) ,(2 ,A) ,(2 ,B) ,(3 ,B) ,(3 ,A) (4 ,A) ,
                  (5 ,A) ,(6 ,A) ,(6 ,B) ,(6 ,C) ,(6 ,D) ,(6 ,E) > (0.25)
```

implementation. The example presented in Listing 9 illustrates well on how partially correct answers can be mathematically seen. When the expected output is a *set* of values, we can see the associated partially correct answers as *subsets* of the correct set. We believe that it could be a benefit to extend the OSSL grammar to allow not only *subset* definitions, but also to support ranges and other subtype definitions like subsequences or subgraphs, concerning the compound values currently supported and their meaning in terms of partially correct output definition. Besides that, this example also shows what is the main benefit of the proposed output typification: support the definition of output patterns for describing both correct and partially correct answers. These extensions will allow a simpler definition of the correct answers and improve the readability of the output definition.

Moreover, and regarding the literature studied [41, 36], the support for automatic test data generation is not a closed option in the future. The proposed architecture is able to support it by exploiting PExIL functionalities and with the implementation of an OSSL generator. However, in this initial phase of the project, it is irrelevant how the set of tests is defined – it is not the focus of this paper. A manual definition of the set of tests will not interfere with the OSSL language and its main features.

After extending OSSL, we will use Quimera [10] system to integrate and test the FDA. As soon as the new system is available, we intend to test it with real users. We plan to design and implement an experiment in real learning environments to assess the usability and performance of the proposed system. This experiment will also allow us to evaluate the benefits of the learning approach defended along this document.

Acknowledgements The authors are in debt to the anonymous Referees for their valuable comments that have clearly contribute for the progress of our proposal as well as for the improvement of the paper. We also acknowledge the numerous fruitful discussions with Nuno Oliveira and Ismael Vilas Boas – Quimera's co-author and a permanent project contributor.

References

1 Kirsti Ala-mutka, Toni Uimonen, and Hannu matti Järvinen. Supporting students in C++ programming courses with automatic program style assessment. *Journal of Information Technology Education*, 3:245–262, 2004.

2 Kirsti M. Ala-Mutka. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education*, 15(2):83–102, 2005.

3 F. AlShamsi and A. Elnagar. An automated assessment and reporting tool for introductory Java programs. In *Innovations in Information Technology (IIT), 2011 International Conference on*, pages 324 –329, april 2011.

4 S D Benford, E K Burke, E Foxley, and C A Higgins. The Ceilidh system for the automatic grading of students on programming courses. In *Proceedings of the 33rd annual on Southeast regional conference*, ACM-SE 33, pages 176–182, New York, NY, USA, 1995. ACM.

5 Don Colton, Leslie Fife, and Andrew Thompson. A Web-based Automatic Program Grader. *Information Systems Education Journal (ISEDJ)*, 4(114), November 2006.

6 Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3), September 2005.

7 Stephen H. Edwards. Improving student performance by evaluating how well students test their own programs. *J. Educ. Resour. Comput.*, 3(3), September 2003.

8 Christopher C. Ellsworth, James B. Fenwick, Jr., and Barry L. Kurtz. The Quiver system. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, SIGCSE '04, pages 205–209, New York, NY, USA, 2004. ACM.

9 J. English. Automated assessment of GUI programs using JEWL. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, page 137–141, Leeds, United Kingdom, 2004. ACM.

10 Daniela Fonte, Ismael Vilas Boas, Daniela da Cruz, Alda Lopes Gançarski, and Pedro Rangel Henriques. Program analysis and evaluation using quimera. In *ICEIS'2012 — 14th International Conference on Enterprise Information Systems*, pages 209–219. INSTICC, June 2012.

11 G. E. Forsythe and N. Wirth. Automatic Grading Programs. Technical report, Stanford University, 1965.

12 E. Foxley, C. Higgins, T. Hegazy, P. Symeonidis, and A. Tsintsifas. The CourseMaster CBA System: Improvements over Ceilidh. *Fifth International Computer Assisted Assessment Conference*, 2001.

13 Eric Foxley, Colin Higgins, Edmund Burke, and Cleveland Gibbon. The Ceilidh system. *Asian Technology Conference in Mathematics*, pages 430–441, 1997.

14 J. B. Hext and J. W. Winings. An automatic grading scheme for simple programming exercises. *Commun. ACM*, 12(5):272–275, May 1969.

15 Jack Hollingsworth. Automatic graders for programming classes. *Commun. ACM*, 3(10):528–529, October 1960.

16 Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. Identifying and correcting Java programming errors for introductory computer science students. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, SIGCSE '03, pages 153–156, New York, USA, 2003. ACM.

17 Mike Hukk, Dan Powell, and Ewan Klein. Infandango: Automated Greding for Student Programming. In *ITiCSE 2011*, page 330, Darmstadt, Germany, 2011. Association for Computing Machinery.

18 S. Hung, L. Kwok, and R. Chan. Automatic program assessment. *Computers and Education*, 20(2):183–190, 1993.

19 IEEE. IEEE standard VHDL language reference manual. *IEEE Std 1076-1987*, 1988.

20 David Jackson and Michelle Usher. Grading student programs using ASSYST. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, SIGCSE '97, pages 335–339, New York, NY, USA, 1997. ACM.

21 Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. Technical report, , 1975.

22 Edward L. Jones. Grading student programs - a software testing approach. In *Proceedings of the second annual CCSC on Computing in Small Colleges Northwestern conference*, pages 185–192, USA, 2000. Consortium for Computing Sciences in Colleges.

23 Al Lake and Curtis Cook. Style: an automated program style analyzer. *SIGCSE Bull*, 22(3):29–33, August 1990.

24 José Paulo Leal. Managing programming contests with Mooshak. *Software—Practice & Experience*, 2003.

25 José Paulo Leal and Fernando Silva. Mooshak: a Web-based multi-site programming contest system. *Software: Practice and Experience*, 33(6):567–581, May 2003.

26 José Paulo Leal and Fernando Silva. Using Mooshak as a Competitive Learning Tool. *The 2008 Competitive Learning Symposium*, 2008.

27 José Paulo Leal and Ricardo Queirós. CrimsonHex: A Service Oriented Repository of Specialised Learning Objects. In *Enterprise Information Systems*, volume 24 of *Lecture Notes in Business Information Processing*, pages 102–113. Springer Berlin Heidelberg, 2009.

28 José Paulo Leal and Ricardo Queirós. Defining Programming Problems as Learning Objects. In *International Conference on Computer Education and Instructional Technology (ICCEIT)*, 2009.

29 M.E. Lesk and E. Schmidt. Lex — A Lexical Analyzer Generator. *Unix Time-sharing system: Unix programmer's manual*, 2B, July 1975.

30 S. A. Mengel and J. Ulans. Using Verilog LOGISCOPE to analyze student programs. In *Proceedings of the 28th Annual Frontiers in Education - Volume 03*, FIE '98, pages 1213–1218, Washington, DC, USA, 1998. IEEE Computer Society.

31 Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

32 G. Michaelson. Automatic analysis of functional program style. In *Australian Software Engineering Conference, 1996., Proceedings of 1996*, pages 38 –46, jul 1996.

33 D.S. Morris. Automatically grading Java programming assignments via reflection, inheritance, and regular expressions. In *Frontiers in Education, 2002. FIE 2002. 32nd Annual*, volume 1, pages T3G–22, 2002.

34 Kevin A. Naudé, Jean H. Greyling, and Dieter Vogts. Marking student programs using graph similarity. *Comput. Educ.*, 54(2):545–561, February 2010.

35 Lutz Prechelt, Guido Malpohl, and Michael Phlippsen. JPlag: Finding plagiarisms among a set of programs. Technical report, , 2000.

36 Ricardo Queirós and José Paulo Leal. Programming Exercises Evaluation Systems - An Interoperability Survey. In *Proceedings of the 4th International Conference on Computer Supported Education (CSEDU)*, pages 83–90, 2012.

37 Ricardo Queirós and José Paulo Leal. A Survey on eLearning Content Standardization. In *Information Systems, E-learning, and Knowledge Management Research*, volume 278 of *Communications in Computer and Information Science*, pages 433–438. Springer Berlin Heidelberg, 2013.

38 Ricardo Queirós and José Paulo Leal. Making Programming Exercises Interoperable with PExIL. In *Innovations in XML Applications and Metadata Management: Advancing Technologies*, chapter 3, pages 38–56. IGI Global, 2013.

39 Dave Raggett, Arnaud Le Hors, and Ian Jacobs. *HTML 4.01 Specification*, Dec. 1999.

40 Kenneth A. Reek. The TRY system -or- how to avoid testing student programs. In *Proceedings of the twentieth SIGCSE technical symposium on Computer science education*, SIGCSE '89, pages 112–116, New York, NY, USA, 1989. ACM.

41 R. Romli, S. Sulaiman, and Kamal Zuhairi Zamli. Automatic Programming Assessment and Test Data Generation: a review on its approaches. In *2010 International Symposium in Information Technology (ITSim)*, volume 3, pages 1186–1192, 2010.

42 Riku Saikkonen, Lauri Malmi, and Ari Korhonen. Fully automatic assessment of programming exercises. In *Proceedings of the 6th annual conference on Innovation and technology in computer science education*, ITiCSE '01, pages 133–136, New York, USA, 2001. ACM.

43 Tom Schorsch. CAP: an automated self-assessment tool to check Pascal programs for syntax, logic and style errors. In *Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education*, SIGCSE '95, pages 168–172, New York, USA, 1995. ACM.

44 Anuj Shah. Web-CAT: A Web-based Center for Automated Testing. Technical report, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, 2003.

45 Nghi Truong, Peter Bancroft, and Paul Roe. A web based environment for learning to program. In *Proceedings of the 26th Australasian computer science conference*, volume 16 of *ACSC '03*, pages 255–264, Darlinghurst, Australia, 2003. Australian Computer Society.

46 Nghi Truong, Paul Roe, and Peter Bancroft. Static analysis of students' Java programs. In *Proceedings of the Sixth Australasian Conference on Computing Education - Volume 30*, ACE '04, pages 317–325, Darlinghurst, Australia, 2004. Australian Computer Society, Inc.

47 Urs Von Matt. Kassandra: the automatic grading system. *SIGCUE Outlook*, 22(1):26–40, January 1994.

48 Tiantian Wang, Xiaohong Su, Peijun Ma, Yuying Wang, and Kuanquan Wang. Ability-training-oriented automated assessment in introductory programming course. *Comput. Educ.*, 56:220–226, January 2011.

49 Tiantian Wang, Xiaohong Su, Yuying Wang, and Peijun Ma. Semantic similarity-based grading of student programs. *Information and Software Technology*, 49(2):99 – 107, 2007.

50 N. Zamin, E. E. Mustapha, S. K. Sugathan, M. Mehat, and E. Anuar. Development of a Web-based Automated Grading System for Programming Assignments using Static Analysis Approach. In *International Conference on Technology and Operations Management (ICTOM'06)*, Institute Technology Bandung, Indonesia, December 2006.

51 K. Zen, D.N.F.A. Iskandar, and O. Linang. Using Latent Semantic Analysis for automated grading programming assignments. In *2011 International Conference on Semantic Technology and Information Retrieval (STAIR)*, pages 82 –88, june 2011.

52 Xiao Zhao, Liu Xuefeng, and Hou Yumo. Research and Implementation of Automatic Scoring System about Programming. In *International Conference on Computer Science Service System (CSSS)*, pages 225 –227, aug. 2012.