Pythia Reloaded: An Intelligent Unit Testing-Based Code Grader for Education

Sébastien Combéfis
École Centrale des Arts et Métiers
Promenade de l'Alma 50
1200 Woluwé-Saint-Lambert, Belgium
s.combefis@ecam.be

Alexis Paques
Computer Science and IT in Education ASBL
1348 Louvain-la-Neuve, Belgium
alexis.paques@csited.be

ABSTRACT

Automatic assessment of code to support education is an important feature of many programming learning platforms. Unit testing frameworks can be used to perform a systematic functional test of codes; they are mainly used by developers. Competition graders can be used to safely execute code in sandboxed environments; they are mainly used for programming contests. This paper proposes a platform combining the advantages of unit testing and competition graders to provide a unit testing-based grader. The proposed platform assesses codes and produces relevant and "intelligent" feedbacks to support learning. The paper presents the architecture of the platform and how the unit tests are designed.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques

General Terms

Verification, Measurement, Security

Keywords

Code grader, Unit testing, Education

1. INTRODUCTION

Automatic assessment of codes has become a very demanded feature for many programming learning platforms [6]. This demand recently increased with the apparition of *Massive Open Online Courses* (MOOCs), for which it is not possible to handle code assessments by hand, especially due to the massive character of MOOCs [3].

This paper proposes a platform that can automatically grade codes, so that the grading, and in particular the generated feedback, is suited for education and help learners. The proposed platform, which has previously been prototyped [4], has been completely rewritten and released as an open-source project available at the following address: `http://www.pythia-project.org/`. Pythia has already been used to support university courses, as well as for a MOOC [3].

CHESE'15, July 14, 2015, Baltimore, MD, USA
ACM. 978-1-4503-3711-3/15/07
http://dx.doi.org/10.1145/2792404.2792407

1.1 Motivation

Two major kinds of code assessment tools exist: unit testing frameworks and competition graders. Unit testing frameworks, available for most programming languages, are used to perform functional tests on given programs. Competition graders are used to execute code in sandboxes and add constraints such as the maximal amounts of memory and execution time that can be used for the execution of the graded programs. Moreover, competition graders must guarantee uniform conditions for all the executed tests, since they are used to establish rankings for competitions.

When it comes to codes assessment for educational purposes, neither classical unit testing frameworks nor competition graders are suitable. The main lack of unit testing frameworks is that they output a report only stating which tests succeeded or failed, with the inputs and the produced and expected outputs. Whereas such reports are useful for developers, it is not a suitable feedback for learners. Another issue is related to the safe execution of learners' codes. Competitions graders are isolating the execution into sandboxes. Any malicious code, or unintentionally dangerous code such as an infinite loop, for example, will therefore be isolated and will not compromise the grading platform. The main issue with competition graders is that they are very specific to satisfy the constraints they have to meet. Therefore, they often only support a single programming language.

These observations motivated the development of Pythia [4], a platform that combines a unit testing framework with a competition grader. The idea taken from competition graders is the isolated sandbox for the safe execution of code. The unit testing framework brings a systematic way to test codes produced by the learners. In addition to those two elements, an "intelligent" feedback system has been added to the platform to make it supporting learning.

1.2 Related Work

Other platforms using concepts from unit testing have been developed to support education. Code Hunt [2] asks its learners to correct a code given the results of unit tests of the code. It therefore works in the reverse way, com-

pared to Pythia, which runs tests sets on a code that is written by learners. Python Koans [1] is an interactive tutorial for learning Python that requires the learner to make tests pass in order to progress. The FW4EX framework is very similar to Pythia and proposes a mechanised grading of code for MOOCs [7]. The main differences are the use of heavy virtual machines and less focus on feedbacks relevant for learning. Singh *et al.* proposes a method to automatically provide feedback for introductory programming problems [8]. Their system is able to identify the minimal correction to make to the student's wrong code, compared to a reference solution. Compared to Pythia, their feedbacks are more precise and target directly the changes to bring to the code, rather than a higher level description of what is wrong, at the problem level. Finally, Marmoset is a platform that provides learners with a limited access to the teacher's private tests to encourage them to work earlier on their assignments [9]. A difference with Pythia is the focus of Marmoset on snapshots, making it possible to track learners' programming habits.

The remainder of the paper is organised as follows. Section 2 presents the architecture of the Pythia platform, and the structure of the tasks that are executed on it. Section 3 then explains how classical unit tests have been enriched to provide "intelligent" feedbacks to learners.

2. ARCHITECTURE

The architecture of Pythia is organised in three layers, shown on Figure 1. The system layer is responsible of the safe execution of jobs. The task layer structures the jobs to add feedbacks. The problem layer adds context and input constraints to tasks, and organises them into problems.

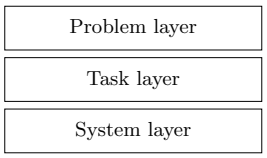

Figure 1: Pythia consists of three layers.

2.1 System Layer

The system layer handles the execution of jobs in a safe environment. The architecture of this layer relies on a queue as shown on Figure 2. The other components, namely the pools and the front-ends, connect to the queue. The front-ends are components communicating with the outside world and receive execution queries. The pools are components managing the safe execution of jobs on the machine.

As it was the case with the prototype version [4], a disposable virtual machine is created by the pool component for every job to be executed. Pythia uses User-mode Linux [5] with a trimmed down version of ArchLinux able to boot in under one second. New pools have to declare their existence to the queue by connecting to it. A pool has a given capacity, *i.e.* the number of virtual machines it can launch and handle at the same time. Front-ends can be of very different nature, and connect to the queue to launch the execution

[1] `https://github.com/brainstorm/python_koans`.

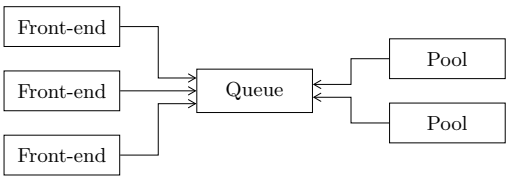

Figure 2: The system layer relies on a central component called queue on which other components are connected.

of a job. The front-end provides inputs to feed the job and receives output upon completion of the job execution.

2.2 Structure of Jobs

At the system layer, a job can be seen as a blackbox containing executable code, which receives input data and produces output data. A job is configured with a JSON file defining the execution environment, the task filesystem containing the executable code and a set of constraints and limits.

Figure 3 shows the configuration file for an *"hello world"* job example, in Python. The contraints and limits that can be set are: maximum execution time (s), maximum amount of memory (Mo), maximum disk space (Mo) and maximum output length (number of characters).

```
{
  "environment":  "python",
  "taskfs":  "hello-world-python.sfs",
  "limits":  {
    "time":  60,
    "memory":  32,
    "disk":  50,
    "output":  1024
  }
}
```

Figure 3: The configuration file for a job defines the execution environment, the task filesystem, constraints and limits.

The task filesystem is mounted in the virtual machine when the job is executed. It should therefore contain an executable file which is automatically executed by Pythia. The task filesystem is mounted on `/task` as a read-only filesystem. If a job need to write files, it has to do it in `/tmp`. Finally, the entry point of a job is an executable file `control` containing a sequence of commands to execute. Figure 4 shows the files contained in the task filesystem for *"hello world"* example.

```
/task/hello.py
```

(a) `control`.

```
Hello World!
```

(b) `data.txt`.

```
#!/usr/bin/python3

with open('/task/data.txt', 'r') as file:
  print(file.read())
```

(c) `hello.py`.

Figure 4: The task filesystem is mounted in the virtual machine and contains the code to be executed.

3. UNIT TESTING-BASED GRADING

Using the services offered by the system layer, the task layer proposes several different kinds of tasks. For example, a task can evaluate the time complexity of an algorithm by executing it with different input sizes, and plotting the execution times with respect to the input sizes. Another kind of task consists in running unit tests and then providing useful feedback to help the learner to find his/her error.

As detailed in the introduction, only using unit tests is not enough for learning purpose. It is very important to provide relevant feedbacks to the learners, that is directly related to the code he/she is writing and to the test case that failed. The task layer defines the format of the input and output received and generated by a job, for all the different offered kinds of tasks.

3.1 Main Executable

The main executable, *i.e.* the `control` file, of a unit testing-based task is organised as a sequence of six steps:

1. **Preprocess**: the *preprocessing* phase integrates the snippets of code written by the learner into skeleton files. Skeleton files are generally source code files that contain placeholders for the snippets of code.
2. **Compile**: the whole program to be executed is then *compiled* if the programming language used requires such a compilation step. Any other kinds of analyses to perform on the source code are done in this step.
3. **Generate**: random test sets are then *generated*, according to instructions from the test configuration file. The tests sets are saved in a file.
4. **Execute**: the code of the learner is then *executed* with the tests sets previously generated. The produced output are saved to a file, along with any additional data.
5. **Postprocess**: any *postprocessing* based on data produced in the previous steps are analysed, for example to build a plot with measured time complexities.
6. **Feedback**: finally, *feedbacks* about the results of the analyses of the code of the learner performed in the previous steps are generated.

Steps 2 and 4 are language-dependent steps. Since they involve code written by the learner, they are executed by a restricted user inside the virtual machine who has very few privileges. In addition to the `control` executable file, the task filesystem is composed of the following directories:

- `skeleton` contains the skeletons files, that is, files that contains placeholders for code snippets from the learner;
- `static` contains the files that do not need any changes or treatment;
- `config` contains the configuration files needed by the different execution steps described above;
- and finally, `scripts` contains the scripts that manage the different execution steps described above.

3.2 Test Configuration

Creating a unit testing-based tasks do not require a lot of code to be produced. It mainly consists in writing several configuration files. Figure 5 shows the main configuration file for a unit testing-based task which requires the learner to write a function to compute the sum of two integers. The top-most label `q1` is the task identifier. Each task is described by four elements:

- `argc` contains the number of arguments of the function to be written by the learner;
- `predefined` is a set of predefined tests to which customised feedback can be attached;
- `random` defines a set of random tests that will be automatically generated;
- and finally `code` contains a correct code for the function that will be used to compute the correct answers for the tests sets.

```
{
  "q1":  {
    "argc":  2,
    "predefined":  {
      "argv":  [{
        "data":  "(10, 5)",
        "feedback":  {
          "10":  "Have you summed the 2nd parameter?",
          "5":  "Have you summed the 1st parameter?"
        }
      }, {
        "data":  "(7, 15)"
      }, {
        "data":  "(-1, 2)",
        "feedback":  {
          "*":  "Have considered negative parameter?",
        }
      }, {
        "data":  "(12, 0)"
      }]
    },
    "random":  {
      "n":  10,
      "args":  ["int(-20,20)", "int(-20,20)"]
    },
    "code":  "def sum(a, b):\n return a + b"
  }
}
```

Figure 5: The configuration file for a unit testing-based task for a function defines the number of arguments, predefined and random tests and the correct code.

The predefined tests must cover cases that will test the errors that are more often made by learners. Those tests are associated with specific feedback messages that help the learner to find the error that was made. In the example above, the first predefined test set is $(10, 5)$. If the code of the learner produces the erroneous output 10, he will get the following feedback message:

> *"Your code failed for the input* $a = 10, b = 5$.
> *The expected result is* 15 *and your code produced* 10.
> *Have you summed the 2nd parameter?"*

The random tests are used to avoid the learner to hardcode the output for all the predefined test sets, in the code they submitted. The configuration file specifies the number of tests to generate and the type and ranges for each parameter. In this example, `int(-20,20)` means that integer values will be randomly picked between −20 and 20.

3.3 Job Input and Output

The input that is provided to the job is a JSON file containing the task identifier, and the code snippets produced by the learner with the associated field identifiers. Figure 6 shows an example of input for the *"sum"* task described in the previous section. The task `q1` asked one code snippet to the user, identified by `f1`. The learner has just written the "`return a`" code snippet.

```
{
  "tid": "q1",
  "fields": {
    "f1": "return a"
  }
}
```

Figure 6: Input provided to a unit testing-based task contains the code snippets of the learner, with their identifiers.

The output produced by the job after its execution is also a JSON file that contains the task identifier, the execution status and feedback information. Figure 7 shows the output generated for the *"sum"* task example described above. The status is `failed`, meaning that the task has not been solved successfully. Two feedback items have been produced:

- the `example` item provides information about the input, expected and actual output of the test that failed;
- and the `message` item provides an additional message.

Other possible feedback items include `score` and `graph`.

```
{
  "tid": "q1",
  "status": "failed",
  "feedback": {
    "example": {
       "input": "(10, 5)",
       "expected": "15",
       "actual": "10"
     },
    "message": "Have you summed the 2nd parameter?"
  }
}
```

Figure 7: Output generated by a unit testing-based task contains feedback messages.

4. CONCLUSION

Unit testing framework is generally used by developers in test-driven development [1], for example. However, for educational purpose, the output produced by such frameworks is not enough. Feedbacks must be provided to the learner when a test fails, in order to support his/her learning. Just showing the expected output is not enough. Some kind of "intelligent" feedbacks must be provided, relating the test case to the function to be written and the context.

This paper proposes Pythia, an open-source platform containing a unit testing-based grader specifically designed for education. Pythia is a genuine combination of a competition grader and a unit testing framework. Pythia is more than just a unit testing-based grader, it a a modular code execution platform which is capable to generate "intelligent" feedbacks to support learning of programming.

Future work includes specifying precisely and developing new kinds of tasks with the corresponding feedback. For example, it could be possible to implement tasks whose time complexity, quality of code, or good application of good programming patterns are assessed. It could also be possible to analyse the code of learner at others levels such as analysing a class, or evaluating whether all the allocated memory in a C program has been freed, for example.

5. REFERENCES

[1] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2nd edition, 2004.

[2] J. Bishop, R. N. Horspool, T. Xie, N. Tillmann, and J. de Halleux. Code Hunt: Experience with coding contests at scale. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*. ACM, May 2015.

[3] S. Combéfis, A. Bibal, and P. V. Roy. Recasting a traditional course into a mooc by means of a spoc. In *Proceedings of the European MOOCs Stakeholders Summit 2014 (EMOOCS 2014)*, pages 205–208, feb 2014.

[4] S. Combéfis and V. le Clément de Saint-Marcq. Teaching programming and algorithm design with pythia, a web-based learning platform. *Olympiads in Informatics*, 6:31–43, 2012.

[5] J. Dike. A user-mode port of the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference*. Usenix, 2000.

[6] V. Pieterse. Automated assessment of programming assignments. In *Proceedings of the 3rd Computer Science Education Research Conference (CSERC 2013)*, pages 45–56, apr 2013.

[7] C. Queinnec. An infrastucture for mechanised grading. In *Proceedings of the 2nd International Conference on Computer Supported Education (CSEDU 2010)*, pages 37–45, apr 2010.

[8] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*, pages 15–26. ACM, jun 2013.

[9] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. Padua-Perez. Experiences with Marmoset: Designing and using an advanced submission and testing system for programming courses. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2006)*, pages 13–17. ACM, jun 2006.