Towards Automated Testing for Simple Programming Exercises

Pierre Ortegat
pierre.ortegat@student.unamur.be
University of Namur
Namur, Belgium

Benoît Vanderose
benoit.vanderose@unamur.be
NaDI, University of Namur
Namur, Belgium

Xavier Devroey
xavier.devroey@unamur.be
NaDI, University of Namur
Namur, Belgium

ABSTRACT

Automated feedback and grading platforms can require substantial effort when encoding new programming exercises for first-year students. Such exercises are usually simple but require defining several test cases to ensure their functional correctness. This paper describes our initial effort to leverage automated test case generation for simple programming exercises. We rely on grey-box fuzzing and random combinations of method calls to test the students' solutions and compare their execution to the results produced by a reference implementation. We implemented our approach in a prototype, called SimPyTest, openly available on GitHub. We discuss its usage and possible future extensions.

CCS CONCEPTS

• **Social and professional topics** → **Software engineering education**.

KEYWORDS

programming education, automated software testing, fuzzing

ACM Reference Format:
Pierre Ortegat, Benoît Vanderose, and Xavier Devroey. 2022. Towards Automated Testing for Simple Programming Exercises. In *Proceedings of the 4th International Workshop on Education through Advanced Software Engineering and Artificial Intelligence (EASEAI '22), November 18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3548660.3561334

1 INTRODUCTION

Following the *learning by doing* philosophy, first-year programming and algorithmic courses usually require students to learn programming through practical sessions. Typically, such sessions include tasks where students design algorithms and data structures and write programs for simple (yet not trivial) problems. Typical examples include defining list structures and various sorting algorithms. Over the years, such courses started to rely on automated grading platforms [8, 11], such as INGInious [6, 7], allowing (i) to cope with the ever-growing number of students, (ii) provide a larger number of exercises together with (iii) rapid formative feedback.

Designing and encoding exercises in platforms such as INGInious is not trivial and requires a substantial effort for a given course [13]. The teacher has to design the exercise together with a way to provide feedback to the students when they submit their code. Such feedback must check that the code compiles and executes correctly *w.r.t.* what is expected for the given exercise. As proving the correctness of the student's code would be too costly [6], INGInious relies on testing. When students submit their code, the system executes it against a set of unit tests defined by the teacher to ensure that the code behaves as expected and that proper feedback is provided to the student. For each exercise, the teacher has to define a set of unit tests covering expected behaviours and preventing invalid ones (for instance, by testing corner cases). In addition, the teacher will have to define a correct program to check that the tests work as expected. Multiplied by the number of exercises, designing practical sessions for a programming course can be time-intensive.

Over the years, researchers have developed many automated test case generation and execution approaches relying on various artefacts to generate tests. Among those approaches, search-based test case generation [10] depends on the source code of an application to generate tests exercising different paths of the code. The generation process is driven by one or several criteria like, for instance, branch coverage, requiring the tests to execute the different branches of the program. Unlike search-based test case generation, fuzzing [14] does not necessarily require access to the application's source code under test. For instance, *black-box* fuzzing [14] approaches will generate a large number of random input values to try to provoke a crash of the application under test. Other fuzzing approaches rely on partial (*grey-box* fuzzing [14]) or full (*white-box* fuzzing [14]) analysis of the program under test to drive the input generation process and exercise new paths during the execution.

In this paper, we explore how automated test case generation can help reduce the effort of creating new programming exercises while providing useful feedback. Unlike other approaches [9], we rely on fuzzing and random combinations of method calls, as the programming exercises are small, with well-defined inputs and outputs and come with a (presumably) *correct reference implementation* of the program the students have to write. This correct version allows comparing the outputs of the students' programs against the output produced by the correct version. We provide a prototype, SimPyTest, openly available at https://github.com/reirep/unamur-tests-python-auto.

2 CONTEXT

The context of this research is a first-year bachelor's course teaching the basics of algorithmic and data structures at the *University of Namur*. Teaching activities include lectures on specification using pre-post conditions, recursion, dynamic programming, memoisation, greedy algorithms and data structures (list, red-black-trees, *etc.*), and practical sessions during which students have to write algorithms for simple problems [5]. The course relies on Python and is taught for one semester off-site from the main campus, which means that the teaching team is available only one day per week

EASEAI '22, November 18, 2022, Singapore, Singapore

for in-person meetings with students. In this context, the teaching team has decided to use INGInious to help students develop their programming skills outside the practical sessions.

From our context, we identify the following challenges: (1) the course started recently in January 2022 and creating a set of programming exercises with their test cases is tedious and time-consuming [13]. (2) The availability of the teaching team on-site is limited, which poses several challenges in providing appropriate feedback to students. This is also an opportunity to work in a two-step process where we first provide automated formative feedback and, if needed, we can provide additional feedback. (3) The practical sessions have two different types of exercises: solving *algorithmic problems* for which the students have to apply one of the program construction method (*i.e.,* recursion, memoisation, dynamic programming, or greedy algorithms) seen during the course, and *data structures definitions* and manipulations, for which students have to specify and implement operations for a data type (*e.g.,* sets, piles, queues, *etc.*) using one of the data structures (*i.e.,* lists, trees, red-black trees) seen during the course [5]. (4) Automated feedback is challenging as it requires providing information that can help students improve. We also assume that the exercises are (relatively) simple and well defined, which is an opportunity to use automated test case generation approaches [10, 14].

For each exercise, we have the following elements: (1) a *description* of the problem to solve, with some examples illustrating the inputs and expected outputs; (2) a *specification* in the form of a semi-formal definition of the pre and post-conditions for algorithmic problems or the definition of the data structure for data structure definition; (3) a *reference implementation* of the algorithm in the form of a function (for algorithmic problems) or a class implementing the internal structure and different operations (for data structure definitions). Only the descriptions are sent to the students. The teachers use the specifications and the implementations to provide feedback on the exercises during the practical sessions.

To answer the different challenges, we envision an extension of INGInious able to automatically generate and execute tests on students' code and provide feedback about the conditions in which a test fails. The following section details the implementation choices of our prototype for *Simple Python code Testing* (SimPyTest).

3 SIMPYTEST PROTOTYPE

INGInious [6, 7] is a free open source web-based automated grading system for programming exercises. Similar to other platforms [4, 12], it is primarily used to help teach programming to beginners. It has been developed since 2014 and adapted to other courses [3]. We select INGInious as a basis to implement our prototype primarily due to its availability and the extension possibilities it offers.

3.1 Automated Testing

As explained in Section 2, we have a semi-formal specification and a reference implementation of the solution for each exercise. Since semi-formal specifications cannot be used as-is for test case generation, we considered various automated test case generation approaches (listed in Table 1) able to work from the source code.

Random testing [1] consists in randomly generating input values for a program under test and checking if, when executed with those values, the program crashes or not. **Property-based testing** [1] requires to define properties that the program under test should satisfy. In our case, one would define such properties according to the pre and post-conditions. A random generator then tries to generate counter-examples input values for which the output would not respect the defined properties. **Fuzzing** [14] relies on the rapid generation of structured input data (*e.g.,* formatted strings) and execution of the program under test to trigger unexpected behaviours and cause crashes. A typical fuzzer starts from an initial set of valid inputs (*i.e.,* the initial seeds) and will iterate over the following steps for a given time budget: select a random seed, *mutate* it to generate a new seed (*e.g.,* adding, removing or replacing a character in a string), execute the program with this new seed, monitor the execution and report unexpected behaviours, and move to the next iteration. The performance of a fuzzer depends on the balance between the throughput (*i.e.,* the number of inputs it generates and executes the program on) and the complexity of the analysis performed during each iteration. **Black-box fuzzing** [14] is easy to implement but limited as it only considers the input and output values (similarly to random testing). In addition to the output, **grey-box fuzzing** [14] also considers the program's internal structure to collect initial seeds and monitor the blocks of code executed for each given input. **White-box fuzzing** [14] offers the highest guarantee to cover the different parts of the code at the cost of heavier program analysis and the usage of constraint solvers to generate input values [2]. Finally, **Search-based testing** [10] relies on evolutionary computation to evolve and refine a population (*i.e.,* a set) of test cases using evolutionary operators (*i.e.,* mutation and crossover) for a given amount of time. Unlike fuzzing, search-based testing generates full-fledged tests, including calls to various functions and the generation of assertions.

Table 1: Automated test case generation techniques

Technique	Pros	Cons
Random testing [1]	Very easy to implement	No guarantee of coverage, high number of executions
Property-based testing [1]	Covers input and output domains	Requires to formalise and encode the properties
Black-box fuzzing [14]	Easy to implement	Performance depends on the initial seeds
Grey-box fuzzing [14]	Relatively easy to implement, does not require initial seeds	Achieves a lower coverage compared to white-box fuzzing
White-box fuzzing [14]	Allows high coverage of the code	Requires heavy instrumentation of the code, complex to put in place
Search-based testing [10]	Allows high coverage of the code, generates complete unit tests (not only input values)	Requires heavy instrumentation of the code, complex to put in place

3.2 Testing solutions to algorithmic problems

As algorithmic problems should be written as functions with well-defined parameters and return value types, SimPyTest leverages fuzzing to test students' solutions automatically. As explained in Table 1, different fuzzing techniques have different pros and cons. We experimented with the different fuzzing techniques and found that black-box fuzzing was very easy to implement but performed poorly. White-box fuzzing was complex to put in place as it required heavy instrumentation. Moreover, it requires the usage of a constraint solver to generate inputs, which can cause scalability issues when evaluating a large number of students' code. We found

```
1  from correcteur.feedback.textRepporter import TestRepporter
2  from correcteur.fuzzing.fuzz import fuzz
3
4  [...]
5
6  # Run the fuzzer
7  reporter = TestRepporter()
8  valid_modules = ["student_code"]
9  fuzz(reporter, student_code.adder, adder_validate, valid_modules, runs
       =1000)
10
11 # Get the results
12 if len(reporter.get_output()) == 0:
13     print("ok")
14 else:
15     print(reporter.get_text_output())
```

Listing 1: Example of automated correction for a simple algorithmic problem (the addition of two integers)

```
1  from correcteur.feedback.textRepporter import TestRepporter
2  from correcteur.steps.StepsRunner import StepRunner
3  from correcteur.steps.Step import Step
4
5  # Run the fuzzer
6  reporter = TestRepporter()
7  runner = StepRunner(stop_on_first_error=True)
8  runner.add_step(Step([student_code.init],
9      [init], max_depth_local=1, [...]))
10 runner.add_step(Step([lambda: student_code.add(6), student_code.remove,
       student_code.last],
11     [lambda: add(6), remove, last], max_depth_local=10, [...]))
12 runner.compare_codes(reporter)
13
14 # Get the results
15 if len(reporter.get_output()) == 0:
16     print("ok")
17 else:
18     print(reporter.get_text_output())
```

Listing 2: Example of automated correction for a simple data structure definition (a list)

that *grey-box fuzzing* offers a good balance between coverage of the program under test and the overall execution time.

Listing 1 illustrates an example of Python script providing automated feedback in INGInious for a given simple exercise (here, a function `adder(int,int)` returning the sum of two integers). The call to the `fuzz` function (at line 9) requires a `reporter` collecting the results of the execution, the student's implementation `student_code.adder` of the `adder` function, the reference implementation `adder_validate` of the `adder` function, the Python modules that can be used (`valid_modules`), and the number of iterations of the fuzzer (`runs=1000`). After the execution, the results are retrieved from the `reporter` object and printed to the student (line 12). In case of an error or mismatch between the student's implementation and the expected value, the message indicates the returned and expected values or the error message.

3.3 Testing data structures definitions

For a data structure definition, students have to define a class implementing the different methods working with the data structure. For instance for a list, students might have to implement the initialisation of an empty list (`init`), the addition (`add`), removal (`remove`), and retrieval of the last value of the list (`last`). Automatically testing such classes is not as simple as for single functions. As long as the list has been initialised, the different methods might be called in various order and should behave accordingly. For instance, one might add several values before retrieving the last value in the list.

Similarly to what is done in *search-based testing*, testing a data structure with SimPyTest consists in generating and executing combinations of method calls. However, unlike *search-based testing*, the generation is a random search, not driven by any fitness function. Exploring how search-based techniques can be leveraged to increase the coverage of the tests is part of our future work.

Listing 2 provides an example of automatically testing data structure definitions with SimPyTest by declaring different *steps* (lines 8 and 10). Each step contains several methods that might be called in random order. Methods of one step will always be called before the methods of the next step. In our example, the `init` method (at line 8) will always be called once (specified by the `max_depth_local=1` parameter) before any combinations of maximum 10 calls (specified by the `max_depth_local=10` parameter) to other methods (at line 10). For instance, combinations might be `<init, add(6), remove>` or `<init, remove, remove, last>`. For

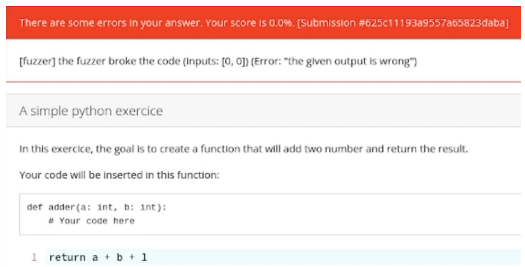

Figure 1: Example of usage of SimPyTest in INGInious for a simple algorithmic problem (the addition of two integers)

each random combination, the execution of the student's implementation (*e.g.,* `student_code.init` at line 8) is compared to the reference implementation (*e.g.,* `init` at line 9). As for algorithmic problems, after the execution, the results are retrieved from the `reporter` object and printed to the student (line 15). In case of an error or mismatch between the student's implementation and the reference implementation, the message indicates the returned and expected values or the error message.

3.4 Providing Feedback

One of the main challenges when dealing with automated testing is to provide feedback that could be understood by the students [6]. For now, SimPyTest limits this feedback to indicate which input values or combinations of method calls did not work on the student's code. For instance, Figure 1 presents an example of feedback for the `adder` function discussed in Section 3.2. The message indicates that the call to the `adder` function with parameter values `(0, 0)` did not return the right value.

Similarly, Figure 2 presents an example of feedback for the list definition discussed in Section 3.3. The message indicates that for the different methods implemented (only the `init` and `add` methods are shown in the Figure due to space constraints), the sequence `<init, add, last>` did provoke an list index out of range exception when executed on the student's code.

As seen from the Figures, the feedback is (for now) minimal, and several improvements are part of our future work. First, the different error messages need to be printed using string templates to ease the understanding and express the error using the same

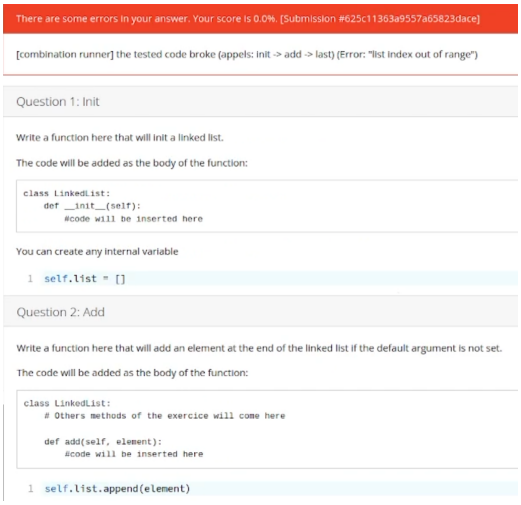

Figure 2: Example of usage of SimPyTest in INGInious for a data structure definition (a list)

language level as the exercise description. For instance, "*adding 0 to 0 did not work*" instead of the message of Figure 1. Second, the feedback has to provide enough details for the student to identify the problem in their code and fix it. One solution could also be to provide the result produced by the reference implementation in SimPyTest. We plan to evaluate different feedback forms for algorithmic problems and data structure definitions.

3.5 Architecture overview

SimPyTest is implemented in Python to be easily integrated to INGInious. Its internal architecture is composed of three main parts: (1) `fuzzing` contains the different classes and functions handling the automated testing of solutions to algorithmic problems. The grey-box fuzzer is based on the implementation of Zeller *et al.* [14] with the necessary modifications to compare the output produced by the students' implementation to the output produced by the reference implementation. (2) `steps` contains the different classes and functions handling the automated testing of solutions to data structure definitions, including the generation of combinations of method calls to test various usage scenarios of the student's classes. (3) `feedback` contains the different classes and functions handling the feedback after executing the student's code. As explained in Section 3.4, this feedback is currently limited, and further developments are needed to enhance the information provided to the students. The source code of SimPyTest is available on GitHub: https://github.com/reirep/unamur-tests-python-auto.

4 CONCLUSION AND FUTURE WORK

In this paper, we illustrated how automated test case generation could be leveraged to automatically correct simple programming exercises to provide rapid feedback to the students while keeping the workload of test case generation reasonable. We have introduced SimPyTest, an open-source prototype relying on fuzzing and random combinations of method calls to test solutions to algorithmic problems and data structure definitions in the context of a first-year algorithmic class at the *University of Namur*. SimPyTest has been implemented as a plugin for INGInious. The approach could be applied to any other automated feedback and grading platform relying on testing to evaluate students' solutions.

SimPyTest can be improved in several ways. First, it has to be evaluated and validated with students in a full-fledged experiment. Also, feedback for students is, for now, limited and could be improved by using string templates to provide more readable messages and additional information such as the expected output value. From a testing perspective, SimPyTest could also be enhanced to provide more guarantees that the students' code behaves as expected, for instance, by using white-box fuzzing for exercises having many corner cases that need to be checked. We would also like to consider other non-functional aspects like the time and space complexity of the students' implementations. Finally, we will evaluate different configurations of SimPyTest with students to strike a good balance between information feedback provided to the student, learning objectives, and the student's motivation.

ACKNOWLEDGMENTS

This research was partially funded by the CyberExcellence (No. 2110186) project, funded by the Service Public de Wallonie (SPW Recherche).

REFERENCES

[1] Mauricio Aniche. 2022. *Effective Software Testing: A Developer's Guide.* Simon and Schuster.
[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2019. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3 (May 2019), 1–39. https://doi.org/10.1145/3182657
[3] O. Bonaventure, Q. De Coninck, F. Duchêne, A. Gégo, M. Jadin, F. Michel, M. Piraux, C. Poncin, and O. Tilmans. 2020. Open educational resources for computer networking. *ACM SIGCOMM Computer Communication Review* 50, 3 (July 2020), 38–45. https://doi.org/10.1145/3411740.3411746
[4] Sébastien Combéfis and Vianney le Clément de Saint-Marcq. 2012. Teaching Programming and Algorithm Design with Pythia, a Web-Based Learning Platform. *Olympiads in Informatics* 6 (2012), 31–43.
[5] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms.* MIT press.
[6] Guillaume Derval, Anthony Gego, Pierre Reinbold, Benjamin Frantzen, and Peter Van Roy. 2015. Automatic grading of programming exercises in a MOOC using the INGInious platform. In *EMOOCS'15.* Mons, Belgium, 86–91.
[7] Guillaume Derval, Anthony Gégo, Ludovic Taffin, and al. 2022. *INGInious.* UCL-INGI. Retrieved July 12, 2022 from https://github.com/UCL-INGI/INGInious
[8] Piotr Duch and Tomasz Jaworski. 2018. Dante - Automated Assessments Tool for Students' Programming Assignments. In *HSI'18.* IEEE, Gdansk, Poland, 162–168. https://doi.org/10.1109/HSI.2018.8431146
[9] Nobuo Funabiki, Ryota Kusaka, Nobuya Ishihara, and Wen-Chung Kao. 2017. A Proposal of Test Code Generation Tool for Java Programming Learning Assistant System. In *AINA'17.* IEEE, Taiwan, 51–56. https://doi.org/10.1109/AINA.2017.60
[10] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. Automated Unit Test Generation for Python. In *SSBSE'20.* Springer International Publishing, 9–24. https://doi.org/10.1007/978-3-030-59762-7_2
[11] Ruan Reis, Gustavo Soares, Melina Mongiovi, and Wilkerson L. Andrade. 2019. Evaluating Feedback Tools in Introductory Programming Classes. In *FIE'19).* IEEE, Covington, KY, USA, 1–7. https://doi.org/10.1109/FIE43999.2019.9028418
[12] Felipe Restrepo-Calle, Jhon Jairo Ramírez-Echeverry, and Fabio A. Gonzalez. 2018. Uncode: Interactive System for Learning and Automatic Evaluation of Computer Programming Skills. In *EduLearn'18.* Palma, Spain, 6888–6898. https://doi.org/10.21125/edulearn.2018.1632
[13] Thomas Staubitz, Ralf Teusner, and Christoph Meinel. 2017. Towards a repository for open auto-gradable programming exercises. In *TALE'17.* IEEE, Hong Kong, 66–73. https://doi.org/10.1109/TALE.2017.8252306
[14] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2021. *The Fuzzing Book.* CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/ Retrieved 2021-10-26.