A Domain-Specific Language for Dynamic White-Box Evaluation of Java Assignments

Afonso B. Caniço
Instituto Universitário de Lisboa (ISCTE-IUL), Portugal

André L. Santos
Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR-IUL, Portugal

Abstract

Programming exercises involving algorithms typically involve time and spatial constraints. Automated assessments for such implementations are often carried out in a black-box manner or through static analysis of the code, without considering the internal execution properties, which could lead to falsely positive evaluations of students' solutions. We present Witter, a domain-specific language for defining white-box test cases for the Java language. We evaluated programming assignment submissions from a Data Structures and Algorithms course against Witter's test cases to determine if our approach could offer additional insight regarding incomplete algorithmic behaviour requirements. We found that a significant amount of student solutions fail to meet the desired algorithmic behavior (approx. 21%), despite passing black-box tests. Hence, we conclude that white-box tests are useful to achieve a thorough automated evaluation of this kind of exercises.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging; Software and its engineering → Domain specific languages; Social and professional topics → Student assessment

Keywords and phrases White-box assessment, student assessment, programming education

Digital Object Identifier 10.4230/OASIcs.ICPEC.2024.2

1 Introduction

Students of introductory-level programming courses, such as Algorithms and Data Structures, are expected to develop implementations that conform to specific algorithm behaviour to ensure the correct application of the algorithms under study. As a standard example, students might be tasked with implementing a specific sorting algorithm. Ideally, formative assessment of this kind of exercises should verify algorithmic behaviour – the essence of the subject – by measuring white-box aspects such as the number of operations executed or memory allocation. Automated constructive feedback that allows students to understand any possible mistakes and deepen their understanding is valuable [18, 13], saving time on human feedback and fostering autonomous learning.

While assessment tools providing feedback about the correctness of the outputs of a solution (i.e. black-box testing) are generally available, assessment tools that check internal algorithmic behavior are not (i.e. white-box testing) [9, 14, 12]. We believe that a technique for deeper evaluation of exercises could serve as the backbone for more elaborated automated assessment systems with richer feedback.

In this paper, we present an evolution of our previous work on Witter [3][1], a library for white-box testing of Java code. We augmented the library with an internal domain-specific language (DSL) written in Kotlin, allowing instructors to define white-box test cases for Java source code that with stateful execution – a limitation of the initial approach. This kind of tests are appropriate to test data structures implemented with classes.

[1] `https://github.com/ambco-iscte/witter`

We tested the DSL against a set of real student programming assignment submissions from the Algorithms and Data Structures course offered at our institution. We observe that students can effectively be misled if only the outputs produced by their implementations are considered in their assessment, and thus conclude that programming assignments where the internal algorithmic behaviour is relevant could benefit from a tool providing execution information.

This paper proceeds as follows. Section 2 discusses related work on automated assessment. Section 3 presents background on our previous work on Witter. Section 4 describes our DSL for test specification. Section 5 describes the evaluation of our approach with student submissions. Section 6 discusses conclusions and outlines future work.

2 Related Work

Recent surveys [9, 14, 12, 5] show that research and development on automated white-box programming assessment systems focus primarily on well-known white-box assessment methods like static analysis or bytecode instrumentation, with our literature review yielding scarce mentions of assessment through the dynamic collection of details of a program's execution – the main novelty of our approach.

Programming languages or libraries such as AspectJ [10][2], Javassist [4][3], ASM [2][4], and ByteBuddy[5], require specilized knowledge on code instrumentation, allow existing systems to be augmented with deeper assessment functionalities, but are limited in that they do not allow for the collection of information regarding the code's execution out-of-the-box. The analyses have to be programmed and tailored to specific needs. Similarly, Valgrind[6] is a code instrumentation toolkit that enables detection of memory management and threading bugs, along with program profiling. While these tools lay the groundwork upon which an automated assessment system may feature white-box analysis, we consider the requirement of specialised knowledge to be a negative factor when it comes to the tools' general accessibility. Additionally, significant analysis or instrumentation is required to accurately collect a relevant amount of information during a program's execution, negatively impacting the applicability of these tools.

Lizard[7] is a multi-language static analysis tool supporting the Java language, focusing on determining the cyclomatic complexity of implemented functions, among other forms of static code analysis. While cyclomatic complexity is a useful tool for measuring and therefore managing the complexity of a program, it does not provide detailed algorithmic behaviour of a solution. Similar tools, like Cppcheck[8] for C and C++, focus on bug detection through static code analysis, providing no dynamic analysis functionalities for a program's execution.

ConGu [6] is a runtime verification tool that enables the assessment of Java classes against formal algebraic specifications. Its main goal is to test abstract data types against function domain restrictions and algebraic conditions or axioms that functions must verify.

[2] `https://www.eclipse.org/aspectj`
[3] `https://www.javassist.org`
[4] `https://asm.ow2.io`
[5] `https://bytebuddy.net/`
[6] `https://valgrind.org/`
[7] `https://github.com/terryyin/lizard`
[8] `https://cppcheck.sourceforge.io/`

Jeed[9] is a toolkit for Java and Kotlin in-memory execution with a focus on safety and performance. While Jeed's goals align with those of our proposed library by enabling code execution in a sandboxed environment providing access to code evaluation metrics, its assessment is focused on source code analysis rather than dynamic runtime events. Namely, Jeed supports linting, cyclomatic complexity analysis, and a listing of which language features are present in a given program.

Mooshak [11] is a programming assessment tool that checks whether a submitted program functions correctly. To this effect, Mooshak analyses the programs for their returned or printed outputs, and if any compilation or runtime errors were produced [17]. Mooshak's goal is broad, aiming to be a full online programming context judge for several programming languages [11, 17], and as such, enables third-party extensions to execute custom static and dynamic analysers, which might perform white-box analyses. This aligns with our goal of providing Witter as a library which is easily integrable into existing assessment systems.

JavAssess [8] is a Java library used to integrate deeper code analysis capabilities into existing automated assessment tools. This approach relies on combining traditional black-box unit testing with code instrumentation and meta-programming functionalities, yet does not offer a way to dynamically collect white-box execution metrics. Nonetheless, the goals of JavAssess are considerably similar to Witter's, aiming to be a library that can be integrated into existing automated assessment systems to facilitate a deeper analysis of student code, and the two libraries could work in tandem to provide a comprehensive assessment toolkit. AutoGrader [7] is a similar assessment library, leveraging on the meta-programming functionalities of the Java language along with typical unit testing for code assessment.

3 Background: Witter Library

In previous work we developed Witter, a library for specifying white-box test cases for Java source code [3]. The core functionality of Witter relies on Strudel[10], a library providing an interpreter allowing clients to perform fine-grained observation of code execution events.

In the first version of Witter, test cases were defined by annotating reference solutions with header comments containing different directives that specified each test case and which runtime metrics should be considered. Figure 1 presents an example of using Witter to specify a test for the *insertion sort* algorithm. We can see two test cases and directives to measure the number of array reads/writes and to check if the desired side-effects are met. The white-box testing is based on comparing the measurements of the reference solution with those of a students' solution. For example, one may implement *selection sort* and end up with a sorted array, but the array read/writes will not match, implying that the implementation is not as intended.

While the initial version of Witter was suitable to evaluate algorithms that could be implemented as standalone methods (e.g., involving arrays), it lacked support for assessing stateful or object-oriented solutions. Furthermore, the limited Java support of Strudel at the time also constrained the scope of code solutions we are able to analyze. Meanwhile, we further developed Strudel to support a larger subset of Java's language features and therefore broaden the scope of programs that Witter is able to evaluate.

[9] `https://github.com/cs125-illinois/jeed`
[10] `https://github.com/andre-santos-pt/strudel`

```
/*
@Test({5, 4, 3, 2, 1})
@Test({3, 2, 5, 3, 1})
@CountArrayReads(2)
@CountArrayWrites(1)
@CheckSideEffects
*/
static void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        for (int j = i; j > 0; j--) {
            if (a[j] >= a[j - 1]) break;
            int tmp = a[i];
            a[i] = a[j];
            a[j] = tmp;
        }
    }
}
```

Figure 1 Reference solution of *insertion sort* annotated with Witter tests (initial version).

Table 1 Runtime metrics and corresponding values measured.

Metric	Usage	Measure
Loop Iterations	`CountLoopIterations(`*[margin]*`)`	Number of loop iterations
Array Reads	`CountArrayReadAccesses(`*[margin]*`)`	Number of array read accesses
Array Writes	`CountArrayWriteAccesses(`*[margin]*`)`	Number of array write accesses
Recursive Calls	`CountRecursiveCalls(`*[margin]*`)`	Number of recursive calls
Object Allocations	`CheckObjectAllocations`	Object allocations per type
Array Allocations	`CheckArrayAllocations`	Array allocations per type
Side Effects	`CheckSideEffects`	Side effects on arguments

4 Approach: DSL for White-Box Tests

In order to tackle the inherent difficulty of supporting the assessment of object-oriented implementations through annotated code solutions, we implemented an internal DSL in Kotlin providing a programmatic way for defining stateful test cases. We argue that the DSL requires less effort from instructors for defining test cases when compared to the required knowledge on specialised topics like code instrumentation or meta-programming, since there are only a few DSL directives to be used in a declarative, high-level style.

Figure 2 illustrates Witter's DSL with a *test suite* for list data structures, containing two *test cases*. These test cases can be configured to use any number of white-box metrics either throughout the test or within a bounded scope (*using* directive). As in the initial version of Witter, evaluation metrics (summarised in Table 1) can be optionally instantiated with a *margin* parameter that specifies an acceptable deviation interval from the reference value, in order not to constrain students' code to a single, rigid solution.

An object can be created using the *new* directive by passing the name of the class to instantiate followed by a list of arguments to one of the class constructors. References to the created objects can be stored using *ref*. Class methods can be invoked by using the *call* directive on a previously declared reference. A sequence of these directives defines a stateful test case.

```
val tests = TestSuite(referencePath = "path/reference/List.java") {
    Case("testContains") {
        // Create new object and store a reference to it
        val list = ref { new("List") }

        // Executed without white-box metrics (black-box only)
        list.call("size") // 0
        list.call("add", "hello")
        list.call("size") // 1
        list.call("add", "world")
        list.call("size") // 2

        using(CountLoopIterations() + CountArrayReadAccesses()) {
            // These calls compare loop iterations
            list.call("contains", "hello") // true
            list.call("contains", "algorithm") // false
        }
    }

    // All the calls within this case compare loop iterations
    Case(CountLoopIterations(), "testIsEmpty") {
        val list = ref { new("List") }
        list.call("isEmpty", expected = true)
        list.call("add", "hello")
        list.call("isEmpty", expected = false)
    }
}
```

Figure 2 Witter's DSL syntax. Example Test Suite for a list data structure comprising the operations *add*, *contains* and *isEmpty*.

The *call* directive is used by specifying the name of the method to be invoked and a list of arguments. We may use the "dot notation" to perform calls on instance methods given its reference (*ref.call(...)*). For every call, the return values of the evaluated method are compared to the reference solution, allowing for regular black-box testing. Additionally, if the optional *expected* argument is passed, Witter will assert that both the reference solution and the solution under evaluation produce the expected result. This verification allows educators to assert that their solution works as expected, preventing the accidental usage of a faulty reference solution as the ground truth for evaluating students' implementations.

Consider an assignment where a student must implement a function for calculating the average of an array of double values. We wish to assess not only the correctness of the produced result, but also that of the algorithm behaviour by checking that the number of loop iterations matches that of the reference solution. Figure 3 illustrates an evaluation scenario for this assignment using Witter's API, composed of: reference solution (3a), test suite (3b), a solution to check against the tests (3c), a snippet of using Witter as a library (3d), and the output of executing the tests (3e).

Educators design an exercise by providing a pair of artifacts consisting of a reference solution and a corresponding test suite. Different solutions to the exercise (submitted by students) may be checked against the test suite through Witter's API. The example submission (Figure 3c) has a defect, given that the array iteration starts at index 1 (rather than 0). Witter runs the tests simultaneously for the reference solution and for the solution

(a) Reference solution (Average.java).

```
static double average(double[] a) {
    double sum = 0.0;
    for (int i = 0; i < a.length; i++) sum += a[i];
    return sum / a.length;
}
```

(b) DSL test suite.

```
val tests = TestSuite("path/to/reference/Average.java") {
    Case(CountLoopIterations()) {
        call("average", listOf(1,2,3,4,5), expected = 3.0)
        call("average", listOf(0,2,3,5,7), expected = 3.4)
    }
}
```

(c) Solution under testing (Solution.java) – with a defect, starting at index 1.

```
static double average(double[] a) {
    double sum = 0.0;
    for (int i = 1; i < a.length; i++) sum += a[i];
    return sum / a.length;
}
```

(d) Invoking the execution of a test suite to a solution under evaluation.

```
val results: List<ITestResult> = tests.apply(
    subjectPath = "path/to/Solution.java"
)
results.forEach { println("$it\n") }
```

(e) Output of the test results.

```
[fail] average([1, 2, 3, 4, 5])
        Expected: 3.0
        Found: 2.8

[fail] average([1, 2, 3, 4, 5])
        Expected loop iterations: 5
        Found: 4

[pass] average([0, 2, 3, 5, 7])
        Expected: 3.4

[fail] average([0, 2, 3, 5, 7])
        Expected loop iterations: 5
        Found: 4
```

Figure 3 Exercise evaluation scenario using Witter's API.

under testing, comparing the two to evaluate the latter's execution. In this case, a mismatch of iterations is detected, given that the solution under testing perform always performs one less iteration in contrast to the reference solution. Notice that in the output of the test results (Figure 3e) this is being reported as a failure, while it succeeds in one of the test inputs.

An automated assessment system may import Witter as a third-party library. In this example we are merely displaying the objects of the test results to the console, but these can be inspected for custom reporting, depending on the assessment system.

5 Evaluation

In order to evaluate the feasibility and usefulness of the approach, we carried out an experiment using the proposed DSL to evaluate student assignments.

5.1 Context

We collected a set of 2,389 student assignment submissions spanning two offerings of the Algorithms and Data Structures course taken by first year undergraduate students of Computer Science and Engineering and related bachelor's degrees. The submission process was independent from our approach, hence no constraints were posed regarding Witter's limitations. For this reason, some submissions could not be handled. We analyzed five distinct assignments with the following guidelines:

1. Implement three different classes, each solving the dynamic connectivity (union-find) problem using different approaches: Quick-Find, Quick-Union, and Weighted Quick-Union with Path Compression.
2. Implement a Queue data structure supporting the String data type. Use a circular resizing array implementation to store the data internally without needing to limit the queue's memory capacity *a priori*.
3. Implement an optimised version of the Insertion Sort algorithm which executes fewer array access operations by avoiding swap operations and instead shifting elements directly one position to the right as needed.
4. Implement a generic List data structure utilising a dynamic sequence of simply-linked nodes internally to store elements.
5. The implementation of the Heap Sort algorithm seen in the lectures assumes the array indices begin at 1. Modify the algorithm's implementation to support sorting arrays starting at index 0.

Each assignment was given to students in this order throughout the semester. The guidelines were translated from Portuguese, which included more API details that have been truncated for presentation in this paper.

Table 2 presents the number of valid submissions used for evaluating the DSL for each assignment. We consider a submission to be valid if the submitted file matches the name and extension indicated in an assignment's guidelines, and if Java can successfully compile the source code. The inconsistent number of submissions is explained by two factors: our observations of a growing rate of absenteeism as the semester progresses; and, every assignment being present in the two course offerings considered except assignment A3, which was only present in a single offering.

Table 2 Number of student submissions used for evaluating Witter's DSL.

Assignment	Description	Total	Valid
A1	Dynamic Connectivity	584	424 (72.6%)
A2	Resizing Array Queue of Strings	573	490 (85.5%)
A3	Improved Insertion Sort	212	120 (56.6%)
A4	Generic Linked List	550	476 (86.5%)
A5	Heap Sort	470	266 (56.6%)
		2389	**1776 (74.3%)**

5.2 Method

Figure 4 presents the test specifications for each assignment using Witter's DSL. Each student submission was evaluated using Witter through the corresponding test specification, and the results of the evaluation were processed to find cases where the student's implementation passed black-box tests but failed white-box tests. Given the nature of the chosen assignments, the evaluation focused on the metrics for counting loop iterations, array read and write access operations, allocated memory (for resizing array operations), and argument side effects (for checking whether sorting algorithms effectively sorted the input array).

In order to assess Witter's suitability for large-scale assessment processes, we measure the average execution time for the evaluation of each submission. The execution took place in a laptop system with a 12-core 2.6GHz Intel i7-9750H CPU and 16GB of RAM. While a completely isolated simulation is impossible in a standard system, care was taken to minimise the impact of other operating system processes on the performance of Witter's execution.

It is usually the case that computer engineering students have a general propensity to cheat in programming assignments [1]. We took this factor into consideration during Witter's evaluation by running plagiarism analysis on all student submissions using JPlag[11], a plagiarism checking tool resistant to a broad range of obfuscation techniques and which provides easily-interpretable results of its analysis [15, 16]. While a connection between plagiarism checking and white-box assessment has not been observed, we include this analysis as a means to safeguard our evaluation against possible accidental biases stemming from students having plagiarised the same incorrect sources (e.g. copying from a fellow classmate who made mistakes).

5.3 Results

Table 3 summarizes the evaluation results. Witter successfully loaded a total of 1,526 student submissions from the 1,776 valid submissions described in Table 2, constituting approximately 86% of all valid submissions. For these, all the assignments had a black-box tests pass rate greater than 90%. However, the white-box tests failure rate ranged approximately between 9% and 45%, with assignment A4 exhibiting the largest failure rate.

Figure 5 presents the distribution of white-box metrics that produced failures in each assignment. Approximately 50% of failed assertions for assignment A1 were caused by an incorrect number of array write operations, with the majority of the remaining errors relating to an incorrect number of array write operations. Out of the considered metrics, assignments A2, A3, and A5 contained white-box errors relating to the number of loop iterations and array

[11] `https://github.com/jplag/JPlag`

(a) Test specification for assignment A1.

```
Case(CountLoopIterations(1) + CountArrayWriteAccesses(1) +
CountArrayReadAccesses(1)) {
    val uf = ref { new("QuickFindUF", 100) }
    val connected = mutableListOf<Pair<Int, Int>>()
    (1 .. 100).forEach { _ ->
        val p = (100 * random()).toInt()
        val q = (100 * random()).toInt()
        uf.call("union", p, q)
        connected.add(Pair(p, q))
    }
    connected.forEach { uf.call("connected", it.first, it.second) }
}
```

(b) Test specification for assignment A2 (excerpt).

```
Case(CountLoopIterations(1), "testDequeue") {
    val queue = ref { new("Queue") }
    queue.call("enqueue", "witter")
    queue.call("enqueue", "is")
    queue.call("enqueue", "cool")
    queue.call("dequeue")
    queue.call("dequeue")
    queue.call("dequeue")
}
Case(CountLoopIterations(1) + CountMemoryUsage()) {
    val queue = ref { new("Queue") }
    (1..100).forEach { queue.call("enqueue", it.toString()) }
}
```

(c) Test specification for assignment A4 (excerpt).

```
Case(CountLoopIterations(2) + CheckObjectAllocations, "testSize") {
    val list = ref { new("List") }
    list.call("size")
    list.call("add", "hello")
    list.call("size")
    list.call("add", "world")
    list.call("size")
}
```

(d) Test specification for assignments A3 and A5.

```
Case(CheckSideEffects + CountLoopIterations(1) +
CountArrayWriteAccesses(1) + CountArrayReadAccesses(1)) {
    call("sort", listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
    call("sort", listOf(7, 3, 2, 1, 5, 6, 10, 8, 9, 4))
    call("sort", listOf(7.32, 3.14, 2.14, 1.93, 5.99, 6.74, 10.21,
                        8.84, 9.26, 4.56))
    call("sort", listOf("sorting", "algorithms", "are", "really",
                        "very", "cool"))
    call("sort", listOf(10, 9, 8, 7, 6, 5, 4, 3, 2, 1))
}
```

Figure 4 Test specifications for the evaluated assignments (A1–5).

Table 3 Number of black-box and white-box passes and failures for loaded submissions. Loaded submissions are given as a % of valid submissions. Submission pass/fails are given as a % of loaded submissions. Execution time is the average over all submissions of the corresponding assignment.

Assignment	Valid	Loaded	Blackbox Pass	Whitebox Fail	Avg. Time
A1	424	413 (97.4%)	407 (98.5%)	79 (19.1%)	850 ms
A2	490	433 (88.4%)	391 (90.3%)	98 (22.6%)	300 ms
A3	120	110 (91.7%)	107 (97.3%)	49 (44.5%)	20 ms
A4	476	347 (72.9%)	347 (100%)	32 (9.22%)	100 ms
A5	266	233 (83.8%)	221 (99.1%)	58 (26.0%)	30 ms
	1776	**1526 (85.9%)**	**1473 (96.5%)**	**316 (20.7%)**	

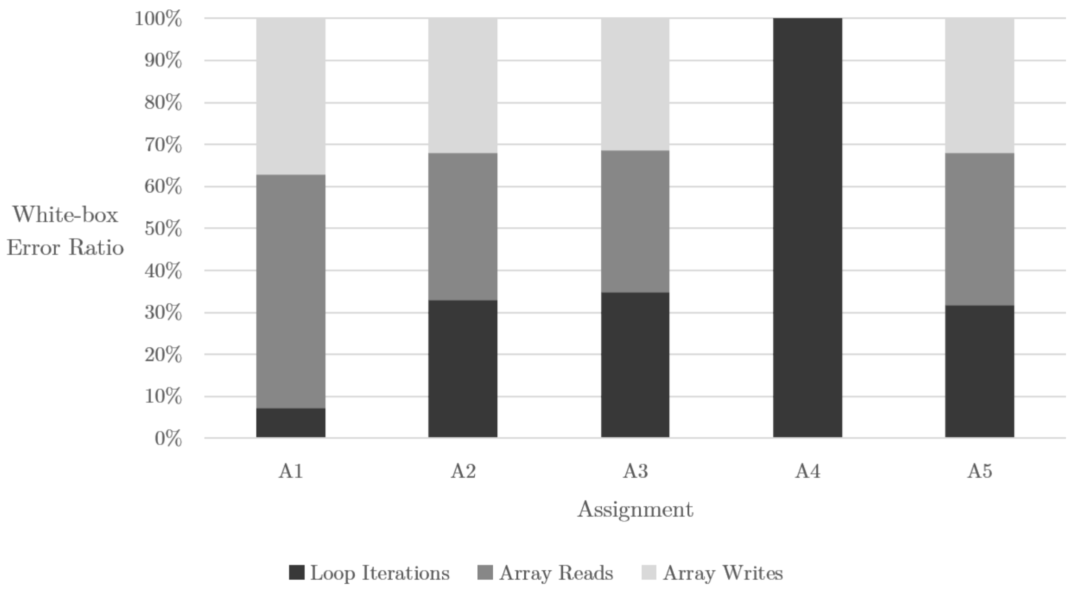

Figure 5 Ratio of each white-box metric failure per assignment.

read and write operations, the proportion of which are approximately equal, each constituting approximately one third of each assignment's detected errors. Assignment A4 produced the simplest results, with all failed assertions relating to the number of loop iterations.

Finally, our suspicion regarding code similarity was confirmed by JPlag's analysis, with the average similarity between submissions ranging approximately from 20% to 64%, as summarised in Table 4.

6 Discussion

The results of our evaluation show that a considerable number of students produce faulty implementations when it comes to algorithmic behaviour, which would go unnoticed by an evaluation process focusing only on the results produced by the students' code. Even in the assignment which contained the least faulty implementations (A4) 32 students could be misled by a black-box-only assessment process, a number which we argue is not negligible. Furthermore, in assignment A3 (lowest submission similarity), which required arguably the most originality in developing an alternative version of a standard algorithm, we saw a 44.5% failure rate, which constitutes nearly half of all considered submissions. We can

Table 4 Average submission code similarity per assignment.

Assignment	**Average ± Std. Dev.**
A1	64.2 ± 20.0%
A2	25.7 ± 17.5%
A3	19.8 ± 17.6%
A4	22.6 ± 14.0%
A5	44.1 ± 24.0%

thus conclude that the usage of an assessment tool which provides information about the execution of students' implementations is useful, not only from the perspective of instructors aiming at accurate grading, but also to prevent misleading students by informing them of the correctness of their implementation's results regardless of how those results were produced.

The current version of Witter successfully tackles the main limitations of our previous work [3], with the internal development of Strudel broadening the scope of supported language features and therefore that of supported assignments, coupled with the development of a DSL for test specification, allowing the specification of stateful tests for assignments using objects (e.g., for implementing data structures).

Our current work on Witter and Strudel allowed an acceptable coverage of standard Java constructs and language features, as seen by the overall 85.9% successful file loading rate seen during the evaluation phase. Nevertheless, further work should be carried out to extend the scope of supported functionalities and therefore enable the usage of more diverse assignments for a more detailed evaluation.

The average time taken to execute the evaluation of each submission reveals an acceptable performance for using Witter in educational contexts, with each submission taking from a few milliseconds to no more than one second to be evaluated. The execution time is dependent on the complexity of the specified tests, with longer or more comprehensive tests corresponding to a longer execution time.

Care should be taken in future work to tackle the limitations introduced by the characteristics of the dataset chosen for evaluation. For instance, taking Strudel's supported functionalities into account during the submission process could provide a larger usable dataset, enabling a more significant evaluation. Furthermore, a more unbiased evaluation could be carried out by utilising an external, publicly-available dataset of programming student submissions, guaranteeing the usage of code produced by students outside of our institution. Additionally, this could tackle possible issues of cheating or plagiarism, whose presence was made evident by our evaluation process and, as hypothesised, could skew the results by introducing a bias relating to students unknowingly plagiarising from incorrect sources. While an average code similarity between submissions of approximately 20%, as for assignment A3, can fall within reasonable expectations for an assignment of this scope, values of 44% or 61% average similarity, as seen for assignments A5 and A1, respectively, begin to raise concerns of considerable plagiarism and how it can affect the results of our analysis.

We continue to envision Witter as a tool not only aiming to be integrated into existing automated assessment systems as a way to extend the scope of their assessment functionalities, but also as courseware to be used by introductory programming students in a classroom environment, providing a learning process augmented through instant feedback on their attempts to solve programming exercises. To this effect, we envision the implementation of more high-level metrics, such as counting the number of array swap or move operations, which are standard when analysing programs from an algorithm complexity standpoint, and thus relevant for introductory programming classes focusing on algorithms.

While we have not yet been able to conduct a user study with programming students or instructors, this is likely to be our main focus in the future as the scope and stability of both Witter and Strudel's functionalities increase. Namely, a study with introductory programming students is necessary to gauge whether Witter is useful for its envisioned context, and a study with programming instructors can offer insight into the effort required to develop assignments adopting Witter's DSL. Finally, given a longer time frame, a study could be conducted in the context of an introductory programming course to analyse the long-term effect on the usage of Witter or similar assessment tools in students' learning outcomes.

References

1 C.L. Aasheim, Paige Rutner, L. Li, and S.R. Williams. Plagiarism and programming: A survey of student attitudes. *Journal of Information Systems Education*, 23:297–314, January 2012.

2 Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.5769`.

3 Afonso B. Caniço and André L. Santos. Witter: A library for white-box testing of introductory programming algorithms. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E*, SPLASH-E 2023, pages 69–74, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3622780.3623650`.

4 Shigeru Chiba. Load-time structural reflection in java. In Elisa Bertino, editor, *ECOOP 2000 — Object-Oriented Programming*, pages 313–336, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

5 Sébastien Combéfis. Automated code assessment for education: Review, classification and perspectives on techniques and tools. *Software*, 1(1):3–30, 2022. `doi:10.3390/software1010002`.

6 Pedro Crispim, Antónia Lopes, and Vasco T. Vasconcelos. Runtime verification for generic classes with congu2. In *Proceedings of the 13th Brazilian Conference on Formal Methods: Foundations and Applications*, SBMF'10, pages 33–48, Berlin, Heidelberg, 2010. Springer-Verlag.

7 Michael T. Helmick. Interface-based programming assignments and automatic grading of java programs. *SIGCSE Bull.*, 39(3):63–67, June 2007. `doi:10.1145/1269900.1268805`.

8 David Insa and Josep Silva. Automatic assessment of java code. *Computer Languages, Systems & Structures*, 53:59–72, 2018. `doi:10.1016/j.cl.2018.01.004`.

9 Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. A systematic literature review of automated feedback generation for programming exercises. *ACM Trans. Comput. Educ.*, 19(1), September 2018. `doi:10.1145/3231711`.

10 Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming*, pages 327–354, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

11 José Paulo Leal and Fernando Silva. Mooshak: a web-based multi-site programming contest system. *Software: Practice and Experience*, 33(6):567–581, 2003. `doi:10.1002/spe.522`.

12 Marcus Messer, Neil C. C. Brown, Michael Kölling, and Miaojing Shi. Automated grading and feedback tools for programming education: A systematic review. *ACM Trans. Comput. Educ.*, December 2023. Just Accepted. `doi:10.1145/3636515`.

13 Samim Mirhosseini, Austin Z. Henley, and Chris Parnin. What is your biggest pain point? an investigation of cs instructor obstacles, workarounds, and desires. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, pages 291–297, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3545945.3569816`.

14 José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. Automated assessment in computer science education: A state-of-the-art review. *ACM Trans. Comput. Educ.*, 22(3), June 2022. doi:10.1145/3513140.

15 Timur Sağlam, Moritz Brödel, Larissa Schmid, and Sebastian Hahner. Detecting automatic software plagiarism via token sequence normalization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3597503.3639192.

16 Timur Sağlam, Sebastian Hahner, Larissa Schmid, and Erik Burger. Obfuscation-resilient software plagiarism detection with jplag. In *46th IEEE/ACM International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion. Institute of Electrical and Electronics Engineers (IEEE), 2024. doi:10.1145/3639478.3643074.

17 Manuel Sánchez, Päivi Kinnunen, Cristóbal Flores, and J. Ángel Velázquez-Iturbide. Student perception and usage of an automated programming assessment tool. *Computers in Human Behavior*, 31:453–460, February 2014. doi:10.1016/j.chb.2013.04.001.

18 Anne Venables and Liz Haywood. Programming students need instant feedback! In *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20*, ACE '03, pages 267–272, AUS, 2003. Australian Computer Society, Inc.