Combining Dynamic and Static Analysis for the Automation of Grading Programming Exams

DOUWE OSINGA, University of Twente, The Netherlands

Manual grading of programming exams can take a significant amount of time and funding, and may additionally be subject to human variances in grading. Autograders can solve the aforementioned problems. However, there are currently no autograders available that utilise *both* a combination of dynamic and static analysis *and* that link test criteria to the Intended Learning Outcomes of a module. Therefore, we propose a theoretical autograding model that utilises the aforementioned analysis techniques and links test criteria to Intended Learning Outcomes. Additionally, we offer guidance for creating grading criteria based on Intended Learning Outcomes, and we demonstrate a proof-of-concept implementation of the aforementioned model, called Thoth. We verify Thoth by comparing the grading of a selection of exercises from an introductory programming exam. With this verification, we demonstrate the potential for autograders to aid in (partially) grading introductory programming exams.

Additional Key Words and Phrases: automated programming exam grading, autograder, static program analysis, dynamic program analysis, intended learning outcomes

1 INTRODUCTION

The manual grading of programming exams can take a significant amount of time and funds, and may additionally be subject to variances in grading. These variances in grading can result in students getting different grades, while their solutions should get the same grade, which is undesirable. This can occur due to implicit biases when grading students with a personal connection to the grader [13] or the varying interpretations, methodologies, and levels of competence between graders.

An autograder might solve some of the problems described above. However, there are some concerns surrounding autograders. This can include the possibility that autograders incorrectly grade exams due to improper test sets, or that autograders can potentially unfairly automatically correct submissions of students, in case platforms offer this functionality [2, 7, 20][1].

Aside from the aforementioned downsides, there can be tremendous benefit in using autograders, as they can reduce the required man-hours needed for grading and personal biases present in human grading [13]. Additionally, autograders can improve the accuracy of grading with respect to the Intended Learning Outcomes (ILOs) of the module, which is a set of sentences describing the skills "an instructor desires for students to gain from a course" [25]. This is of course only possible if the test criteria are properly linked to the ILOs of a module, which may not always be the case [5].

[1]The results of Bhatia & Signh (2016) in particular seem to unfairly overcorrect sometimes, particularly for the bottom two examples of Figure 1c [7, p.4]

TScIT 41, July 5, 2024, Enschede, The Netherlands

1.1 Research Questions

In order to be able to provide evidence of the effectiveness of autograders in introductory programming exams, a main research question (**RQ**) has been established:

> *To what extent can autograders replace human grading in the context of grading introductory programming exams?*

We aim to answer this research question by answering the following sub-research questions:

RQ 1: Which (combination of) program analysis methods is/are most effective for the grading of introductory programming exams?

RQ 2: To what extent can ILOs be translated into test criteria for autograders?

RQ 3: Which existing autograders (if any) are likely to be a good fit for grading introductory programming exams in terms of analysis methods, availability of source code, method of linking ILOs to test criteria, and ease of adoption by institutions?

RQ 4: How does an autograder that fits the criteria in **RQ3** compare to human grading in terms of consistency and fairness in the context of an introductory programming exam?

In section 2, we answer **RQ1** and **RQ3** by providing more information on the theory behind autograding, as well as explaining the techniques and limitations of a selection of autograders. In section 3, we discuss what steps were taken to complete this research. In order to answer **RQ2**, we provide guidance for the translation of ILOs into grading criteria in section 4.1, after which we present our autograder model in section 4.2, describe this model in section 4.3, and answer **RQ4** by verifying the implementation of the model in section 4.4. Afterwards, in section 5, we discuss how these results can be interpreted, after which we hypothesise what still needs to happen to make Thoth suitable for real exam grading in section 6. Finally, we conclude the research by answering the research questions in section 7.

2 RELATED WORKS

Automatic grading is based on the fundamental principles of program verification, which can be divided into static and dynamic analysis [6]. Dynamic analysis is the practise of running code against test cases in order to verify whether the program produces the correct output, while static analysis observes the *structure* of the code (without executing it) [6].

While dynamic analysis has several downsides, including (1) not terminating on programs that never finish executing, (2) possibly having to run it in a containerised or other virtual environment in the case of running unknown, possibly malicious programs, and (3) being unable to provide absolute truths about a program if it relies on external state (as it may therefore not produce the same result for subsequent executions). However, despite the downsides, dynamic analysis can provide a strong link between the inputs and outputs of a program [6].

In comparison, static analysis can provide true invariants of a program, but may (1) may suffer from abstraction issues[2] [6], and (2) may not finish executing within reasonable time for larger programs[3] [6]. Combining dynamic and static analysis can bring out the positives of both strategies, providing the best compromise in terms of speed and accuracy for the verification of programs.

Unfortunately, while there is a large body of knowledge concerning autograders, there seems to be quite a small number of papers regarding the translation of ILOs into testing criteria. In our search, we found exactly one paper, one from King & Duke-Williams [2001] that talks about verifying ILOs with automated tools, which sadly did not provide a methodology for translating ILOs [16].

2.1 Existing autograders and their limitations

Several solutions for automatically grading exercises or exams already exist. Some are closed-source [11, 20], while other platforms are open-source [10, 21, 29], and some are even in active use by universities [24]. There was also an autograder developed by a Master student at the University of Twente named APOLLO++ [22]. However, this solution was developed as a tool for grading programming *projects*, which have a far more loosely defined structure than exams[4], making this tool not easily adaptable to exam grading.

Roughly half of the discovered papers and `GitHub` repositories used purely dynamic analysis [10, 20, 21, 29]. However, a decent number of autograders opted for a combination of dynamic and static analysis [12, 17, 26, 27]: some use linting[5] to check for code quality or style [17], while others expand upon the concept and evaluate the control structures of functions, either for comparing it to a reference implementation [18] or for partitioning the code into sections in order to more effectively verify submissions [27].

However, there turned out to be very little papers on autograders that also have published their solutions online (see appendix B). Notably, of all the solutions, only ArTEMiS [17] had both a published paper *and* source code available on GitHub [24], and of all publicly available autograders (not necessarily with a corresponding paper), only ArTEMiS and CodeGrade [11, 17] utilised dynamic *and* static analysis. See appendix B for more information on the found autograders.

[2]Some parsers may leave out some of the details of the language, as a consequence abstracting away details that could have been useful for spotting certain issues in code.
[3]The definition of 'reasonable time' or 'larger programs' depends on the speed of the static analyser and the desired maximum run time.
[4]The author described their tool as being "flexible and supporting".
[5]Linting is the practise of enforcing coding standards, finding unused variables and spotting common bugs in code [1]

Because the autograder is intended to be used for grading exams, we believe it is vital to know the inner workings of the software in order to ensure that the product uses the proper analysis techniques (see section 2) and that the product behaves exactly as intended. With this restriction, the selection of available platforms becomes quite small: only ArTEMiS and CodeGrade have the desired analysis methods, and CodeGrade is closed-source, leaving only ArTEMiS as a potentially usable solution. Unfortunately, ArTEMiS is primarily a Learning Management System (LMS), containing much more functionality than needed, which would force the UT to transfer over all student accounts, which does not aid in the ease of adoption. The aforementioned issues, along with having some unfortunate technical issues[6] make ArTEMiS unattractive to adopt for autograding.

In conclusion, every existing platform discovered (appendix B) has some number of (technical) issues or shortcomings that make it unattractive for our use case. Additionally, none of the systems showed any functionality to connect test criteria to ILOs, and there were no papers found that explicitly discussed the translation of ILOs into test criteria.

3 METHODOLOGY

In order to answer the main research question, we performed a limited-scope literature study in order to obtain information on the current state of the art in terms of program analysis methods and autograders, with terms such as "dynamic analysis program verification", "static analysis program verification", and "automatic grading programming university", "autograder programming", "automated grading programming" respectively. Searches for papers and/or autograders were performed on `Google Scholar`, `Inciteful.xyz`, `essay.utwente.nl`, and `GitHub`.

The translation of ILOs has been researched separately, using search terms such as "ILO translation to test cases", "intended learning outcomes program verification", and specifically "Anderson and Krathwohl 2001", as they revised Bloom's taxonomy, which resulted in the King & Duke-Williams source [16]. Searches were performed on the same platforms as the searches for autograders and program analysis techniques, excluding `GitHub`.

For developing the autograding model, the dataset was inspected to form a general idea of the structure and format of the exam, which formed the basis of the model (see section 4.2).

For developing the proof-of-concept implementation (Thoth, section 4.3), the autograding model was taken as a basis, and extended and refined to support additional features such as importing datasets.

[6]The repository fails to compile natively on Windows due to some of the file path names being longer than the Windows path character limit, therefore limiting ease of development and maintainability

4 RESULTS

4.1 ILO translation into grading criteria

As mentioned in section 1, Intended Learning Outcomes (ILOs) are a set of sentences describing the skills "an instructor desires for students to gain from a course" [25]. There are a few methods of generating ILOs, including (1) deriving them based on the verbs in questions, (2) extrapolating them from existing exam questions, and (3) by using examples. These methods, however, all have various downsides, which can result in "misleading" and "unclear" ILOs [16, p.4]. Therefore, a structural approach for designing ILOs is key to generating clear, specific ILOs.

There exist a few popular methods to do this, most prominently the Structure of Observed Learning Outcomes (SOLO) [8] and Bloom's Revised Taxonomy [4, 28]. For this paper, we will focus on Bloom's Revised Taxonomy, as it, unlike SOLO, concentrates on the depth of knowledge [3, p.158], and because it seems to offer more granular levels of understanding compared to SOLO: SOLO offers the levels "Unistructural", "Multistructural", "Relational", and "Extended Abstract" while Bloom's Revised Taxonomy offers the levels "Remembering", "Understanding", "Applying", "Analysing", "Evaluating", and "Creating" [9, p.6-7].

Next to offering information about which levels of knowledge they address, ILOs should also indicate which kind of knowledge they relate to. Biggs [2011] claims there are two kinds of knowledge: Declarative and Functioning knowledge [9]. Other sources, such as King & Duke-Williams [2001], split the kinds of knowledge up into four categories: Factual, Conceptual, Procedural, and Metacognitive [16]. An extensive list of examples of Intended Learning Outcomes can be found in Kennedy [2006] [15, p.32][7]. For the purposes of this research, we will be using the four kinds of knowledge from King & Duke-Williams [2001].

4.1.1 *Relation to introductory programming exams.*

Introductory programming exams, however, often do not test all levels and kinds of knowledge. Simon et al. [2012] states that in the observed programming exams, the five code-related skills they defined (writing, tracing, explaining, debugging, and modifying code) covered 81% of exam questions, "the remainder being taken by knowledge recall (10%), design (7%) and (...) testing [(2%)]" (see figure 3a and 3b) [23, p.66]. For ILOs, this suggests that the vast majority of introductory programming questions are focused on the levels of knowledge related to "Analysing" (tracing), "Evaluating" (debugging), and "Creating" (modifying, writing) (see appendix C) [4], and on the kinds of knowledge related to "Functioning" knowledge [9] or "Procedural" knowledge (knowing how and where to utilise algorithms, techniques and methods) [16]. Therefore, we will only offer advice for translating ILOs for this subset of levels and kinds of knowledge.

4.1.2 *Guidance on translating ILOs.*

[7]Do keep in mind that these learning outcomes have been written with Bloom's original taxonomy, and not the revised version from Anderson & Krahtwohl [2001] [4]

```
public static ArrayList<Integer> getDiff(
    int[] arr1, int[] arr2) {
    //TO-DO: implement here.
}
```

Fig. 1. An example question from the dataset.
"This method receives two int[] as argument and calculates the differences between each position of both arrays, i.e., arr1[0] - arr2[0]. (...) The method returns an ArrayList with the differences between each position of both arrays. (...) In case the two arrays are not of the same length, this method throws an IllegalArgumentException (...)". ILOs tested: 1,2,3,4,6 (see appendix F).

In order to translate the ILOs of an introductory programming exam into testable criteria, it can help to first imagine situations where this ILO is being applied. For example, one of the ILOs of the dataset states that a student should be able to "Express algorithmic solutions that use repetition structures", i.e., a student should be able to use `for` and/or `while` loops (ILO 4 in appendix F).

After having established the situation where this ILO is applicable, one can start thinking of example questions that require the student to create a program that works within this situation.
For the aforementioned ILO, an exercise can be used such as the example in figure 1. Additional hints or requirements may be given to ensure that a student uses the intended data structures or language concepts ("Use one `for`-loop", "Do not make use of the `Set` class", or "Throw an `IllegalArgumentException` when (...)").

After an exercise has been made, one should verify if other ILOs are (indirectly) being tested by this exercise. In the case of the second example, primitive datatypes could also be tested (if the student uses an index-based for-loop), complex datatypes are tested (arrays are manipulated) and basic arithmetic is tested (addition, for the for-loop and the counter). The final list of ILOs of this example exercise can be found in the description of figure 1.

After all the ILOs in an exercise have been identified, one can start to make criteria that test these ILOs. In the example of the 'repetition' ILO, one can make dynamic tests that test that the output of a submission matches for some inputs. Additionally, one can generate static tests that verify whether a student in fact used a `for` or `while` loop to perform this task. Ideally, each criterion should be tested by means of a combination of dynamic and static criteria, to ensure that the behaviour *and* internals of the function are sound in terms of completeness, scope and precision [6, p.217].

With this guidance, we aim to improve the quality of the questions asked and the grading criteria used in exams, in order to increase the potential effectiveness of automatic grading.

4.2 Autograding model

We have developed a simplified model for autograding introductory programming exams, which can be viewed in figure 4.

4.2.1 *Explanation of model.*

The model is split into various blocks: the **Submission model**, the **Correction model**, the **Grading model** and the **Result model**.

The **Submission model** determines in which format an exercise is presented. The **Correction model** represents the grading criteria to use for a particular exercise, which contains the dynamic and static test suites used for grading. Each dynamic test suite is a collection of tests to run for a particular method, whereas each static test suite is a collection of static criteria the submission code must reach, such as having certain function calls or using certain data structures. We enforce the structure of $ILO \rightarrow TestSuites \rightarrow TestCases$ because we believe this forces teachers to think explicitly about which ILO their tests relate to. Furthermore, the aforementioned structure requires test cases to be in test suites. This allows for linking a test suite to a particular (piece of a) grading criterion, which can be verified with the test cases in that test suite.

The **Grader model** should take in the aforementioned Correction and Submission model and grade the Submission based on the Correction model. After the grader has finished grading, it should produce a **Result model**. This model contains the necessary information to convey which grade was achieved, along with which scores were achieved for which ILO, for which specific test suite within that ILO and for each test case within that test suite.

4.3 Implementation of model: Thoth

The autograding model (discussed in section 4.2) has been implemented in a command-line utility called Thoth [19], which specialises in grading single-function Java questions.

4.3.1 *Architecture.*

For testing and evaluation, we implemented the described model, adding features for importing/exporting the dataset and serialising configurations. Additionally, we decided that, for Thoth, both dynamic and static test suites only award full points if all test cases pass, and no points otherwise. This was chosen to force a test suite to test only one concept, and only award points if all the desired behaviour is achieved. If partial points are desired, one can split up the test suite, with each fragmented test suite awarding partial points.

4.3.2 *Dynamic Analysis.*

Before any dynamic analysis is performed, the submissions are parsed by the `JavaParser` library into an Abstract Syntax Tree (AST) format. This AST is then formatted to suit dynamic testing[8]. After the solution is formatted, it is converted back into a String, after which they are compiled into memory using a custom `InMemoryJavaCompiler`. After the submission is compiled, its class is loaded into the program, the method to test is looked up, and test cases are ran on this method.

4.3.3 *Static Analysis.*

Static analysis takes the parsed ASTs of the students (as described in section 4.3.2) before they are formatted. It runs these submission ASTs through several "Criteria Verifiers", which internally traverse the AST to assert whether the solution uses certain control structures, or whether a solution uses certain (argument or object-specific) function calls.

[8]If the submission contains only a function it is placed in a class, class names are changed to be a constant name, and imports are reduced to an allowed selection.

4.4 Verification

For verifying our autograding solution, we use an anonymised dataset from a first-year Business and IT exam given at the University of Twente.

4.4.1 *Format of exam.*

The dataset we use is similar to exam papers 15, 17, and 19 from figure 3a, as it only expects answers containing code fragments. The exam is divided into three distinct sections, each related to a level of competence: "Entry level", "Intermediate Level", and "Target level". The Entry Level section tests basic concepts such as primitive datatype transformations, "sequence and selective structures" (e.g., `if` and `switch` structures), and exceptions. The Intermediate Level section builds on these fundamentals, introducing loops (`for`/`while`) and linear data structures (e.g. `Lists`). The Target Level section expands upon this by testing more complex, non-linear data structures (e.g. nested `Lists`, `Maps`, or multiple types of data structures in the same exercise).

4.4.2 *ILOs and grading rubrics.*

Luckily, the ILOs of the module were constructed with Bloom's revised taxonomy in mind, which was apparent from the "Bloom's level" tags above each question's ILOs. Additionally, each question had a list of ILOs that question aimed to test, which made it much easier to connect the grading criteria with an ILO.

In the end, the original grading criteria was split up and connected to ILOs, which formed the basis of the testing criteria for the verification.

4.4.3 *Verification of model.*

In order to verify the model, we compared the manual grading of a selection of questions from the dataset with the automatic grading done by Thoth. For the verification, we decided to focus on questions asking individual functions (questions 1-7), instead of Object-Oriented Programming (OOP)-related questions (questions 8-10), as testing OOP concepts requires more advanced static analysis, which would have put the already strict time constraints under even more stress.

For the Beginner Level, we evaluated the grading of exercise 2 (`notString`). We did not evaluate exercise 1 (`countDigits`) as we believe it has a more limited set of possible answers than question 2, and we want to test out the autograder for a more wide range of possible answers. Additionally, if we successfully grade question 2, this means that question 1 would probably also show good results. We also did not evaluate exercise 3 (`weekDayName`), as it aims to test conditionals[9], but can also be solved by other methods (e.g. array indexing), which does not test all ILOs (see appendix F). For the Intermediate Level, we evaluated the grading of exercise 5 (`countCommonWords`). This is partly due to the poor compile rate of exercise 4, which would not yield statistically robust results (only 17 submissions compiled out of 109, see appendix F), and because this

[9]This aim is demonstrated by the grading rubric "1.5 - points: for correctly travers[ing] through each "case" in [the] Switch statement (if sta[te]ment[s] [are] also possible).". Taken from the official grading rubric, edited with brackets ([]) to ensure coherent English.

exercise involves a bit more complexity than exercise 6 (which was about subtracting integers in arrays). For the Target Level, there is only one question (number 7, "Reverse Map"), which we will be evaluating in addition to the two previous exercises.

In order to set up the grading configurations for Thoth, we followed the official grading rubric of the exam quite closely. We either copied over the example tests in the exercise, and the points in the rubric directly, or we split up rubric points into multiple smaller test suites, in order to give partial points.

We manually corrected some function names and access modifiers of submissions that incorrectly spelled the names of functions to test (e.g. "`countWords`" instead of "`countCommonWords`") or inputted the wrong function modifiers ("`private static String notString`" instead of "`public static String notString`"). Apart from this, no other modifications were made to the dataset.

5 DISCUSSION

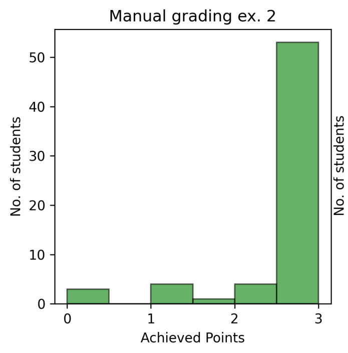
5.1 General Discussion

After running the autograder for each exercise with some corrected submissions (section 4.4) with a grading configuration constructed according to section 4.4.3, we can see in figure 2 that the configurations made for exercise 2 and 7 resemble the human grading fairly accurately. Unfortunately, the configuration made for exercise 5 fails to resemble human grading, either awarding full points for correct solutions or 'pity points'[10], possibly even for solutions that would be worth more than 0.5 points. This is likely due to the fact that the grading configuration contained quite a bit of dynamic testing suites, which tend to fail in large numbers for small mistakes in the solution. We could have improved the configuration of exercise 5, however, we believe that showing the effects of a less-than-ideal grading configuration is just as important as highlighting the benefits of a good grading configuration.

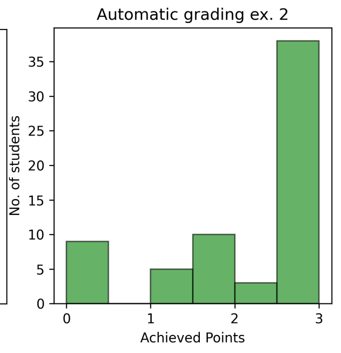
In general, though, these results should be taken with a grain of salt because, as long as the autograder is working as intended, its grading performance is entirely dependent on the configuration put into it. Additionally, the manual grading in this dataset seems to differ slightly from the grading rubrics, meaning that some liberties were taken by graders in grading the submissions. On top of that, the graphs for autograding are slightly skewed to the negative side figure 2, as autograded solutions that parse but fail to compile are also included[11].

In conclusion, while the performance of the autograder is highly dependent on the configuration (as demonstrated in figure 2c and figure 2d), and the efficiency gains entirely depend on how many parsing and compiling solutions are handed in by students, an autograder such as Thoth can have the potential to grade faster[12] and more consistently than human grading.

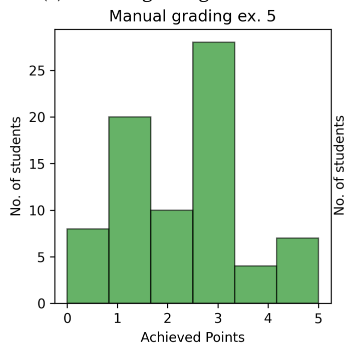
[10]The static analysis for this exercise is configured to award 0.5 points for using a `for` or `while` loop.
[11]This includes submissions such as one from exercise 5 that received 5 points by a TA, but that used a `HashSet`, which at the time of writing the autograder was not included in the 'add import' list, which triggered a compile failure.
[12]The observed grading times were in the order of seconds for questions 2 and 5, and roughly 1.5 minutes for question 7.



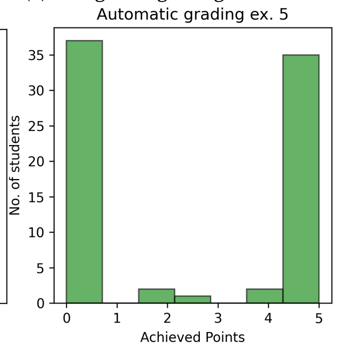
(a) Manual grading distr. ex. 2



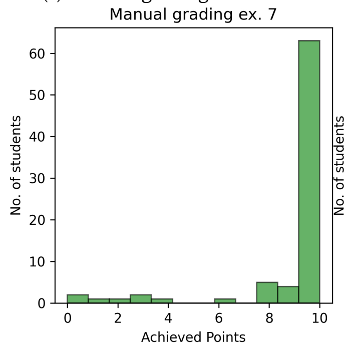
(b) Autograder grading distr. ex. 2



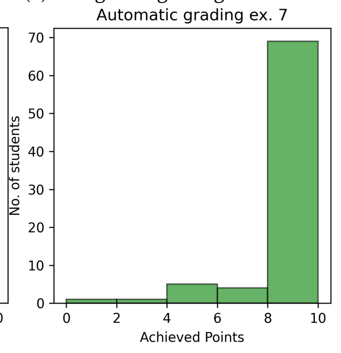
(c) Manual grading distr. ex. 5



(d) Autograder grading distr. ex. 5



(e) Manual grading distr. ex. 7



(f) Autograder grading distr. ex. 7

Fig. 2. Grade distributions for manual grading (left) and automatic grading (right) of exercises 2, 5, and 7, for all *parsing* solutions of candidates. Solutions that did not parse were filtered out for the manual grading graphs, and empty solutions were not considered in grading.

5.2 Need for training

Ideally for autograding, questions of the exam should be made by translating the ILOs into questions, which then can receive grading criteria (section 4.1). Additionally, implementing autograding requires filling in detailed configurations in order to make it accurately award points based on the rubric (with partial points in mind), both of which are non-trivial tasks[13].

Therefore, we firmly believe there should be required training for teachers to, firstly, know about how to translate ILOs into grading

[13]Translating the ILOs is hard as demonstrated by section 4.1. The level of difficulty of entering grading configurations comes from personal experience

criteria and, secondly, know what features the autograder has, so that they can create criteria effectively, and configure the grading in such a way that the autograder grades in a nuanced and forgiving way. Unfortunately, existing methodology is (as far as was discovered) non-existent, and our own methodology for this (see section 4.1) is far from complete, meaning that effort should also be spent in this area, in order to develop more robust strategies for creating the rubrics.

5.3 Caveats while implementing Thoth

During development, a number of issues popped up:

- Not every student submission parsed successfully, and compile rates were even lower (see appendix F). This was surprising to see, as students were provided with an IDE while taking the exam. Unfortunately, these solutions will have to be corrected manually, as no automatic (syntax) correction is implemented. Parse and compile rates could be improved in future exams by emphasising that student submissions must contain valid Java code.
- Not every submissions was purely a function: some students wrapped their functions in classes (with varying names). This issue was resolved by including a `ClassFormatter` that wraps function definitions in classes and renames classes to a known name. Clearer instructions in the exam could be provided about handing in only a function or class definition, however, we believe this also places an additional unnecessary cognitive load on the student, as formatting is automatically performed anyway.
- Some solutions used Java data structures without importing them. This was remedied (but not completely resolved) by implementing an `ImportFormatter` which removes all imports and adds a limited set of allowed imports to each submission[14]. While one could argue that students should have put these imports into the code themselves, we believe that it is be beneficial for security to not allow arbitrary student imports, and instead override the students imports with a known set of imports.

5.4 Limitations of Thoth

While Thoth works well as a proof-of-concept, it is missing some features required to make it ready for real exams. In particular, the following items could use some work:

(1) OOP concepts are currently not able to be dynamically tested by Thoth, as the dynamic analyser is fixed to only accept one class or function definition. This could be fixed by adjusting the dynamic analyser to compile multiple classes for OOP questions.

(2) Inputting exam configurations is currently done by manually editing a JSON file, which scales horribly[15] for more than a couple of questions. Therefore, developing a visual tool would aid in being able to test more exercises or larger test sets.

(3) Thoth does not containerise dynamic analysis in any way. This means that Thoth compiles and runs student code directly on the host machine. This allows students to write malicious code that would execute on the grader's machine[16] and potentially cause harm. For real exams, dynamic analysis should be containerised or virtualised.

(4) Thoth currently only adds a few predefined imports to submissions. This list is incomplete and should be extended with the most common Java data structures (`HashMap`, `LinkedList`, etc.).

6 FUTURE WORK

For future research, the following two items could be of interest:

(1) Improve Thoth with help of the limitations defined in section 5.4, and test it 'in the field' in an introductory programming exam, or a practice exam.

(2) Improve upon the methodology defined in section 4.1 for translating ILOs into testable criteria, in order to help improve the quality of ILOs in modules and the effort needed to switch from human grading to autograding.

7 CONCLUSION

In this research, we discussed some theory behind program analysis in order to determine the most effective combination of analysis methods, which turned out to be a combination of dynamic and static analysis (**RQ1**). We then inspected a selection of autograders, and discovered that most autograders either only exist in theory, or exist in practise but either (1) do not utilise the combination of analysis techniques we desire, (2) are closed-source, (3) are not well-documented and/or not actively maintained, or (4) are part of bigger systems (**RQ3**). Additionally, none of the solutions contained features to help link ILOs to test cases (**RQ2**). We therefore decided to make our own model for autograding, along with a proof-of-concept implementation called Thoth. After additionally discovering that there was no literature available for writing ILO-based test criteria, we created a set of guidelines for translating ILOs into test criteria based on Bloom's (Revised) Taxonomy and related works. We found that with questions that are tightly linked to ILOs and a properly made grading configuration, there is real potential for autograders to help in grading exams, thereby reducing the time it takes to grade exams and reducing human biases and mistakes made by manual grading (**RQ4**).

ACKNOWLEDGMENTS

I would like to thank Leon de Vries for providing support and assistance during my research, Wallace Corbo Ugulino for kindly providing the dataset and assisting in the supervision, and Sarah Holtrop for always being there for me, also during these interesting, but intense weeks.

[14]The imports are currently limited to Java native libraries.
[15]The configuration file grew to roughly 500 lines for the verification of exercises 2, 5, and 7.
[16]Given that the student code does not import libraries that Thoth removes.

REFERENCES

[1] 2023. SublimeLinter — SublimeLinter 3.4.24 documentation. http://web.archive.org/web/20230606032819/http://www.sublimelinter.com/en/v3.10.10/about.html

[2] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. (TRACER) Compilation error repair: for the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET '18)*. Association for Computing Machinery, New York, NY, USA, 78–87. https://doi.org/10.1145/3183377.3183383

[3] Satu Alaoutinen and Kari Smolander. 2010. Student self-assessment in a programming course using bloom's revised taxonomy. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education (ITiCSE '10)*. Association for Computing Machinery, New York, NY, USA, 155–159. https://doi.org/10.1145/1822090.1822135

[4] LW Anderson and DR Krahtwohl. 2001. *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives: complete edition.* Addison-Wesley. https://eduq.info/xmlui/handle/11515/18824

[5] Simon Paul Atkinson. 2015. Graduate Competencies, Employability and Educational Taxonomies: Critique of Intended Learning Outcomes. *Practice and Evidence of the Scholarship of Teaching and Learning in Higher Education* 10, 2 (July 2015), 154–177. https://www.pestlhe.org/index.php/pestlhe/article/view/104

[6] Thomas Ball. 1999. The Concept of Dynamic Analysis. *ACM SIGSOFT Software Engineering Notes* (1999). https://doi.org/10.1145/318774.318944

[7] Sahil Bhatia and Rishabh Singh. 2016. Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks. https://doi.org/10.48550/arXiv.1603.06129 arXiv:1603.06129 [cs].

[8] John Biggs and Kevin Collis. 1989. Towards a Model of School-based Curriculum Development and Assessment Using the SOLO Taxonomy. *Australian Journal of Education* 33, 2 (Aug. 1989), 151–163. https://doi.org/10.1177/168781408903300205 Publisher: SAGE Publications Ltd.

[9] John Biggs and Catherine Tang. 2011. Train-the-Trainers: Implementing Outcomes-based Teaching and Learning in Malaysian Higher Education. *Malaysian Journal of Learning and Instruction* 8 (2011). https://doi.org/10.32890/mjli.8.2011.7624

[10] codeboard.io. 2015. CodeBoard.io - Github. https://github.com/codeboardio

[11] CodeGrade. 2017. CodeGrade. https://www.codegrade.com

[12] Michael T. Helmick. 2007. Interface-based programming assignments and automatic grading of java programs. *ACM SIGCSE Bulletin* 39, 3 (June 2007), 63–67. https://doi.org/10.1145/1269900.1268805

[13] John M Malouff and Einar B Thorsteinsson. 2016. Bias in grading: A meta-analysis of experimental research findings. https://journals.sagepub.com/doi/10.1177/0004944116664618

[14] Arelia Jones and Github. 2020. Introducing autograding for GitHub Classroom and the GitHub Teacher Toolbox. https://github.blog/2020-03-12-github-teacher-toolbox-and-classroom-with-autograding/

[15] Declan Kennedy. 2006. *Writing and using learning outcomes: a practical guide.* University College Cork. https://hdl.handle.net/10468/1613

[16] Terry King and Emma Duke-Williams. 2001. Using Computer-Aided Assessment to Test Higher Level Learning Outcomes. (2001). https://repository.lboro.ac.uk/articles/Using_computer-aided_assessment_to_test_higher_level_learning_outcomes/9490043/files/17115707.pdf

[17] Stephan Krusche and Andreas Seitz. 2018. ArTEMiS: An Automatic Assessment Management System for Interactive Learning. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, Baltimore Maryland USA, 284–289. https://doi.org/10.1145/3159450.3159602

[18] Xiao Liu, Shuai Wang, Pei Wang, and Dinghao Wu. 2019. Automatic Grading of Programming Assignments: An Approach Based on Formal Semantics. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, Montreal, QC, Canada, 126–137. https://doi.org/10.1109/ICSE-SEET.2019.00022

[19] D. Osinga. 2024. osingaatje/Thoth-Autograder-Public. https://github.com/osingaatje/Thoth-Autograder-Public original-date: 2024-06-20T15:11:37Z.

[20] Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. 2017. (GradeIT) Automatic Grading and Feedback using Program Repair for Introductory Programming Courses. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17)*. Association for Computing Machinery, New York, NY, USA, 92–97. https://doi.org/10.1145/3059009.3059026

[21] James Perretta. 2020. AutoGrader.io - Github. https://github.com/eecs-autograder

[22] Arthur Rump. 2023. Apollo++: Automated Assessment of Learning Outcomes in Programming Projects. (Nov. 2023). https://essay.utwente.nl/97567/

[23] Simon, Judy Sheard, Angela Carbone, Donald Chinn, Mikko-Jusi Laakso, Tony Clear, Michael de Raadt, Daryl D'Souza, Raymond Lister, Anne Philpott, James Skene, and Geoff Warburton. 2012. Introductory programming: examining the exams. *Proceedings of the Fourteenth Australasian Computing Education Conference* (2012), 61–70. https://nova.newcastle.edu.au/vital/access/services/Download/uon:23938/ATTACHMENT02

[24] TUM Applied Software Engineering. 2016. Artemis - Github. https://github.com/ls1intum/Artemis#universities--schools-with-artemis-in-use

[25] University of Yale. 2017. Writing Intended Learning Outcomes. https://poorvucenter.yale.edu/IntendedLearningOutcomes

[26] Milena Vujošević-Janičić, Mladen Nikolić, Dušan Tošić, and Viktor Kuncak. 2013. Software verification and graph similarity for automated evaluation of students' assignments. *Information and Software Technology* 55, 6 (June 2013), 1004–1016. https://doi.org/10.1016/j.infsof.2012.12.005

[27] Jinshui Wang, Yunpeng Zhao, Zhengyi Tang, and Zhenchang Xing. 2020. Combining Dynamic and Static Analysis for Automated Grading SQL Statements. *Journal of Network Intelligence* 5, 4 (2020). https://doi.org/10.36108/jni.2020.5.4.001

[28] Leslie Owen Wilson. 2016. Bloom's Taxonomy Revised. *Quincy College* (2016), 1–7. https://www.quincycollege.edu/wp-content/uploads/Anderson-and-Krathwohl_Revised-Blooms-Taxonomy.pdf

[29] Stanislav Zmiev. 2020. zmievsa/autograder. https://github.com/zmievsa/autograder

APPENDICES

A AI STATEMENT

During the preparation of this work, the author used no Artificial Intelligence tools.

B AUTOGRADER OVERVIEW

Name	Has paper (Y/N)	Available (Y/N)	Open-Source (Y/N)	Technique (D/S)
ArTEMiS [17]	Y	Y	Y	D S
GradeIT [20]	Y	N	?	D
Wang et al. [27]	Y	N	?	D S
Liu et al. [18]	Y	N	?	S
Helmick [12]	Y	N	?	D
Vujošević-Janičić et al. [26]	Y	N	?	D S
Github Teacher Toolbox* [14]	N	Y	N	D
Zmiev Autograder [29]	N	Y	Y	D
EECS Autograder [21]	N	Y	Y	D
CodeIO CodeBoard** [10]	N	Y	Y	D
CodeGrade*** [11]	N	Y	N	D S

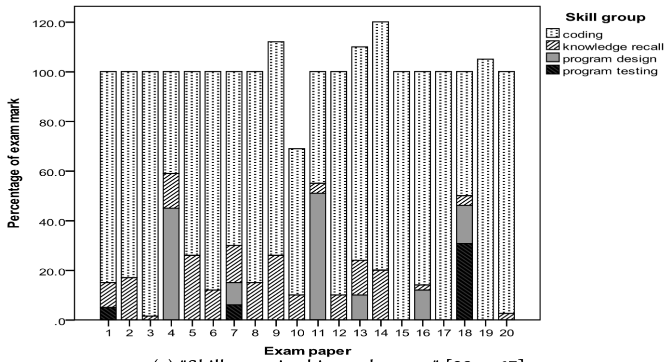
Table 1. An overview of all found autograders and their details: Do they have a paper? Can they be found online? Are they open-source? Which techniques do they utilise (dynamic analysis (D), static analysis (S), both (DS))?
* = Requires GitHub Teacher
** = Not well-maintained (>1 year inactive)
*** = Requires subscription

C BLOOM'S REVISED TAXONOMY

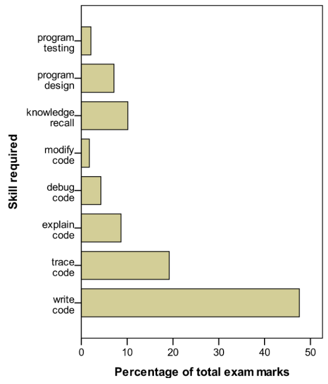
Cognitive Level	Verbs
Remembering	define, describe, draw, find, identify, label, list, match, name, quote, recall, recite, tell, write
Understanding	classify, compare, exemplify, conclude, demonstrate, discuss, explain, identify, illustrate, interpret, paraphrase, predict, report
Applying	apply, change, choose, compute, dramatize, implement, interview, prepare, produce, role-play, select, show, transfer, use
Analyzing	analyze, characterize, classify, compare, contrast, debate, deconstruct, deduce, differentiate, discriminate, distinguish, examine, outline, relate, research, separate, organize, structure
Evaluating	appraise, argue, assess, choose, conclude, criticize, decide, evaluate, judge, justify, predict, prioritize, prove, rank, rate, select, monitor
Creating	construct, design, generate, hypothesize, invent, plan, produce, compose, create, invent, make, perform, plan, produce, design, develop

Table 2. Bloom's Revised Taxonomy for ILOs [9, p.7]

D SKILLS REQUIRED IN PROGRAMMING EXAMS



(a) "Skills required in each exam" [23, p.67]



(b) "Skills required to answer questions" [23, p.66]

Fig. 3. Skills required for programming exams, according to a study by Simon et al. [2012] [23, p.66-67].

E MODEL DIAGRAM

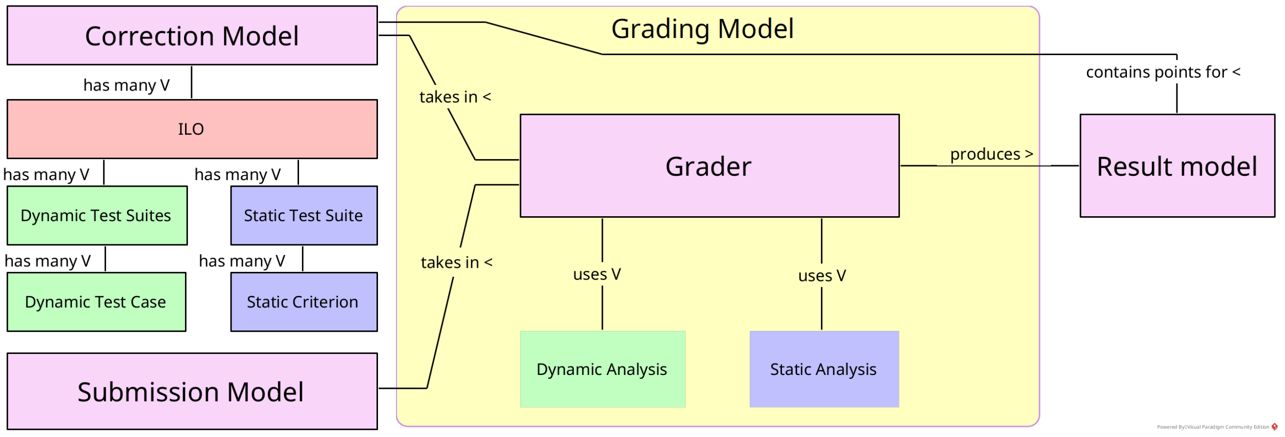


Fig. 4. Abstract model of the autograding model

F DATASET EXERCISES

ILO Number	ILO Description	Knowledge Level	Knowledge Kind
1	Select and use primitive datatypes	Applying	Procedural
2	Develop data transformation statements for primitive datatypes using appropriate operators	Applying	Procedural
3	Express sequence and selection structures (conditionals)	Applying	Procedural
4	Express repetition structures (loops)	Applying	Procedural
5	Express unexpected circumstances in execution flow such as exceptions	Applying	Procedural
6	Select and use linear data structures	Applying	Procedural
7	Select and use non-linear data structures	Applying	Procedural

Table 3. Levels, submissions counts, parse rates and compilation rates for the non-OOP exercises of the data set (1-7). The percentage behind the parse and compilation metrics refer to the fraction of the total submission count (e.g., Total Compiled = 75% means that three quarters of all submissions compiled successfully).

Level	Ex. Num.	Submissions*	Total Parsed	Total Compiled	Explanation	ILOs tested
ENTRY	1	111	84 (~76%)	69 (~62%)	`int countDigits(int number)`: returns number of digits of a number	1,2
ENTRY	2	114	65 (~57%)	57 (~50%)	`String notString(String s)`: Returns `s` if `s` begins with `"not "` (case-insensitively), otherwise `"not " + s`	1,2,3
ENTRY	3	188	63 (~34%)	47 (~25%)	`String weekDayName(int weekDay)`: Returns day of the week (1: Sunday, 2: Monday, ...)	1,2,3,5
INTERMEDIATE	4	109	26 (~24%)	17 (~16%)	`double similarityIndex(String s1, String s2)`: returns percentage of matching `Char`s of `String`s at the same index	1,2,3,4,6
INTERMEDIATE	5	111	77 (~69%)	47 (~42%)	`int countCommonWords(String s1, String s2)`: counts matching words in two `String`s (case-insensitively).	1,2,3,4,6
INTERMEDIATE	6	109	84 (~77%)	80 (~73%)	`List<Integer> getDiff(int[] arr1, int[] arr2)`: calculates `arr1[i]` - `arr2[i]` for each array index.	1,2,3,4,6
TARGET	7	107	80 (~75%)	72 (~67%)	`Map<String, String> reverseMap(Map<String, String> map)`: Swaps around the keys and values of a map.	1,2,3,4,5,6,7

Table 4. Levels, submissions counts, parse rates and compilation rates for the non-OOP exercises of the data set (1-7). The percentage behind the parse and compilation metrics refer to the fraction of the total submission count (e.g., Total Compiled = 75% means that three quarters of all submissions compiled successfully).
* = only counting non-empty submissions