WIP: Leveraging Static Code Analysis for Peer Code Review in Computer Science Education

Raz Landau
Blavatnik School of Computer Science
Tel Aviv University
Tel Aviv, Israel
razlandau@tauex.tau.ac.il

Dr. Amir Rubinstein
Blavatnik School of Computer Science
Tel Aviv University
Tel Aviv, Israel
amirr@tau.ac.il

***Abstract*—In this work-in-progress innovative practice paper we make - arguably for the first time - an effort for combining together two rich bodies of knowledge that have so far coexisted independently in computer science education research: static code analysis and peer code review. We achieve this by putting a sophisticated static code analysis system to work in a peer code review process implemented in an introductory computer science course for (1) matching reviewers based on the heterogeneity of their initial submissions and (2) analyzing the subsequent effects on resubmissions. We demonstrate the potential value of such usage and discuss the surprising trends we discovered, as well as broader ideas on leveraging static code analysis for peer code review in computer science education.**

I. Introduction

In computer science (CS) education, peer reviews that involve students assessing one another's code, termed *peer code review (PCR)*, have become a popular teaching tool, offering multiple benefits to both students and teachers [1]. Over the years, numerous studies have implemented PCRs in educational contexts and addressed various issues, from choosing when to assign reviewers [2] to the impacts of gamification [3]. Despite this, very little work has taken advantage of the rich knowledge in source code analysis – commonly known as *static code analysis (SCA)* – for analyzing the submissions throughout the process. In particular, not a single work out of 51 recently reviewed [4] has used SCA in its PCR implementation.

SCA, however, has much to offer to PCR. Modern SCA techniques are able to detect faults and errors, evaluate programming styles, highlight common misconceptions, verify understanding of specific concepts – and more [5]. While most initiatives in CS education focus on using SCA to deliver formative feedback to students, other applications have also started to emerge, urging to use it for additional benefits [6].

In this work, we answer this call by integrating Sense [7], the eponymous SCA system of the company that developed it, into a PCR process implemented in an introductory CS course for matching reviewers based on the heterogeneity of their initial submissions and analyzing the subsequent effects on resubmissions. We do not claim that this approach is necessarily superior or preferable, but rather that it is emblematic of the kinds of PCR optimizations and analyses that can be generated using SCA tools such as Sense. We believe this demonstrates the potential power of leveraging SCA for PCR.

This research was supported by Len Blavatnik and the Blavatnik Family foundation.

II. Related Work

A. PCR in CS Education

Interest in using PCR as a teaching tool in CS education has grown only relatively recently, mostly as a result of the successful practice of PCR in the software industry [1]. One of the first works using to do so was done by Wang [8], who proposed a PCR model for programming courses, where students take on the roles of author, reviewer and reviser in different steps of the process. Surprisingly, almost all PCR implementations recently reviewed by Indriasari et. al [4] have inexplicably left out the revision phase, in which students submit an updated version of their program as suggested by the reviewers' comments, inevitably preventing analysis of the changes in students' submissions throughout the process. In rare cases where revisions are submitted, such as in the work of Politz et al. [9], their analysis offers only input-output comparisons. Moreover, although the importance of group composition in PCRs has been highlighted early on by Wang himself [10], nearly all implementations reviewed simply matched reviewers at random. Even when less naive techniques are employed, such as by Sun et al. [11], the focus is solely on students' personal characteristics. Overall, not a single work out of the 51 reviewed has used SCA in any of its PCR implementations.

B. SCA in CS Education

Attempts to automate the process of analyzing programming assignments in CS courses have already been made over sixty years ago, when Hollingsworth [12] used punched cards to automatically run students' assembly code and produce either "wrong answer" or "program complete". Naturally, automatic code analyses have since then evolved, with SCA most notably leading to "a new era of automated assessment" by applying state-of-the-art algorithms from domains such as graph theory and machine learning to abstract representations of code [13]. One prime example can be seen in the work of Choudhury et al. [14], who used unsupervised classification and abstract syntax tree comparisons to analyze students' code submissions

and provide them with real-time instructor authored guidance. Similarly, most of the research on using SCA in CS education has focused solely on using it to generate formative feedback to students in order to facilitate their learning [15]. Only relatively recently Joyner et al. [6] have proposed that the knowledge gained from SCA can also be used for additional significant improvements, such as informing revisions to their online CS1 course's core materials and delivery methods. Our work can therefore be seen as an extension of this viewpoint, examining the feasibility of using the knowledge gained from SCA to improve common learning activities in CS education, such as PCR.

III. Methodology

Our study included 247 students of an introductory CS course, taught in Python to students taking CS as their single or double major in the first year of studies the first year of studies [16] [1]. The PCR process can be described in terms of four phases spanning five weeks:

A. Submission (Weeks 1-2)

During the first week of the semester, students were required, as part of the first homework assignment of the course, to implement `max_even_seq(n)` (see Figure 1) and upload their implementation to the Workshop module in the course's Moodle website [17] within two weeks. To ensure implementations truly reflect students' programming skills, we instructed students to work on their implementation alone and notified beforehand that any submission, even if not completely correct, will receive a full score.

In this exercise we will implement a function that, given an integer $n \geq 0$, calculates the maximum length of a sequence of even digits in n. For example, for $n =$ 23300247524689 the maximum length of a sequence of even digits is 4 (there are two sequences that fit this length: the sequence 0024 which starts with index 3 and the sequence 2468 which starts with index 9).

Remarks:

- If n does not contain even digits, the maximum length is 0.
- You may assume that the input is correct and there is no need to check for it.

Example:

```
>>> max_even_seq(23300247524689)
4
```

Figure 1: Instructions for `max_even_seq(n)`

B. Grouping & Practice (Week 3)

After the deadline for submission had passed, Sense was used to analyze all implementations submitted. We identified two "conversation points" that we believed could benefit the review process among peers (demonstrated in Figure 2):

1) *High-level strategy*: The most common implementation iterated `n`'s digits using a single loop while keeping track of the maximum even sequence length using two counters. Submissions using this implementation splitted into two: casting `n` into a string and then casting each character back into an integer, termed S1, and extracting each digit using floor division and modulus operators, termed S2. Other submissions used miscellaneous uncommon strategies, and were therefore grouped together and termed S*.
2) *Edge-case handling*: For S1 and S2 implementations, choosing when to update the maximum sequence length becomes crucial. If the update occurs only when processing odd digits, termed H*, it is absent if either the first or last digit of `n` is even – returning a wrong result when a strictly longest sequence appears at the "edge" of `n`'s digits (e.g., returning 0 for either `n=12` or `n=21`). These cases were correctly handled by either updating only when processing even digits, termed H1; updating again after exiting the loop, termed H2; or updating when processing both odd and even digits, termed H3.

```python
def max_even_seq(n):
    cur_cnt = 0
    max_cnt = 0
    for digit in str(n):
        if int(digit)%2 == 0:
            cur_cnt += 1
            if cur_cnt > max_cnt:
                max_cnt = cur_cnt
        else:
            cur_cnt = 0
    return max_cnt
```

```python
def max_even_seq(n):
    cur_cnt = 0
    max_cnt = 0
    for digit in str(n):
        if int(digit)%2 == 0:
            cur_cnt += 1
            if cur_cnt > max_cnt:
                max_cnt = cur_cnt
        else:
            cur_cnt = 0
    if cur_cnt > max_cnt:
        max_cnt = cur_cnt
    return max_cnt
```

```python
def max_even_seq(n):
    cur_cnt = 0
    max_cnt = 0
    if n == 0:
        return 1
    while n > 0:
        if n%2 == 0:
            cur_cnt += 1
        else:
            cur_cnt = 0
        if cur_cnt > max_cnt:
            max_cnt = cur_cnt
    return max_cnt
```

```python
def max_even_seq(n):
    cur_cnt = 0
    max_cnt = 0
    if n == 0:
        return 1
    while n > 0:
        if n%2 == 0:
            cur_cnt += 1
        else:
            if cur_cnt > max_cnt:
                max_cnt = cur_cnt
            cur_cnt = 0
        n //= 10
    return max_cnt
```

```python
def max_even_seq(n):
    odds = [
        '1', '3', '5', '7', '9'
    ]
    n = str(n)
    for odd in odds:
        n = n.replace(odd, ' ')
    return len(
        max(n.split(),key=len)
    )
```

```python
def max_even_seq(n):
    n = str(n)
    max_cnt = 0
    for i in range(len(n)):
        cur_cnt = 0
        j = i
        while int(n[j])%2 == 0:
            cur_cnt += 1
            j += 1
            if j == len(n):
                break
        if cur_cnt > max_cnt
            max_cnt = cur_cnt
    return max_cnt
```

Figure 2: Examples of S1 high-level strategies (top) using H1 (left) and H2 (right) edge-case handlings, S2 high-level strategies (middle) using H3 (left) and H* (right) edge-case handlings and S* high-level strategies (bottom)

Accordingly, all submissions were labeled by their high-level strategy (S1, S2, or S*), with S1 and S2 submissions also labeled by their edge-case handling (H1, H2, H3, or H*), and then randomly split into experiment (EXPT) and control (CTRL), such that each had roughly the same number of submissions with each label. As the ratio of high-level strategies was roughly 2:1:1 while the ratio of edge-case handlings was roughly 2:2:1:1, students in both EXPT and CTRL were ultimately grouped into teams of four such that students in CTRL were grouped at random and students in EXPT were grouped by maximizing the heterogeneity of labels in each team using the simulated annealing algorithm [18].

[1] The course's permanent website is http://tau-cs1001-py.wikidot.com.

While waiting for their teams to be formed, students participated in a one-hour guided code review session called "Tips for 'Good' Code". As its name suggests, the aim of this session was not to convey the idea that there is a universal definition of "good code", but rather help students develop their own coding review capabilities. For example (shown in Figure 3), students were asked to choose between two implementations for reversing a given list, one using a loop and the other using Python's list comprehension. After highlighting the elegance of the latter, students were asked to choose between two implementations for finding the second largest element of a given list, one using a loop and the other using Python's `sort` function. After noting that this time the elegance of the latter came at the cost of a potentially slower running time, students were ultimately advised to use Python's built-in shortcuts only after considering all consequences and possible downsides.

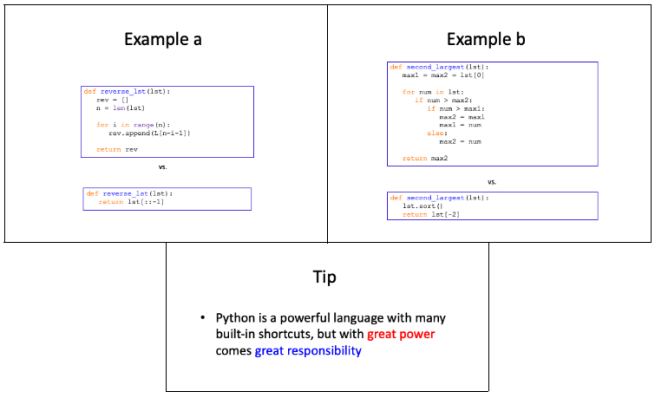

Figure 3: Slides used in the "Tips for 'Good' Code" session

C. Review (Week 4)

Once the class session had been completed, students were required, as part of the second homework assignment of the course, to review their teammates' implementations within a week based on three rubrics (demonstrated in Figure 4):

1) *Correctness*: testing the reviewed implementations against four different inputs of their choosing, accumulating the number of passed tests to a score of 0-4 and listing the inputs that were tested.
2) *Complexity*: measuring the actual running time of the reviewed implementations as well as their own implementation on four large inputs of their choosing using Python's `timeit` module [19], accumulating the number of times each reviewed implementation was faster than their own to a score of 0-4 and list all timings measured.
3) *Clarity*: evaluating the readability of the reviewed implementation using a score of 0-4 based on the tips discussed in the "Tips for 'Good' Code" class session and explaining the scoring as descriptively as possible.

Students were notified beforehand that in order to allow them to be honest in their analysis and feedback, the review process will be double-blinded, ensuring they will not be aware of who they are interacting with, and that similar to the submission phase, any review following the instructions will receive a full grade, regardless of its quality.

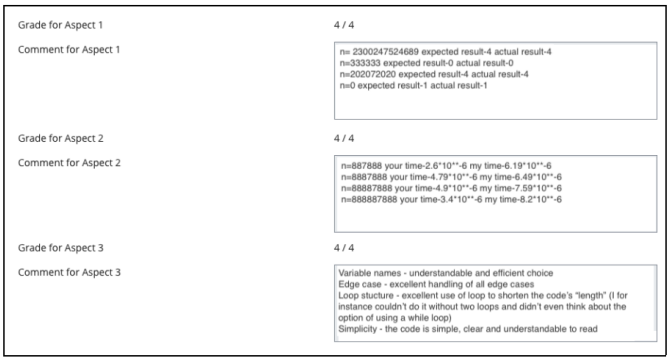

Figure 4: Review example

D. Resubmission (Week 5)

Finally, after every review had been submitted, students were given access to the reviews written by their teammates and required, as part of the third homework assignment of the course, to resubmit within a week the best implementation of `max_even_seq(n)` they possibly could based on the reviews made by their peers. At this point, students were also required to consent that their data will be used for our research.

IV. Preliminary Results

Out of 247 participating students, 51 either did not resubmit an implementation or did not consent for their data to be used in our research and were therefore left out of our analyses, leaving 100 students in CTRL and 96 students in EXPT while roughly maintaining 2:1:1 and 2:2:1:1 ratios of high-level strategies and edge-case handlings respectively in both (see Tables 1 and 2).

To understand how our peer matching affected students' resubmissions, Sense was used again to label all resubmissions by their high-level strategy and edge-case handling. By comparing labels of resubmissions to the labels of their corresponding submissions (see Tables 3-8), three interesting trends were uncovered. First, while many students in CTRL continued to use S* high-level strategy in their resubmission, significantly less did so in EXPT ($p<0.05$ using the Chi-squared test). Second, students in EXPT opting out of S* high-level strategies, as well as those opting out of H* edge-case handlings, seemed more likely to use H2 edge-case handlings in their resubmissions – although these results were not found to be significant ($p<0.05$ using Fisher's exact test, due to cells with expected counts less than 5). Third, the overall trend in other high-level strategies and edge-case handlings was not to not change either of them.

	S1	S2	S*
CTRL	50	26	24
EXPT	49	23	24

Table 1: High-level strategies of submissions

	H1	H2	H3	H*
CTRL	23	27	15	11
EXPT	24	26	13	9

Table 2: Edge-case handlings of submissions

		Resubmission		
		S1	S2	S*
Submission	S1	42	8	0
	S2	7	17	2
	S*	10	3	11

Table 3: High-level strategies in CTRL

		Resubmission		
		S1	S2	S*
Submission	S1	41	8	0
	S2	6	16	1
	S*	14	7	3

Table 4: High-level strategies in EXP

		Resubmission			
		H2	H2	H3	H*
Submission	H1	19	1	2	1
	H2	2	21	2	2
	H3	0	1	11	1
	H*	1	7	1	2

Table 5: Edge-case handlings in CTRL

		Resubmission			
		H2	H2	H3	H*
Submission	H1	20	1	2	1
	H2	1	22	1	1
	H3	2	0	11	0
	H*	2	3	2	2

Table 6: Edge-case handlings in EXPT

	H1	H2	H3	H*
CTRL	0	0	2	0
EXPT	0	1	0	0

Table 7: Edge-case handlings of submissions for S* resubmissions

	H1	H2	H3	H*
CTRL	3	5	2	3
EXPT	5	10	3	3

Table 8: Edge-case handlings of resubmissions for S* submissions

V. Conclusion

Although SCA and PCR have been studied extensively in education contexts, they have surprisingly coexisted so far rather independently. In this work, we have made the arguably first effort for combining the two rich bodies of knowledge together, by using Sense in a PCR process implemented in an introductory CS course for matching peers based on the heterogeneity of their initial submissions' and analyzing the subsequent effects on resubmissions. Our initial findings suggest that while our matching technique was not as impactful as we had expected, it did affect the resubmissions of students who submitted either non-standard or incorrect initial implementations.

With that in mind, much is left for future work. For a start, one can examine in more detail whether changing one or more of the settings chosen for our experiment will lead to different results. Alternatively, SCA can also be used to explore correlations between students' resubmissions and other factors, either within the PCR process (e.g., students' satisfaction with the process) or outside of it (e.g., course grades).

All in all, we believe our work offers two significant contributions to the CS education research:

1) Our peer grouping technique is an exemplary use of SCA for assigning reviewers in PCRs, as well as a novel addition to the relatively scarce literature on the effects of group composition in PCRs
2) Our analysis methodology is an exemplary use of SCA for assessing the outcomes of PCRs, as well as a novel addition to the relatively scarce literature on code revisions in PCRs.

We hope that together, these contributions demonstrate the potential power of leveraging SCA for PCR in CS education.

References

[1] Gehringer, E. F., Chinn, D. D., Pérez-Quiñones, M. A., & Ardis, M. A. (2005, February). Using peer review in teaching computing. In Proceedings of the 36th SIGCSE technical symposium on Computer science education (pp. 321-322).

[2] Wang, Y., Wang, H., Schunn, C., & Baehr, E. (2016). Choosing a Better Moment to Assign Reviewers in Peer Assessment: The Earlier the Better, or the Later the Better?. In EDM (Workshops).

[3] Khandelwal, S., Sripada, S. K., & Reddy, Y. R. (2017, February). Impact of gamification on code review process: An experimental study. In Proceedings of the 10th innovations in software engineering conference (pp. 122-126).

[4] Indriasari, T. D., Luxton-Reilly, A., & Denny, P. (2020). A review of peer code review in higher education. ACM Transactions on Computing Education (TOCE), 20(3), 1-25.

[5] Combéfis, S. (2022). Automated code assessment for education: review, classification and perspectives on techniques and tools. Software, 1(1), 3-30.

[6] Joyner, D., Arrison, R., Ruksana, M., Salguero, E., Wang, Z., Wellington, B., & Yin, K. (2019, February). From clusters to content: Using code clustering for course improvement. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education (pp. 780-786).

[7] https://www.sense.education/.

[8] Wang, Y., Su, X., Hu, Y., & Wang, Q. (2007). How to Evaluate Students' Learning Outcome: A Peer Code Review Model in Undergraduate Programming Class. In Proceedings of International Conference of Computer Science and Engineering (ICCSE'2007) (pp. 1292-1295).

[9] Politz, J. G., Krishnamurthi, S., & Fisler, K. (2014, July). In-flow peer-review of tests in test-first programming. In Proceedings of the tenth annual conference on International computing education research (pp. 11-18).

[10] Wang, Y., Yijun, L. I., Collins, M., & Liu, P. (2008). Process improvement of peer code review and behavior analysis of its participants. ACM SIGCSE Bulletin, 40(1), 107-111.

[11] Sun, Q., Wu, J., Rong, W., & Liu, W. (2019). Formative assessment of programming language learning based on peer code review: Implementation and experience report. Tsinghua Science and Technology, 24(4), 423-434.

[12] Hollingsworth, J. (1960). Automatic graders for programming classes. Communications of the ACM, 3(10), 528-529.

[13] Paiva, J. C., Leal, J. P., & Figueira, Á. (2022). Automated assessment in computer science education: A state-of-the-art review. ACM Transactions on Computing Education (TOCE), 22(3), 1-40.

[14] Choudhury, R. R., Yin, H., Moghadam, J., Chen, A., & Fox, A. (2016, May). Autostyle: Scale-driven hint generation for coding style. In Proceedings of the 13th International Conference on Intelligent Tutoring Systems, ITS (Vol. 201, pp. 122-132).

[15] Keuning, H., Jeuring, J., & Heeren, B. (2018). A systematic literature review of automated feedback generation for programming exercises. ACM Transactions on Computing Education (TOCE), 19(1), 1-43.

[16] Chor, B., & Hod, R. (2012, July). CS1001. py: a topic-based introduction to computer science. In Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education (pp. 215-220).

[17] https://docs.moodle.org/19/en/Workshop_module.

[18] Bertsimas, D., & Tsitsiklis, J. (1993). Simulated annealing. Statistical science, 8(1), 10-15.

[19] https://docs.python.org/3/library/timeit.html.