Using Model-Checking and Peer-Grading to Provide Automated Feedback to Concurrency Exercises in Progvis

Filip Strömbäck
Department of Computer and Information Science
Linköping University
Linköping, Sweden
filip.stromback@liu.se

Linda Mannila
Department of Computer and Information Science
Linköping University
Linköping, Sweden
linda.mannila@liu.se

Mariam Kamkar
Department of Computer and Information Science
Linköping University
Linköping, Sweden
mariam.kamkar@liu.se

ABSTRACT

Previous research has shown that even though many students are aware of overarching problems with concurrency, they are less successful in addressing any issues they have found. This implies that the students have not yet developed a mental model that describes the behavior of concurrent systems with enough accuracy. One way to help students explore the non-determinism of concurrent systems and thereby develop their mental model is through the use of visualization tools. One example of such a tool is Progvis, which provides students with a detailed visualization of the program state, and allows students to single-step individual threads to explore the program's behavior in a concurrent environment. One problem with this type of tools is that they are not able to provide feedback on whether or not a proposed solution is correct, which limits their percieved usefulness.

To increase the percieved usefulness of Progvis, we extended it with a system that utilizes model-checking and peer-grading to provide automated feedback to students. Our hopes were that this would encourage students to further use Progvis to practice concurrent programming. The system was used during two years in a course on concurrency and operating systems. This made it possible to utilize the experiences from the first year to further improve the system for the second year. Overall, the students expressed that they found our additions helpful. Additionally, we observed a slight increase in usage in the second year compared to the first year, which suggests that the improvements in the second year increased students' motivation to some extent.

CCS CONCEPTS

• **Applied computing** → **Education**; • **Human-centered computing** → *Visualization systems and tools*; • **Theory of computation** → *Concurrency*.

KEYWORDS

concurrency, synchronization, visualization, model checking, peer grading, gamification, undergraduate, computer science education

ACE '23, January 30-February 3, 2023, Melbourne, VIC, Australia
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9941-8/23/01.
https://doi.org/10.1145/3576123.3576125

ACM Reference Format:
Filip Strömbäck, Linda Mannila, and Mariam Kamkar. 2023. Using Model-Checking and Peer-Grading to Provide Automated Feedback to Concurrency Exercises in Progvis. In *Australasian Computing Education Conference (ACE '23), January 30-February 3, 2023, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3576123.3576125

1 INTRODUCTION

Many students struggle with learning concurrency. Previous research has shown that while most students are aware of concurrency issues in a program, many fail to address them adequately [12, 14]. Similarly, others have found that many students fail to synchronize small concurrent programs during the final exam in a course on concurrency [19]. This likely means that students have not yet developed a mental model that adequately describes the behavior of concurrent programs. As in many other areas, this requires time and practice. However, the non-deterministic nature of concurrency complicates this process since it makes it difficult to empirically test whether one's mental model is correct or not. This makes students more reliant on feedback from teachers, which in turn makes self-study more difficult. The non-determinism also means that the abstractions used to describe concurrent concepts are quite different compared to those used in sequential programming [20], which further complicates self-study.

One way to help students develop their mental model in spite of the additional complexity is to show students a notional machine by using a visualization tool such as Progvis [21]. While these tools are useful to illustrate the behavior of concurrent programs, they do not solve all problems that students face. Even though visualization tools make it easier for students to find concurrency issues and understand why they occur, they are often not able to provide feedback regarding whether a particular solution is free from concurrency issues or not. Some visualization tools (e.g., Progvis [22] and ConEE [17]) contain model-checkers that are able to help students by finding interleavings that cause the program to misbehave. These act as examples that illustrate why the solution is incorrect. However, even if model-checkers are a step in the right direction, they still rely on the existence of a test program. This means that model-checkers are not able to verify the correctness of abstractions in general, just like it is not possible to test the correctness of sequential abstractions without a suitable test suite. This means that students still have to rely on teachers for feedback to some extent, even if visualization tools and model-checkers improve the situation. As previously mentioned, this makes self-study more difficult and thus less efficient for students.

To help students receive feedback in areas where visualizations and model-checkers are lacking, we extended Progvis with a system that combines peer-grading with the automatic model-checking to provide better (semi-)automatic feedback to students with the aim of reducing students' need to rely on teachers' feedback. Our hopes were that the improved automatic feedback would also encourage students to use the system to a larger extent, and thereby spending time developing their mental models. For further encouragement, students were also awarded points for interacting with it, and the top students were displayed on a leaderboard. This paper describes the iterative design of the system during a period of two years. During this two-year period, the system was evaluated in an undergraduate course on concurrency and operating systems. This allowed us to examine how students used the system and thereby further inform the future development of the system.

The remainder of this paper is structured as follows: Section 2 presents related work and introduces Progvis in more detail. Section 3 then presents an overview of the course in which we used the system. Sections 4 and 5 then describes our system in each of the two years, how students used it, and a short discussion of the results. Finally, we present a conclusion in Section 6.

2 RELATED WORK

This section provides an overview of related work in the area of teaching and learning concurrency in general, followed by an overview of tools that aim to help students. Finally, we provide a more detailed introduction to Progvis, which we use in this paper.

2.1 Teaching and Learning Concurrency

Much research exists on the topic of teaching and learning concurrency. One example is Kolikant [7, 9], who studied high-school students and found that students had difficulties with identifying the required synchronization goals. Once they were identified, students generally managed to arrive at a solution. Lawson and Kraemer [11] studied a similar problem in the same context and additionally found that students sometimes use sleep functions rather than proper synchronization mechanisms to address concurrency issues. As mentioned in Section 1, yet others have studied students' performance in an undergraduate context. Both Lawson et al. [12] and Lewandowski et al. [14] found that students were generally aware of concurrency issues, but were not always able to address the issues in a suitable manner. Similarly, Strömbäck et al. [19] found that most students were able to find concurrency issues in a piece of code on their final issues, but less than half the students were able to produce a correct solution.

There are many reasons why students might fail to correctly solve this type of problems. Kolikant [10] suggests that the issue might be that students have an alternative view of correctness, and that they are thereby satisfied with a solution that only works most of the time rather than all the time. Another reason could be that the non-deterministic nature of concurrent programs makes testing difficult. Therefore, the trial-and-error approach to develop programs (which Lönnberg et al. [16] found was used in concurrency) is no longer viable. Students must therefore transition into relying more on formal reasoning than before [8]. Additionally, the non-determinism means that the abstractions used in concurrent programming leaves more situations undefined compared to abstractions in sequential programming [20]. This difference from sequential programming likely further complicates learning concurrency. This also highlights the importance of teaching suitable models, as highlighted by Ben-Ari and Kolikant [2].

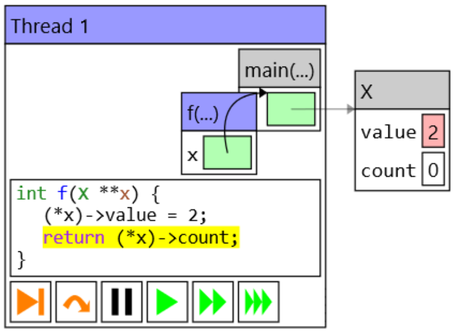
2.2 Tools for Teaching Concurrency

To help students learn concurrency, Ben-Ari and Kolikant [2] argue for the importance of using suitable models to illustrate the semantics of programs. One way to do this without relying on the teacher is by using a visualization tool. A number of such tools have been developed with the aim of aiding students' learning of concurrency. Eludicate [4] is one such example. It allows capturing and examining dynamic execution traces of Java programs, and thus allows students to examine how the execution of different threads interleave. Another similar, but more advanced approach is used by Atropos [15]. It also relies on dynamic execution traces of Java programs, but also visualizes data dependencies between threads, which makes it easier to infer the synchronization goals of the program. As both these tools are based on execution traces, the ability for students to explore different interleavings at will is, however, limited.

Another approach that gives the user more control is taken by tools such as The Deadlock Empire[1], ConEE [17] and Progvis [21]. These tools all draw inspiration from visualizations of sequential programs and allow students to single-step the execution of each running thread. Students are thereby able to explore different interleavings at will, and they may thereby experiment and validate their understanding of many concepts. This does, however, come at a cost. Namely that students need to actively explore many interleavings in order to find out whether their program is correct or not. This is utilized in The Deadlock Empire to engage students through gamification: the tool contains a number of pre-defined programs, each of them containing some form of defect. The student is then asked to find an interleaving that exposes the defect, and causes the program to misbehave. ConEE [17] is similar, but allows visualizing arbitrary programs written in a custom language. It also has the ability to exhaustively search through all possible interleavings, which allows automatic feedback in the form of whether or not the program was correct. Progvis [21] is similar to ConEE, but rather than using a custom programming language, it allows visualizing arbitrary programs in a subset of the C language. To be able to provide an effective visualization of programs written in C, Progvis has the ability to visualize more complex data structures compared to The Deadlock Empire and ConEE. A large benefit of including a rich visualization of data is that it allows students to explore how concurrency interacts with other, more fundamental, programming concepts. Recent improvements to Progvis [22] have further improved the accuracy of Progvis' visualizations, and added a model-checker similar to that in ConEE to allow automatic detection of concurrency issues in the visualized programs.

There are additionally a number of tools that aim to find concurrency errors in programs rather than focusing on visualizing them. One example is Eraser, which Choi and Lewis [3] used to identify data races in students' solutions to concurrency problems

[1]https://deadlockempire.github.io/



```
int f(X **x) {
   (*x)->value = 2;
   return (*x)->count;
}
```

Figure 1: Progvis visualizing a simple program involving pointers and a small data structure. Variables with a red background have just been written to, and variables with a green background have just been read from.

in C. Another example is the Spin model checker, which Ben-Ari found to be simple enough to be usable in teaching even though it was designed for professionals [1]. As with ConEE, the Spin model checker uses a custom language, Promela.

2.3 Progvis

As mentioned previously, Progvis is a visualization tool for concurrent programs [21]. Its focus is not only on concurrency in isolation, but also on how concurrency interacts with other more fundamental concepts, such as pointers and scope. Progvis achieves this by providing a detailed visualization of the program's state and allows students to single-step threads individually to observe how different interleavings affect the correctness of the program. The representation of the fundamental concepts is inspired by visualization tools aimed at introductory courses, such as UUhistle [18], Python Tutor [6] and Jeliot [13]. This representation is shown in Fig. 1, where Progvis visualizes a small program that modifies a data structure through a pointer to a pointer.

Since some of these fundamental concepts are known to be difficult, their inclusion will most likely cause some initial confusion for students. For example, Fisler et al. [5] found that students struggle with scope, mutation and aliasing in their second and third years of undergraduate studies. As we have found in our previous work [19] it is, however, vital for students to understand how these concepts interact with concurrency. For example, it is not possible to determine which variables need to be protected by locks without identifying which variables are shared, which is in turn not possible without understanding how the scoping rules and parameter passing work in both sequential and concurrent programs.

Since its creation, Progvis has also gained the ability to detect and report data races, and to automatically find interleavings that cause the program to fail using a model-checker [22]. The model-checker uses a transition system to verify that none of the many possible interleavings of the visualized program cause the program to fail (e.g., causing a data race or some other crash). It is thus able to verify whether or not a program as a whole is free from concurrency issues. Because of this, the model-checker is not able to verify whether some abstraction (e.g., a set of functions that implement a data structure) follows its specification. To achieve this, the model checker also needs a test program that uses the abstraction in a suitable manner. With the help of this test program, the model-checker is then able to verify that the abstraction behaves according to the expectations of the test program for all possible interleavings. The quality of the feedback from the model-checker for this purpose thus depends on the quality of the test program. The model-checker is described in detail in [22].

3 CONTEXT: THE COURSE

The improvements to Progvis described in this paper were evaluated for, and tested in an undergraduate course on concurrency and operating systems given at Linköping University. The course is given at the end of the second year of a three year bachelor program in computer science. It consists of three parts: a set of lectures, a set of computer lab assignments, and a final exam. All students have previously attended a course that covers the theory behind operating systems. As such, the lectures in the course focuses on concurrent programming in the shared memory model after a brief introduction of the C language (students are already familiar with C++ from prior courses). The main focus of the course is the computer labs where students implement a number of system calls in the educational operating system Pintos.[2] The students first implement the system calls `open`, `close`, `read` and `write` for managing file and console I/O. These can be implemented by utilizing existing code in Pintos, and they do not require the students to consider concurrency. In the next part of the computer labs, students implement the system calls `wait` and `exec` (`exec` in Pintos is equivalent to `fork` followed by `exec` in POSIX). This requires students to synchronize the execution of multiple processes using semaphores. After this, students synchronize the existing file system implementation using locks to ensure that multiple processes can safely access the same files and directories. This part also involves implementing a reader-writer lock to allow multiple processes to read the contents of the same file concurrently, but disallowing writes to occur concurrently with other operations. Finally, students make sure that the system calls are secure by validating that any pointers passed to the system calls refer to valid memory in a region accessible by the process. This part does not involve concurrent programming.

During both years Progvis was available for students to use and the assignment that involved creating a reader-writer lock presented Progvis as a way to debug the implementation in isolation. Due to limitations in Progvis it was, however, not possible to visualize the entirety of Pintos, and therefore few students used it voluntarily. In the second year, two new lab assignments were added to introduce semaphores and locks in Progvis. These assignments asked students to synchronize a small piece of code that resembled what they would later encounter in Pintos.

Many students experience the lab assignments as very time-consuming, partially due to the large size of the Pintos codebase and the scope of the assignments compared to prior courses, but also due to the difficulty of debugging concurrency- and memory issues. A part of these struggles might be due to the need to transition to a more formal approach to programming as mentioned by Kolikant [8]. The course ends with a final exam which contains questions

[2]http://www.scs.stanford.edu/07au-cs140/pintos/pintos.html

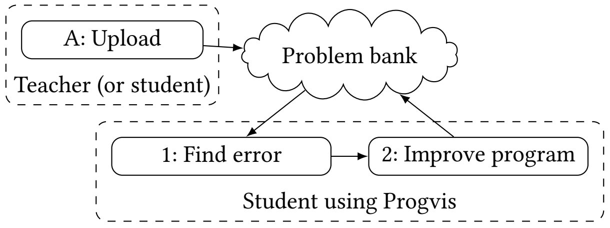


Figure 2: Overview of the peer-grading system added to Progvis during the first year.

where students are given a piece of code, and asked to identify and eliminate any concurrency issues in the code.

The final part of the course is a final exam. The focus of the exam is two or more assignments where students are asked to synchronize a small piece of C code. This style of questions is similar to the questions that were used in the pilot study of Progvis [21], and those used in the system introduced in this paper. The years studied in this paper are 2021 and 2022, and due to the ongoing pandemic the course was taught remotely in 2021.

4 THE FIRST YEAR

The system used in the first year was developed and evaluated before the improved memory model and the model-checker were available in Progvis. As such, Progvis was not able to provide automatic feedback on whether students' solutions contained concurrency errors or not at the time. Therefore, the goal of the first version of the system was to utilize peer-grading to provide this type of feedback without relying on a teacher. To encourage students to use the system, and thereby increasing the likelihood of students receiving feedback, we added gamification elements to the system. In particular, students were awarded points for attempting to solve problems and for finding concurrency issues in other students' solutions. Our hopes were that the ability to get feedback on problems, coupled with the points awarded would motivate students to use Progvis to explore concurrency and thereby also developing their mental model of concurrency.

The design of the first iteration of the system is illustrated in Fig. 2. As can be seen in the figure, the system was designed around a central problem bank. Each problem consisted of a small concurrent program written in the C language. The teacher first uploaded a set of such problems, which we refer to as *initial problems*, to the problem bank (step A in Fig. 2). Each of these initial problems contained some concurrency errors that students would later be asked to identify and fix (i.e., the same type of problems used in [21]). Students were also able to upload initial problems if they wished to receive feedback on their own code.

Students were then able to browse the problem bank to find problems to solve. Problems were solved in two steps. First, the system presented the student with a visualization of the program and asked them to find an interleaving that exposes a concurrency error in the program (step 1 in Fig. 2). If the student succeeded to find such a problematic interleaving, they were then asked to modify the program to fix the error they found (step 2 in Fig. 2). During the latter step, students had the ability to modify the code in the operating system's default text editor, and they could visualize their proposed solution in Progvis as they saw fit. Students who were unable to find a problematic interleaving in step 1 could indicate that they believed the code to be correct by clicking a button.

Two pieces of data were recorded by the system for each problem a student attempted to solve. If the student managed to find a problematic interleaving, the system recorded the interleaving and what type of error the interleaving would expose (e.g., assertion, use after free). If the student believed the code to be correct, this fact would also be recorded, but no interleaving would be recorded. If a student proceeded to fix the issues they found, their modified version of the problem would be submitted back to the problem bank (as indicated in Fig. 2) as an *improved problem*. The improved problems were treated almost exactly like initial problems by the problem bank: other students were able to browse them and attempt to find errors in them as for initial problems. The distinction between initial problems and improved problems exists to allow the author of a problem (both to initial- and improved problems) to easily find and view any improvements made by other students. The author of the problem was also able to view a visualization of any problematic interleavings found by other students. This information was also shown to all students who had solved the particular problem. Thus, the feedback provided by the system was twofold: first, the system would provide problematic interleavings that the original author might have failed to consider, and second, it provided suggestions for how to solve any issues present. These suggestions for further improvements could then further be peer-graded in a similar manner.

Since this approach relies on students using the system for peer-grading to work, it is important to motivate students to use the system. To achieve this, points were awarded for interacting with the system, and the ten students with the highest scores were displayed on a leaderboard in the system. To reward tasks that provide feedback to other students, the number of points awarded for different tasks were based on the task's value from a peer-grading perspective. For example, if a student finds an error in another student's problem, this provides valuable feedback to the author of the problem, and is thus rewarded with a comparatively large number of points. On the other hand, declaring a problem as being correct provides comparatively little value as this might be incorrect, and since the system was unable to verify the correctness of such statements. As such, points were awarded for solving other students' problems as follows:

- Finding an error in a problem: 5 points
- Uploading an improved problem: 2 points
- Declaring a problem as being correct: 1 point

To disallow students collecting points by submitting similar solutions to the same problems, the system only accepted one solution that highlighted a particular issue for each problem. For example, only one solution that caused an assertion would be accepted. Additionally, to encourage students to submit correct solutions the author of a problem received points based on the peer grading as follows:

- Author of a problem declared as correct: 2 points
- Author of a problem someone finds an error in: 1 point

4.1 Results From the First Year

Out of the the 85 students who signed up to the computer lab assignments in the first year, 78 solved at least one of the lab assignments, 11 students signed in to the system at least once (i.e., 13% of the students who signed up), and 5 students were active in the sense that they found at least one error (the other students might have viewed problems, this would not be visible in our data). The timestamps revealed that one of the students was only active in the weeks before the retake exam, and not during the course itself. Each of the active students found between 7 and 14 errors (11.4 on average), and uploaded between 3 and 11 improved problems to the problem bank (7.4 on average). However, only one student attempted to find an error in an improved problem. In this case they correctly concluded that the improved problem was correct.

Since we only have evidence of one student having explored other students' improved problems, we focus on the 15 problems uploaded by the teacher, which are shown in Table 1. The column *correct* shows that only three solutions indicated that an initial problem was correct. These were made by two students: one student initially answered that problems 1 and 3 were correct, but later found an error in problem 1. The other student initially answered that problem 3 contained an error, but later found an error there. The column *found error* contains the number of students that found at least one error. The next column, *no. of errors*, contains the number of errors found. In most cases the numbers in these columns are equal, which means that most students only found one type of error. Some students did, however, find multiple types of errors in problems 4, 7, 12, and 15. Finally, the column *improved* contains the number of improvements submitted to each problem. There was only one instance where a student submitted more than one improvement to a problem (to problem 7). We can also see students submitted an improvement after finding an error in 70% of all cases. The only problems students did not submit an improvement to were problems 14 and 15. For problem 14 this is likely because it uses atomic operations, which are only covered briefly in the course. Problem 15 was covered in detail during a tutorial session with TAs, which is likely why no students felt it necessary to submit a solution to it.

To assess the quality of the solutions we manually graded the 37 improved problems after the end of the course. We found 8 improvements that were incorrect in some way. These were distributed between 6 of the problems as follows:

Problem 3: One student attempted to solve the issue in this problem by using atomic operations to implement a lock. The only issue was that they did not use an atomic store to release the lock.

Problem 4: One student acquired a lock but forgot to release it. This particular issue was not visible in this case as the correct location of the release operation would have been at the end of the `main` function.

Problem 7: Two of the submitted improvements to this problem were incorrect. One failed to wait for the other thread to complete, which was caught by an assertion in the code. The other included an unnecessary lock to protect data already protected by a semaphore.

Problem 8: One of the submitted improvements altered the semantics of the `wait` function by using a global semaphore. The `main` function was modified to compensate for the altered semantics.

Table 1: Overview of the usage of the system in the first year.

Problem	Description	*Correct*	*Found error*	*No. of errors*	*Improved*
1	Shared global	1	5	5	5
2	Shared by pointer	0	5	5	4
3	Global, no threads	2	4	4	1
4	2-thread array summation	0	5	6	4
5	Bounded buffer, single r/w	0	4	4	2
6	Bounded buffer, multi r/w	0	3	3	1
7	Task 1 from [21] (wait for a thread)	0	4	6	5
8	Task 2 from [21] (spawn and wait)	0	3	3	3
9	Task 3 from [21] (bounded buffer)	0	2	2	2
10	Task 4 from [21] (future)	0	3	3	3
11	Variant of problem 4	0	2	2	1
12	Waiting for a thread	0	3	4	3
13	Transaction in a bank	0	3	3	3
14	Linked stack using atomics	0	1	1	0
15	Dining philosophers	0	1	2	0
	Total	3	48	53	37

Problem 9: One student solved the concurrency issues in this problem by protecting the usage of the data structure rather than the data structure itself, contrary to a comment in the code.

Problem 10: One of the two incorrect improvements failed to account for the fact that two threads could try to get the value from a future object concurrently. This could cause their implementation to deadlock. The situation required for the deadlock to occur could, however, not happen in the way the future was used in the provided test program. The other incorrect improvement failed to wait in some cases.

4.2 Discussion of Results From the First Year

Since a majority (29 of 37) of the submitted improvement were correct, this suggests that the visualizations in Progvis aided students when developing their improvements, similarly to the conclusions of previous work [21]. However, the incorrect improvements for problems 8, 9, and the first incorrect improvement for problem 10 highlight a shortcoming of the system. These improvements were technically correct in the sense that they did not contain any concurrency issues when used with the provided example. They did, however, not follow the specification of the data structure as specified by the problem (provided as comments in the code, similarly to [21]). As such, this type of problem would not be found by a model-checker unless a more sophisticated test program was provided. This type of issue with concurrent aspects of abstractions has previously been highlighted as a possible cause for mistakes when solving concurrency problems [20]. As such, this is one of the problems we aimed to solve in the second iteration of the system. Furthermore, as the ability to interpret a specification and construct suitable test cases is also a valuable skill, we aimed to let students practice writing this type of tests as well in the second iteration of the system.

Another observation is that only one student seems to have attempted to find errors in other students' improved problems. This is unfortunate, as it means that students will not get feedback on their improved problems, which in turn reduces the usefulness of the system. One reason for this might be because most improved problems were indeed correct, and that students thus quickly realized that it was usually fruitless to try to find errors in improved problems. Another detail that probably reinforced this belief was that improved problems were labelled "Solution to X" (where X was the name of the initial problem).

A final issue was that only 5 of the 85 registered students used the system actively. There are many possible reasons for this. Since many students experience the lab assignments in the course as very time-consuming, this could be due to a perceived lack of benefits given the time investment required to use the system. Even though Progvis was used during the lectures, it could also be due to a lack of integration into the course, causing students to fail to realize its relevance. Sadly, the lack of students who actively use the system also makes the system less attractive to use, as it reduces the possibility of receiving feedback from peers.

5 THE SECOND YEAR

In preparation for the second year, we used the observations from the first year (Section 4.2) to revise the system and create a second iteration of the system. Since most improved solutions were correct, we opted to shift the focus of the peer-grading from assessing whether or not the code contained concurrency errors to whether or not the implementation of some abstraction (e.g., the implementation of a data structure) followed the specification of its interface. This change was further motivated by the fact that the authors of Progvis implemented the model-checker in Progvis [22] before development of the second version began. As the model-checker allowed Progvis to automatically assess whether a particular program contained concurrency errors or not, the previous approach of using peer-grading to achieve the same goal became redundant. Shifting focus to assessing whether an implementation follows a specification does, however, not have the same problem as the model-checker is not able to assess this aspect automatically without a suitable test suite.

To shift the focus to how well an implementation follows its specification, problems were split into three parts: an *implementation*, a *test*, and a *reference implementation*. As was the case in the first year, each of the three parts consisted of a piece of C code. The *implementation* contained an implementation of some abstraction, typically a data structure. The expected behavior of the abstraction was documented in comments alongside each of the abstractions' functions. This implementation was initially incorrect (e.g., by lacking synchronization altogether). The *test* then contained code that used the abstraction through its interface, and thereby tested some of its behavior. Combining the test with the implementation makes it possible to utilize the model-checker to check whether the behaviors tested by the test exposes some concurrency issues in the implementation or not. The test was also incomplete initially, and it typically only illustrated basic usage of the data structure without making any effort to exercise more interesting behaviors of the implementation. This basic usage was, however, typically enough

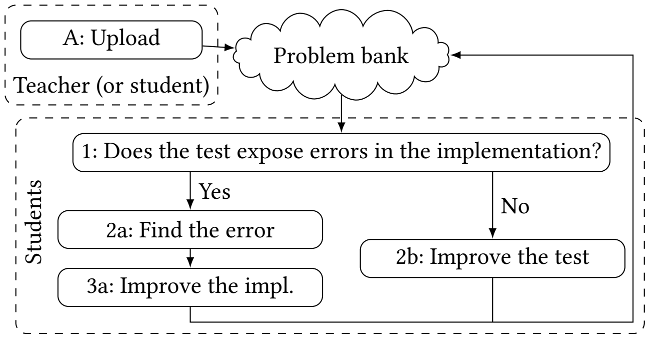


Figure 3: Overview of the system added to Progvis during the second year.

to expose some issue in the incomplete implementation. The final part, the *reference implementation*, contained a known good implementation of the abstraction. This implementation would never be shown to students, but was only used to validate whether or not the test was correct using the model-checker.

The illustration in Fig. 3 shows how these three-part problems were used in the system. Similarly to the first year, the teacher first uploaded a set of problems to the problem bank (step A in Fig. 3). Students were then able to browse the problems in the problem bank and select one to solve. In the second year, problems were solved differently depending on whether the test exposed errors in the implementation or not (step 1 in Fig. 3). Problems where the model-checker found that the test exposed an error in the implementation were solved similarly to the first year: the student would first be shown a visualization of the program, and asked to find an interleaving that exposes an error (step 2a in Fig. 3). One difference from the first year was that studens did not have the option to indicate that they believed the implementation to be correct, since the model-checker had verified that this was not the case. After finding a problematic interleaving, the student would then be asked to improve the implementation to fix the error by modifying it in the operating system's default text editor (step 3a in Fig. 3). When the student believed they had fixed the issue they found, they could submit their improved implementation back to the problem bank. As a part of this process, the system used the model-checker to verify that the improved implementation fixed all errors exposed by the test. The results of this verification were presented to the student before submitting their solution, but they had the option to submit regardless of whether errors were found or not.

Problems where the test did not expose any errors in the implementation were solved differently. In this case, the student would simply be asked to improve the test to make it expose an error in the implementation (step 2b in Fig. 3). During this step, the student was able to use the visualizations in Progvis to explore the impact of their improved test. Before submitting the improved test to the problem bank, the system used the model-checker to verify 1) that the improved test actually exposed an error in the implementation, and 2) that the improved test did not expose an error in the reference implementation. If the improved test exposed an error in the reference implementation, the test would be considered invalid and it would therefore not be possible to submit. In this case, the student would not be allowed to submit their improved test unless it successfully exposed an error in the implementation.

As in the first year, the improved problems were uploaded back into the problem bank along with any problematic interleavings found in step 2a. The system kept track of the relation between the improved problem and the original problem that it improved upon so that the author of the original problem would be able to easily find and view any improvements. In this way, the author of the problem would be able to receive feedback on their problem, either in the form of an improved test that highlighted some problem in the improved implementation, or in the form of an improved implementation that fixed some error exposed by the test.

To motivate students to use the system and thereby provide feedback to their peers, students were again awarded points for interacting with the system. The top ten students were displayed on a leaderboard in the system. Again, the number of points awarded for each task was determined based on the value of the task from a peer-grading perspective, particularly in areas where the auto-grader would not be able to provide automated feedback. As such, a student who provides an improved implementation that correctly solves the issues exposed by the test is awarded with a comparatively large number of points since it provides an example of how problems can be solved. Similarly, improving a test so that it exposes new issues is rewarded with even more points, since this type of feedback is valuable for the author of the problem to highlight parts of the semantics that they might have not understood properly. As such, points were awarded as follows:

- Finding an error (step 2a): 10 points
- Submitting an improved implementation (step 3a): 10 points
- Submitting an improved implementation that solves all issues exposed by the test (step 3a): 50 points
- Improving a test so that it exposes an error in the implementation (step 2b): 100 points

To avoid the risk of a student intentionally submitting faulty solutions to earn more points, the full points listed above were only awarded if a student found an error in, or improved upon another students' problem. To not entirely discourage iterative refinement on one's own problems (e.g., fixing an error just exposed by an improved test), students were awarded a third of the points in the list above for these cases.

Finally, to further integrate the system into the course, Progvis was used during two of the computer labs in the course. Furthermore, students who collected more than 400 points before the final exam were awarded with bonus points corresponding to 3% of the maximum score on the final exam.

5.1 Results From the Second Year

We observed a slight increase of students who used the system in the second year compared to the first year. This year, 90 students signed up to the computer lab assignments and 85 of them solved at least one of the lab assignments. Furthermore, 15 of these students signed in to the system at least once (i.e., 17% of the students who signed up). Out of these, 10 students were considered active as they found at least one error. They found between 1 and 13 errors (8.1 on average). Eight out of these students submitted at least one improvement.

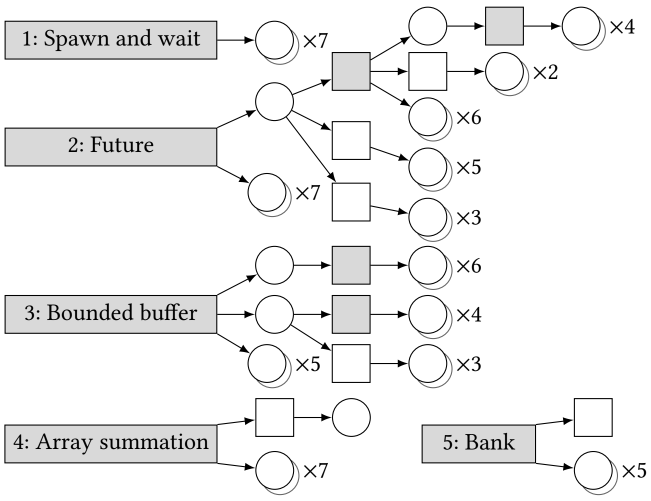


Figure 4: Overview of the usage of the system during the second year. Squares and rectangles represent problems where the test exposes an error in the implementation. Circles represent problems where the associated test does not expose an error in the implementation. Problems with a gray background were uploaded by the lecturer.

These 8 students submitted between 7 and 13 improvements (9.5 on average). Due to the added complexity of splitting problems into test and implementation, only five initial problems were uploaded to the problem bank this year. All of them were adapted from problems used in the first year. In spite of the lower number of initial problems, students found 81 errors in total, three of which were cases where the same student found different types of errors in the same problem. Furthermore, a total of 73 improved implementations were submitted which means that 90% of students who found an error submitted an improved implementation.

In the second year there were many cases where students further improved other students' problems. As such, the table view of students' interactions used to summarize the results of the first year is not suitable for the data from the second year. Instead, we illustrate the students' interactions with the five initial problems in Fig. 4. Each shape represents a problem in the problem bank. Rectangles with labels in them represent the five original problems submitted by the teacher. For all of these problems, the test exposed errors in the implementation. Improved problems are represented by squares and circles. Squares represent problems where the test exposes errors in the implementation, and circles represent problems where the associated test does not expose any errors. Shapes with gray background are problems (initial or improved) submitted by the teacher. To aid the readability of the figure, identical symbols are merged into one as indicated by the notation $\times N$.

From Fig. 4 we can see that 70 out of the 73 improvements to implementations solved all issues exposed by the corresponding test (i.e., all but 3 arrows from squares or rectangles point to circles in Fig. 4, one in each of problems 2, 4 and 5). This is not surprising in and of itself, as the students were notified about the correctness of their solution before submitting it. However, the fact that 81 errors were found, and 70 correct improved implementations were submitted means students managed to correctly fix the error they found in 86% of cases. Furthermore, we were pleasantly surprised to see that three students submitted improvements to tests that successfully exposed problems in other students' solutions. We were initially worried that this might be too difficult for novices, which is why the teacher submitted improved tests in cases where student-submitted improved implementations were incorrect.

Since all but three implementations were correct in the sense that the test program no longer exposed any errors in them, the errors in the improved implementations were due to not accounting for additional threads or some aspect of the abstraction. One example of this is the topmost sequence of improvements to problem 2 in Fig. 4. The first improvement failed to consider that it should be possible to retrieve the value from a future more than once, and the implementation would deadlock in such cases. The improved test submitted by the teacher illustrated this issue by attempting to retrieve the value twice (this case was intentionally omitted in the initial version). The student then improved the implementation once again. This time the implementation was almost correct. A value of zero was used to indicate the absence of a result, and as such the implementation would not work if a zero was posted in the future. This issue was highlighted by the next improved test submitted by the teacher, and subsequently fixed by four students. Another example is the two incorrect improved implementations submitted to problem 3. Both of them only considered the case where one thread added values and another thread removed values (as was the case in the initial test), but failed to account for the fact that multiple threads might add or remove values concurrently. As such, the improved tests all spawned another thread that added values to highlight this issue. As indicated in the figure, this issue was then fixed by multiple students.

Finally, to evaluate students' impressions of the system, we also sent a survey to the 15 students who had signed in to the system at the end of the course. We received 5 responses, out of which one noted that they did not use the system after signing in once. The remaining four responses were positive towards the system, highlighting the extra opportunities for practice, the benefit of the two types of improvements, and problems that caused use-after-free errors. The responses also highlighted that the documentation regarding what types of synchronization primitives were available were lacking and could be improved. They also noted that it was difficult to browse the problem bank when it was filled with many improvements to the same problem, which was the case towards the end of the course.

5.2 Discussion of Results From the Second Year

By comparing students' usage of the system in the first year (Table 1) to the usage in the second year (Fig. 4) we can see that the number of submissions without concurrency issues remained high. From Fig. 4 we can also see that more students improved on other students' problems, and thereby giving other students feedback on their submitted improvements. One reason for this could be that the title of improved problems was changed from "Solution to X" to "Improvement to X" in order to better reflect that the improved problems were not necessarily correct. We do, however, believe that the main reason for the improved engagement was that the three-part problems provided more opportunities to iteratively improve the problems. As previously mentioned, this can be seen in a number of places in Fig. 4, where both the teacher and students provided improved tests that exposed new errors in an improved implementation. When this happened, the author of the improvement found the improved test, and were thus able to further improve their implementation to fix the new errors that were exposed. In the example for problem 2 as mentioned above, this happened three times before no new errors were found in the implementation.

As mentioned previously, we were also pleasantly surprised to find that students attempted and managed to find errors in other students' improved implementations. Finding errors in an implementation in this way is an important skill when working with larger concurrent programs, and it is therefore worthwhile to practice this skill. As can be seen in Fig. 4, the second iteration of the system managed to engage at least a few students to practice this skill. As an additional benefit, these students also provided valuable feedback to their peers in areas where feedback from the model-checker is lacking.

The usage of three-part problems might also be a reason for the slight increase in usage from the first year to the second (alongside the allure of receiving bonus points on the final exam). This type of problems allow students to look at correctness from a different perspective. Instead of treating correctness as the absence of concurrency errors in a particular program, students need to consider whether or not their implementation mirrors the semantics that are described by the abstraction they are implementing. This is something that we have previously found students to struggle with when solving concurrency problems [20], and is thus also an aspect that is worthwhile to illustrate. Furthermore, the combination of the model-checker and peer-grading used in the system means that students both get immediate feedback on whether they have made progress towards a correct solution or not (i.e., if they have fixed all current errors in the implementation, or if their improved test found new errors), and they eventually receive feedback from their peers regarding whether their improved implementation still have inconsistencies compared to the specification (where the model-checker is unable to provide feedback). As the system relies on model-checking and peer-grading, this can be achieved without teacher or TA intervention, at least if enough students use the system regularly.

Even though the usage of the system increased slightly in the second year, the number of students who used it actively was still lower than we had hoped (11% of students who signed up for the lab assignments). As mentioned in Section 4.2, many students experience the lab assignments in the course as very time-consuming. This is likely a large reason for the low usage of the system: only students who managed to solve the lab assignments relatively quickly perceived that they had time to spare to engage with the optional assignments in the system. Another reason for the low usage could, of course, be that students do not find the system valuable. This is, however, likely not the main issue since the few students who answered the survey were positive towards the system. Furthermore, the slight increase in the number of students who used the system indicates that the second iteration of the system is at least better at engaging students compared to the first version.

5.3 Threats to Validity

Due to the small number of self-selected students who used the system, the trends described in this paper should be interpreted with some care. First and foremost, due to the fact that many students find the course to be time-consuming, it is very likely that only students who did not need to spend as much time on the lab assignments found the time to use the system. As such, students with high programming skills are likely over-represented in this population, which means that the results regarding quality of solutions are likely biased towards correct solutions. This is not a problem in and of itself for the purposes of the conclusions in this paper, as we are not interested in students performance when solving the problems. Rather, we are interested in understanding how students use the system, so that we can better understand if it manages to engage students in the peer-grading, and if students find the system valuable. Of course, the skewed cohort likely has an impact on these aspects as well, but we do not believe these effects to be as large as for other metrics, such as correctness.

In the second year, we observed a slight increase in the number of students who used the system compared to the first year. While this change is likely at least partially due to the changes in the system, there are other possible reasons as well. As previously mentioned, the use of Progvis was further integrated into the lab assignments in the course in the second year. These changes led to more students using Progvis during the lab assignments in the second year, which likely has an impact on the number of students who considered using the proposed system. Awarding bonus points on the final exam likely had a similar impact as well. As such, it is difficult to attribute the increased usage to any one of these three factors. We do, however, believe that all of these factors are important to consider when the goal is to encourage students to practice concurrent programming using an approach similar to ours.

5.4 Future Improvements

While the data suggests that the second iteration of the system had some success in encouraging students to provide feedback to their peers in the form of improved tests implementations, there are areas for further improvement. First and foremost, the number of students who used the system in the second year was still lower than we hoped. As previously mentioned, this is likely partially due to the time required for the lab assignments in the course, but it is likely also affected by the degree of integration in the course and how useful the students perceive the system to be. The two latter aspects can be addressed by further promoting the benefits of the system during lectures, and posting more information on the course webpage, for example. Another approach would be to make it mandatory for students to interact with the students to some degree during the lab assignments. This would have the additional benefit of providing data from more students, which would allow a more thorough analysis than what was possible in this paper.

The survey from the second year also highlighted two additional areas of improvement. As both of these issues are related to usability aspects of the system, addressing them will likely also have a positive impact on the number of students who use the system. The first of these issues is the lack of good, and easily accessible documentation on certain aspects of the system. In particular, students noted that it is not easy to find information on which synchronization primitives are available, and how they are used. This was initially not deemed necessary, as the available synchronization primitives are the same as those used in the lab assignments, but clear documentation on the subject is always helpful. Furthermore, the C frontend in Progvis has some limitations that should also be documented clearly.

The second, and perhaps largest, issue that should be addressed is the design of the user interface students use to browse the problem bank. In the second version of the system, students selected problems from one of two lists in a tabbed interface: one for problems where the implementation should be modified, and another for problems where the test should be modified. As noted by some students in the survey, these lists quickly became difficult to navigate since they contained all improvements that all other students had submitted. This approach also has the drawback that students who start using the system later on in the course have access to improved tests compared to students who started earlier (if they manage to find them in the many items in the lists). To address these problems, it would therefore be advisable to re-design this part of the user interface. One possibility would be to separate the process of selecting a problem into two steps. First, the user would select which initial problem to work with. After that, the system would only show problems that are improvements to that initial problem (either directly or indirectly). If this approach is chosen, the system could also require that every user attempts to solve the initial problem before being able to view other students' improvements, thus giving all students an equal starting point regardless of when they started using the system. Another option would be to separate the tests from the implementations. Instead of bundling a specific implementation with a specific test, the system could instead maintain a set of different tests that can be applied to any implementation to see whether the test exposes some concurrency errors in that implementation, similarly to the approach described by for example Wrenn et al. [23, 24]. This way, students would be able to collaborate to create an ever-improving pool of tests that can be applied to any students' implementation. The model-checker could then be used to pick relevant combinations of tests and implementations that students for students to practice finding and fixing errors in the implementations.

6 CONCLUSION

In this paper we have presented two versions of a peer-grading system that we added to Progvis in order to encourage students to practice their concurrent programming skills. The system aims to do this by supplying interesting problems for students to solve, and to provide (semi-)automatic feedback through peer-grading. We tested the system in a course on concurrency and operating systems over the course of two years, which allowed us to iteratively refine the system for the second year. We found that students who used the first version of the system, which focused solely on correctness, managed to solve the problems in the global problem bank. However, only one student attempted to find errors in other students' problems. This shortcoming was addressed in the second version of the system by relying on the model-checker to assess correctness, and instead focus the peer-grading efforts on assessing whether the implementation of an abstraction corresponded to the intended behavior of the abstraction. This was achieved by separating the implementation of the abstraction from the code that tests the abstraction. The data from the second year indicates that these changes were successful in achieving a higher level of peer-grading, even though the number of participants were still smaller than we had hoped. Additionally, data from the second year revealed some areas for further improvement, mainly regarding to how students browse the problems in the problem bank, to further make the system easier and more rewarding to use.

In conclusion, we believe that the system described in this paper (particularly the second version) shows a promising approach for providing students with additional exercises that students can use to further practice concurrent programming in addition to other, more traditional forms of teaching where teaching staff are available. Since the system relies on model-checking and peer-grading to give students feedback, this can be done without much extra work from teachers, and without students having to wait for the teachers to be available to assess their solutions. Furthermore, we believe that the style of problems used in the second version of the system are interesting on their own, as they are able to highlight many subtle aspects that need to be taken into consideration in a concurrent system. Due to the focus on the correspondence between the specification of an abstraction and its implementation, this type of problems might be suitable in other contexts as well, for example to practice software testing.

Finally, it is worth reiterating that the tool is not aimed to replace other forms of teaching, such as laboratory sessions with TAs. Rather, it is aimed to provide additional opportunities for students to practice outside of scheduled sessions without incurring much additional work for the teaching staff. The system presented in this paper is freely available at https://storm-lang.org/.

REFERENCES

[1] Mordechai Ben-Ari. 2007. Teaching Concurrency and Nondeterminism with Spin. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Dundee, Scotland) *(ITiCSE '07)*. Association for Computing Machinery, New York, NY, USA, 363–364. https://doi.org/10.1145/1268784.1268936

[2] Mordechai Ben-Ari and Yifat Ben-David Kolikant. 1999. Thinking Parallel: The Process of Learning Concurrency. In *Proceedings of the 4th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education* (Cracow, Poland) *(ITiCSE '99)*. ACM, New York, NY, USA, 13–16. https://doi.org/10.1145/305786.305831

[3] Sung-Eun Choi and E. Christopher Lewis. 2000. A Study of Common Pitfalls in Simple Multi-Threaded Programs. *SIGCSE Bull.* 32, 1 (March 2000), 325–329. https://doi.org/10.1145/331795.331879

[4] Chris Exton. 2000. Elucidate: A Tool to Aid Comprehension of Concurrent Object Oriented Execution. *SIGCSE Bull.* 32, 3 (July 2000), 33–36. https://doi.org/10.1145/353519.343066

[5] Kathi Fisler, Shriram Krishnamurthi, and Preston Tunnell Wilson. 2017. Assessing and Teaching Scope, Mutation, and Aliasing in Upper-Level Undergraduates. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 213–218. https://doi.org/10.1145/3017680.3017777

[6] Oscar Karnalim and Mewati Ayub. 2017. The Effectiveness of a Program Visualization Tool on Introductory Programming: A Case Study with PythonTutor. *CommIT (Communication and Information Technology) Journal* 11, 2 (2017), 67–76.

[7] Yifat Ben-David Kolikant. 2001. Gardeners and Cinema Tickets: High School Students' Preconceptions of Concurrency. *Computer Science Education* 11, 3 (2001), 221–245.

[8] Yifat Ben-David Kolikant. 2004. Learning Concurrency as an Entry Point to the Community of Computer Science Practitioners. *Journal of Computers in Mathematics and Science Teaching* 23, 1 (2004), 21–46.

[9] Yifat Ben-David Kolikant. 2004. Learning concurrency: evolution of students' understanding of synchronization. *International Journal of Human-Computer Studies* 60, 2 (2004), 243–268. https://doi.org/10.1016/j.ijhcs.2003.10.005

[10] Yifat Ben-David Kolikant. 2005. Students' Alternative Standards for Correctness. In *Proceedings of the First International Workshop on Computing Education Research* (Seattle, WA, USA) *(ICER '05)*. ACM, New York, NY, USA, 37–43. https://doi.org/10.1145/1089786.1089790

[11] Aubrey Lawson and Eileen T. Kraemer. 2020. Sidekicks and Superheroes: A Look into Student Reasoning about Concurrency with Threads versus Actors. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training* (Seoul, South Korea) *(ICSE-SEET '20)*. Association for Computing Machinery, New York, NY, USA, 82–92. https://doi.org/10.1145/3377814.3381706

[12] Aubrey Lawson, Eileen T. Kraemer, S. Megan Che, and Cazembe Kennedy. 2019. A Multi-Level Study of Undergraduate Computer Science Reasoning about Concurrency. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE '19)*. Association for Computing Machinery, New York, NY, USA, 210–216. https://doi.org/10.1145/3304221.3319763

[13] Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A Uronen. 2003. The Jeliot 2000 program animation system. *Computers & Education* 40, 1 (2003), 1–15. https://doi.org/10.1016/S0360-1315(02)00076-3

[14] Gary Lewandowski, Dennis J. Bouvier, Robert McCartney, Kate Sanders, and Beth Simon. 2007. Commonsense Computing (Episode 3): Concurrency and Concert Tickets. In *Proceedings of the Third International Workshop on Computing Education Research* (Atlanta, Georgia, USA) *(ICER '07)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/1288580.1288598

[15] Jan Lönnberg. 2012. *Understanding and Debugging Concurrent Programs through Visualisation*. G5 Artikkeliväitöskirja. Aalto University. http://urn.fi/URN:ISBN:978-952-60-4530-6

[16] Jan Lönnberg, Anders Berglund, and Lauri Malmi. 2009. How Students Develop Concurrent Programs. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95* (Wellington, New Zealand) *(ACE '09)*. Australian Computer Society, Inc., Darlinghurst, Australia, 129–138. http://dl.acm.org/citation.cfm?id=1862712.1862732

[17] Anna Offenwanger and Yves Lucet. 2014. ConEE: An Exhaustive Testing Tool to Support Learning Concurrent Programming Synchronization Challenges. In *Proceedings of the Western Canadian Conference on Computing Education* (Richmond, BC, Canada). Association for Computing Machinery, New York, NY, USA, Article 11, 6 pages. https://doi.org/10.1145/2597959.2597972

[18] Juha Sorva. 2012. *Visual Program Simulation in Introductory Programming Education*. Ph. D. Dissertation. Aalto University, Helsinki, Finland. http://lib.tkk.fi/Diss/2012/isbn9789526046266/

[19] Filip Strömbäck, Linda Mannila, Mikael Asplund, and Mariam Kamkar. 2019. A Student's View of Concurrency - A Study of Common Mistakes in Introductory Courses on Concurrency. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. ACM, New York, NY, USA, 229–237. https://doi.org/10.1145/3291279.3339415

[20] Filip Strömbäck, Linda Mannila, and Mariam Kamkar. 2021. The Non-Deterministic Path to Concurrency – Exploring how Students Understand the Abstractions of Concurrency. *Informatics in Education* 20, 4 (2021), 683–715. https://doi.org/10.15388/infedu.2021.29

[21] Filip Strömbäck, Linda Mannila, and Mariam Kamkar. 2022. Pilot Study of Progvis: A Visualization Tool for Object Graphs and Concurrency via Shared Memory. In *Australasian Computing Education Conference* (Virtual Event, Australia) *(ACE '22)*. Association for Computing Machinery, New York, NY, USA, 123–132. https://doi.org/10.1145/3511861.3511885

[22] Filip Strömbäck, Linda Mannila, and Mariam Kamkar. 2022. A Weak Memory Model in Progvis: Verification and Improved Accuracy of Visualizations of Concurrent Programs to Aid Student Learning. In *Koli Calling '22: 22nd Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli 2022)*. Association for Computing Machinery, New York, NY, USA, Article 14, 12 pages. https://doi.org/10.1145/3564721.3565947

[23] John Wrenn and Shriram Krishnamurthi. 2019. Executable Examples for Programming Problem Comprehension. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. ACM, New York, NY, USA, 131–139. https://doi.org/10.1145/3291279.3339416

[24] John Wrenn, Shriram Krishnamurthi, and Kathi Fisler. 2018. Who Tests the Testers?. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) *(ICER '18)*. Association for Computing Machinery, New York, NY, USA, 51–59. https://doi.org/10.1145/3230977.3230999