A Weak Memory Model in Progvis: Verification and Improved Accuracy of Visualizations of Concurrent Programs to Aid Student Learning

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 many students struggle with solving small concurrency problems after their first course on concurrency. A possible reason for this is that students do not have a suitable mental model of the semantics of the underlying programming language, and are therefore not able to properly reason about the program's behavior. One way to help students learn concurrency and improve their mental model is through the use of visualization tools. Progvis is one such visualization tool that is not only aimed at concepts related to concurrency, but also provides an accurate visualization of more fundamental concepts to illustrate how they interact with concurrency. In previous work, the authors of Progvis performed a small-scale evaluation of the tool, and highlighted some areas of improvement. In this paper, we address these shortcomings by improving the memory model visualized by Progvis and implementing a model checker. We also evaluate Progvis on a larger scale by incorporating it into a course on concurrency and operating systems, which allows assessing whether using Progvis aids students in learning concurrency. The results indicate that Progvis (with our improvements) is successful in helping students realize how concurrency interacts with more fundamental concepts, and that students find it useful in helping them understand the content of the concurrency assignments.

CCS CONCEPTS

• **Applied computing** → **Education**; • **Software and its engineering** → **Model checking**; • **Human-centered computing** → *Visualization systems and tools*; • **Theory of computation** → *Concurrency*.

KEYWORDS

concurrency, synchronization, visualization, computer science education

ACM Reference Format:
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 2022), November 17–20, 2022, Koli, Finland.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3564721.3565947

1 INTRODUCTION

The ability to work with concurrent programs, which we refer to as *concurrent programming*, is an increasingly important skill in order to be able to utilize the many cores available in modern systems. Furthermore, as the non-deterministic behavior of concurrent programs requires reasoning about their behavior rather than testing, Kolikant [11] argues that learning concurrency is also useful for developing students' formal reasoning skills which are applicable in other contexts. Previous work has shown that students struggle with solving simple concurrency problems [24]. One possible reason for this is that students are not yet proficient enough with fundamental concepts and are thus not able to reason properly about the concurrent program. One set of such skills that are central to concurrent programming is scope, mutation, and aliasing, which students have been found to struggle with in their first three years of studying CS [8].

Many tools have been proposed to aid students in learning concurrent programming. One such example is Progvis [26], which not only focuses on concurrency, but also how concurrency interacts with more fundamental concepts. In the paper where we introduced Progvis, we also conducted a small pilot study with eight students, and noted that Progvis assumed a too strong memory model which prevented it from detecting data races. In this paper, we aim to address this shortcoming by implementing an improved memory model that more accurately depicts the behavior of concurrent programs. To complement the improved memory model, we also implement a model checker to help students verify the correctness of concurrent programs. Lönnberg et al. [20] notes that this has the additional benefit of providing students with an example that illustrates why an incorrect program could fail. Finally, we also expand the scope of the aforementioned pilot study by integrating Progvis into a course on concurrency and operating systems to find whether it has any impact on students' learning. As such, we aim to answer the following research questions:

RQ1 How can the memory model in Progvis be improved to be more accurate while remaining easy to visualize in a way that is easy for students to understand?

RQ2 To what extent do students find Progvis helpful when solving lab assignments on concurrent programming?

RQ3 To what extent does using Progvis help students learn concurrent programming?

The remainder of this paper is structured as follows: Section 2 provides a brief introduction to the relevant parts of concurrency, followed by related works in Section 3. We then present our improvements to Progvis in detail in Section 4, followed by a description of the method used to evaluate Progvis in Section 5. Finally, we present the results in Section 6, discuss the results to answer the RQs in Section 7 and conclude the paper in Section 8.

2 BACKGROUND: CONCURRENCY

In this paper we work with concurrency using the shared-memory model (as opposed to e.g., message-passing). In this model, a concurrent program consists of one or more *threads* that execute concurrently. Threads associated with the same process share memory and are thereby able to communicate. As such, when writing concurrent programs it is important to consider the semantics of memory accesses as described by the *memory model*. The simplest and perhaps the most intuitive memory model is sequential consistency [13]. In this model, one can consider the memory accesses made by each thread to happen in some non-deterministic, global order. As such, it is enough to consider all interleavings of the operations from different threads and ensuring that none of them lead to any undesired results.

While sequential consistency is convenient due to its simplicity, it is costly to implement in hardware and disallows many types of optimization by the compiler. Because of this, many languages and processors use a weaker memory model. One such model is the C memory model [2], which is the one we will mainly consider in this paper. In this model, memory accesses are unordered in general. This means that while it is still possible to consider interleavings of operations from different threads, each thread may have its own view of memory. As such, to ensure that stores from one thread are visible to another thread, it is necessary to impose an ordering to the relevant memory accesses, either by using atomic operations, or one of many *synchronization primitives* such as locks, semaphores and condition variables. Failure to protect access to shared memory constitutes a *data race*, the behavior of which the C standard leaves undefined.

3 RELATED WORK

In this section we present an overview of related works. First, we focus on works related to teaching and learning concurrency in general. We then introduce tools that aim to help students to learn concurrent programming.

3.1 Teaching and Learning Concurrency

As mentioned briefly in the introduction, Kolikant [11] argues that learning concurrency is an entry-point into a more formal, academic culture of CS. This culture focuses on using computational models to reason about the behavior of programs, which is important when working with concurrent programming since the inherent non-determinism makes it infeasible to use only empirical methods (e.g., ad-hoc testing by students or unit tests) to test concurrent programs for correctness. This transition between an informal user culture and a formal academic culture can also be seen in the work by Lönnberg et al. [18, 19], who investigated students' approaches to developing concurrent programs. They found approaches ranging from *trial and error* and *coding to understand* which corresponds to the informal user culture to *adapting a known technique* and *adapt known solution* which corresponds to a more formal, academic culture of CS.

As computational models are important when working with concurrent programming, students' experiences of them have also been studied. Ben-Ari and Kolikant [4] found that students' difficulties in learning concurrency can be classified into three categories: 1) the failure to understand the concept of a model (e.g., by inventing new operations), 2) make incorrect assumptions about the model (e.g., assuming ordering of certain unordered operations), and 3) having problems with coordination (e.g., not finding problematic scenarios). Other studies have focused on incorrect assumptions about the computational model. For example, Strömbäck et al. [25] studied student solutions to concurrency problems and found 10 types of such assumptions, related to four broad categories *the memory model*, *the C language*, *synchronization primitives* and *concurrent aspects of other abstractions* (e.g., data structures). Since some of the incorrect assumptions were not related to concurrency, but rather to more fundamental concepts, these results highlight the importance of students having solid foundations when working with concurrent programming. One such set of skills are scope, aliasing and mutation, which Fisler et al. [8] found students in their second and third years to have difficulties with. Additionally, Haglund et al. [9] argue that these fundamental skills are also important to understand and utilize abstractions in programming. Thus, similarly to the reasoning of Kolikant [11], since learning concurrency requires students to understand fundamentals, it might also have a positive impact on the understanding of other CS subjects.

Additionally, many have studied common mistakes to concurrency questions. For example, Kolikant [10, 12] found that the studied high-school students had problems identifying synchronization goals, but were able to solve the problem once the goals were identified. Lawson and Kraemer [14] reached similar conclusions by using similar problems, but also found that students sometimes use *sleep* functions to address concurrency issues instead of synchronization primitives. Others have studied similar problems in an undergraduate context. Both Lawson et al. [15] and Lewandowski et al. [16] studied a question involving selling cinema tickets and found that while students are generally aware of the concurrency issues, they are often unable to address them adequately. Strömbäck et al. [24] studied a problem involving synchronizing a simple data structure with similar results. In this case, the authors hypothesized that some incorrect solutions were due to lacking fundamental skills, in particular related to scope and aliasing.

3.2 Tools for Teaching Concurrency

A number of tools have been developed to help students learn concurrency. While there are many tools that illustrate concurrency concepts in isolation, we will focus on tools that connect these concepts to some form of code as the focus of this paper is concurrent programming. One set of such tools utilize static analysis to help students and teachers to find errors in concurrent programs. Eraser [6] is one such example that uses the lock-set algorithm to identify data races in C programs. Another example is the Spin model checker. While it was not designed with education in mind, Ben-Ari [3] found it simple enough to be usable in teaching. Another similar set of tools collect execution traces and visualize them to help students understand why an error occurred. This is done by Eludicate [7], which lists relevant events from the trace to illustrate how the execution of different threads are interleaved, and how this affects the behavior of the program. Another example is Atropos [17], which uses dynamic dependency graphs to include data dependencies between threads in the visualization to clearly illustrate potential data races and other synchronization issues.

Another approach to visualize concurrent programs is to present the user with the current state of all threads and let the user decide which thread should execute next, and is thereby able to explore different interleavings. One benefit of this approach is that it requires students to actively engage with the visualization, which Sorva [23, ch. 11] shows is beneficial for learning. Two examples of such tools are The Deadlock Empire[1] and ConEE [22]. The Deadlock Empire presents the user with pre-made pieces of code in C# and asks them to find an interleaving that causes a particular error. ConEE supports arbitrary programs in a custom language, and also has the ability to automatically find concurrency errors through exhaustive testing. Both of these tools use a simple data model due to their focus on concurrency. It is therefore not possible to explore how concurrency interacts with other concepts, such as scope, references and aliasing.

Progvis [26] is a visualization tool similar to ConEE and The Deadlock Empire, where students control the program execution. It aims to improve on existing tools by also providing a rich data model that encompasses many of the more fundamental concepts in addition to those relevant to concurrency. In addition to the rich data model it supports visualizing arbitrary programs in a subset of C. This means that students are able to modify example programs to try to solve concurrency errors found within or visualize custom programs, which is not possible in, for instance, The Deadlock Empire. Furthermore, even though supporting C adds complexity to Progvis, it has the large benefit that students do not need to learn a custom language to use the tool, as is the case with ConEE and the Spin model checker. Finally, compared to the tools based on static analysis and execution traces, the ability to actively engage with the visualization in Progvis is something that Sorva [23, ch. 11] argues is beneficial for learning. As mentioned in the introduction, one shortcoming of Progvis, and many other tools, is that it visualizes programs in a sequentially consistent memory model [26], which is not accurate according to the semantics of C and many other languages.

4 IMPROVEMENTS TO PROGVIS

This section describes our improvements to Progvis. First and foremost, we addressed the issues mentioned in our previous paper [26], namely that Progvis visualizes programs using a sequentially consistent memory model and therefore does not report data races. We also implemented a model checker to help students find interleavings that cause the program to misbehave. This helps students to validate their understanding of the computational model as it allows Progvis to highlight problematic interleavings that students may have failed to consider. Our additions to Progvis are available at https://storm-lang.org/.

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

4.1 A Weaker Memory Model and Detection of Data Races

A large benefit of sequential consistency is that it is easy to visualize and explain. Since all operations have a global order, the notion of a unique global state at any point in time is possible. Progvis uses this notion to visualize concurrent programs by pausing each thread after each statement and providing a visual representation of the global scope at that point in time. The user is then able to take the role of a scheduler and decide which thread to execute next. Whenever the selected thread has executed a statement, it is paused yet again, and the global state is updated. The user is thus able to explore different interleavings of the program through their scheduling decisions.

While sequential consistency is convenient in many regards, it does not accurately reflect the semantics of the C memory model. Since Progvis is aimed at students who are novices at concurrent programming, our aim is to implement a memory model that is as simple as possible, yet able to accurately describe the behavior of synchronization primitives (i.e., locks, semaphores and condition variables) and sequentially consistent atomic operations (i.e., relaxed atomics are considered out of scope). In this context, the difference from sequential consistency is that concurrent access to shared data is not well-defined. In particular, two or more threads that access the same data and at least one access is a store, is considered a *data race*, which is undefined according to the C standard [2]. As such, data races must be avoided using the available synchronization primitives or atomic operations.

One possible way to visualize this weak memory model is to use execution graphs similarly to Atropos [17]. However, to keep the simplicity of the existing visualization in Progvis we opted to keep the serialized execution of sequential consistency and model the non-determinism using *fence* instructions. Each fence instruction represents a location where all stores performed by the current thread are published to the other threads, and data published by other threads become visible to the current thread. This means that we can consider the runtime behavior of a thread as a series of *chunks* separated by fence instructions. Chunks can then be viewed as units, as any stores from one chunk are only guaranteed to be visible to another thread after both have advanced to a subsequent chunk. This allows us to model each synchronization operation (i.e., synchronization primitives and atomic operations) by considering them to emit a suitable fence instruction. It is, however, not possible to model the intricacies of relaxed atomics, which were considered out of scope. For the purpose of the model checker, the implementation additionally distinguishes between *acquire* and *release* fences, which will be introduced in detail in Section 4.2. To detect whether a data race has occurred, we need to consider the memory accesses

```
int shared_a;
int shared_b;
struct semaphore sema;

void worker(void) {      ← Wb
    shared_a = 2;
    shared_b = 3;
    sema_up(&sema);      ← W1 (release)
}                        ← We
```

```
int main(void) {             ← Mb
    sema_init(&sema, 0);
    thread_new(&worker);     ← M1 (release)
    shared_a = 8;
    sema_down(&sema);        ← M2 (acquire)
    shared_b = 9;
    printf("A: %d\n", shared_a);
    printf("B: %d\n", shared_b);
    return 0;                ← Me
}
```

Figure 1: Source code of a concurrent program used to illustrate the improvements to Progvis. Lines marked with arrows denote operations Progvis consider to be fences. Wx denote fences in the `worker` function, and Mx denote fences in the `main` function. The beginning (Wb and Mb) and end (We and Me) of the two functions are implicitly considered fences for convenience.

since the last fence instruction in each thread. Any stores represent modified data that is not guaranteed to be visible to other threads. This means that if another thread has accessed the same data (either loading or storing), a data race has occurred and Progvis should notify the user accordingly.

To illustrate this idea, consider the program in Fig. 1. The code contains three operations that involve a fence (W1, M1 and M2) as well as four implicit fences at the beginning (Wb, Mb) and end (We, Me) of each function. As such, we can consider the code to consist of five chunks, two in `worker` (Wb–W1 and W1–We) and three in `main` (Mb–M1, M1–M2, and M2–Me). The `main` function starts by executing the chunk Mb–M1 which initializes a semaphore and starts a second thread. Starting a thread involves a fence as the previous initialization must be visible to other threads. After this, the two threads execute concurrently. The first thread executes M1–M2 and stores 8 in `shared_a`, then waits for the second thread. The second thread executes Wb–W1 which stores 2 to `shared_a` and 3 to `shared_b`, then signals the first thread. In this case, the two chunks end with fences caused by the semaphore operations. At this point, just before the two threads execute their fence instructions we observe that both threads have stored into `shared_a`, which means that a data race has occurred. After this point, the second thread executes its final (mostly empty) chunk W1–We and terminates, while the first thread executes M2–Me, which stores 9 to `shared_b` and loads both shared variables to print them. At this point, no data race is detected as the second thread has not accessed either of the shared variables since executing a fence instruction.

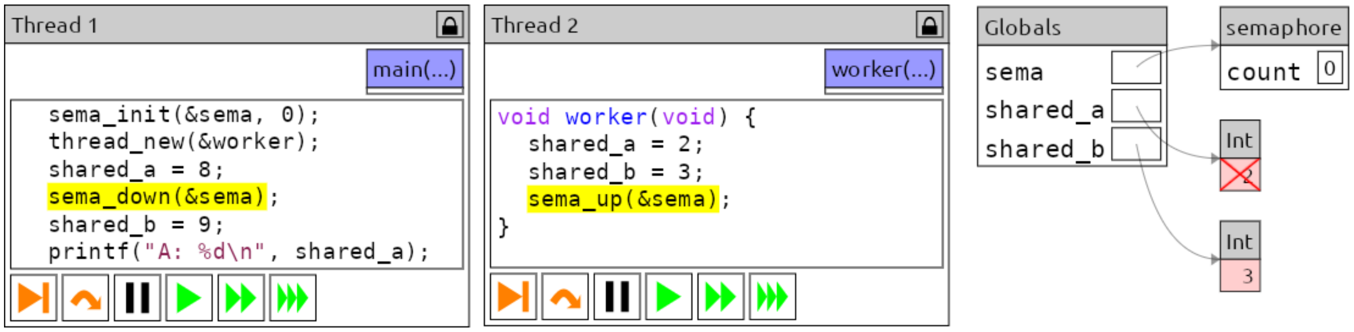
This model was implemented in Progvis by maintaining two sets for each thread: one for loads, and one for stores. To populate these sets, we modified the instrumentation of the visualized program done by Progvis to pause the program after each statement, so that it also tracks all memory accesses and fences. Finally, we implemented the above-mentioned check for data races whenever Progvis pauses the visualized program. Whenever a data race is detected any data involved in a data race is highlighted with a red cross as shown in Fig. 2, and a message explaining the problem is shown. As can be seen in Fig. 2, the collected memory accesses are also used to highlight loads and stores in the visualization. Loads are colored green while stores are colored red. In the figure, the red background of the two integer variables mean that the program has recently stored data there. This is useful not only in concurrent programs, but also to highlight the semantics of more complex statements in sequential programs (e.g., modifying data through a pointer).

4.2 The Model Checker

The improved memory model allows Progvis to detect data races if the user finds a suitable interleaving. While this is trivial to do in the program in Fig. 1, it quickly becomes difficult in more complex programs. This means that students who are still learning concurrent programming may inadvertently fail to consider some important interleaving, perhaps due to a misunderstanding of some part of the computational model. Thus, we implemented a model checker to help students find issues, and thereby also helping them validate their understanding of the computational model. The model checker is similar to that of ConEE [22] as it exhaustively checks all possible interleavings for concurrency errors. However, since the number of interleavings grow quickly with program size and number of threads, we reduce the number of interleavings that are necessary to evaluate through two observations:

(1) When using the memory model from Section 4.1 it is enough to consider interleavings of entire chunks rather than individual statements.
(2) Many interleavings result in the same program state. Such interleavings can be treated as equivalent during further exploration to reduce the search space. For example, consider a program with two threads, 1 and 2. Thread 1 executes chunk A, and thread 2 executes chunks B and C. If we find that the interleaving A, B produces a state identical to B, A, then we can conclude that interleavings B, A, C and B, C, A must be equivalent, and only evaluate one of them.

To utilize observation (1) we implemented a *fence step* operation that advances a thread to the next fence operation rather than the next statement (as is the case for a *single step*). The model checker is then able to use the fence step operation to explore interleavings of entire chunks rather than individual statements. However, a naïve implementation of this operation has two issues that need to be addressed. The first issue is that loops without fence instructions may cause problems. In particular, if a loop waits for the value of some variable to change without proper synchronization (which is a natural first attempt for novices), a fence step would



```
sema_init(&sema, 0);
thread_new(&worker);
shared_a = 8;
sema_down(&sema);
shared_b = 9;
printf("A: %d\n", shared_a);
```

```
void worker(void) {
  shared_a = 2;
  shared_b = 3;
  sema_up(&sema);
}
```

Figure 2: Progvis showing an error in the program in Fig. 1. Thread 1 is about to wait for thread 2 to finish. At this point thread 1 has written to `shared_a`, and thread 2 has written to the variables `shared_a` and `shared_b`. This leads to a data race in `shared_a`, which is indicated by a red cross.

wait indefinitely for the thread to reach a fence. We solve this by having Progvis identify all *back edges* in the program execution (i.e., jumps backwards in the code), and pause program execution at the statement following the execution of a back edge.

The second issue is that all interleavings need to be visible to the model checker. A naïve implementation that pauses execution immediately before a fence operation fails in cases where the operation immediately after the fence could wake a thread. For example, consider a program where two threads are executing. Thread 1 is paused before a fence instruction just before releasing a lock, and thread 2 is waiting for the lock to be released. In this case, a naïve fence step of thread 1 would cause it to release the lock and then execute the remainder of the chunk, without allowing for the possibility that thread 2 executes its chunk before thread 1. While this is not a problem for data race detection, it may prevent the model checker from detecting other problems (e.g., crashes). To allow thread 2 to execute its chunk before thread 1, the implementation distinguishes between *acquire* and *release* fences (*acquire* and *release* fences are also supported). An acquire fence is emitted for operations that may wait for some event (e.g., `lock_acquire`), and a release fence is emitted for operations that may wake a sleeping thread (e.g., `lock_release`). Using this information, we solve this problem by making fence steps starting from a release fence halt execution before the next statement rather than the next fence. Similarly, threads that have been sleeping (e.g., waiting to acquire a lock) are also halted before the next statement in a similar manner. With these modifications, the fence step allows the model checker to explore all interleavings of chunks in the program. The fence step operation was also exposed to users through a new button in the interface to help navigate larger programs (the second button from the left of the bottom of each thread in Fig. 2).

To utilize observation (2) above to avoid exploring redundant interleavings, we model the visualized program as a transition system [1, ch. 2]. A transition system can be seen as a graph where each node represents a unique state reachable by the program and each edge represents some action that alters the state. In the model checker such an action corresponds to advancing a single thread using a fence step. By associating each unique program state with exactly one node in the graph, situations where different interleavings result in the same state are trivially detected as nodes with more than one incoming edge. Furthermore, paths starting from the initial state in the graph correspond to possible interleavings. We can thus check the program for correctness by exploring each edge at least once. This ensures that we traverse all reachable states while excluding redundant interleavings. As an additional benefit, this scheme also ensures that (infinite) loops are handled correctly and efficiently.

To illustrate the behavior of the model checker Fig. 3 contains a transition system corresponding to the behavior of the program in Fig. 1. Each node (numbered 1–18 for clarity) corresponds to a state reachable by the program. In the figure, the state is presented as two lines of text. The first line lists all running threads (Tx), and the operation each thread is about to execute. Mx and Wx mean that the corresponding fence is about to be executed, while Mx' and Wx' mean that the operation after the fence marker is about to be executed. The second line lists the values of all variables in the program. Here, `a` and `b` correspond to `shared_a` and `shared_b` respectively.

Initially, as indicated by node 1, a single thread (T1) is about to execute the `main` function (Mb). At this point, the values of `shared_a` and `shared_b` are both zero and the semaphore is uninitialized. Since only one thread is active, the only possibility is to advance the program to state 2 by fence stepping this thread. This causes the thread to initialize the semaphore and pause before fence M1. Again, the only option is to fence step this thread as no other threads have been started. In state 3, however, the first thread has just started a second thread (T2) that is about to execute `worker` (Wb). As such, there are two possibilities to advance the program from state 3: either fence stepping T1 or T2. At this point we can see that program execution diverges. If T1 is executed first, program execution eventually ends in state 18, while if T2 is executed first program execution eventually ends in state 14. Neither of these states have any outgoing edges, which means that all threads have terminated (as indicated by Me and We). A graph that diverges in this manner corresponds to a program that has non-deterministic behavior. This is not necessarily a problem in and of itself, and is therefore not considered an error. If one of the final states are considered invalid in the context of the program, it is possible to use `assert` statements to verify some particular behavior. In this case, however, the problem is due to a data race, which is detected

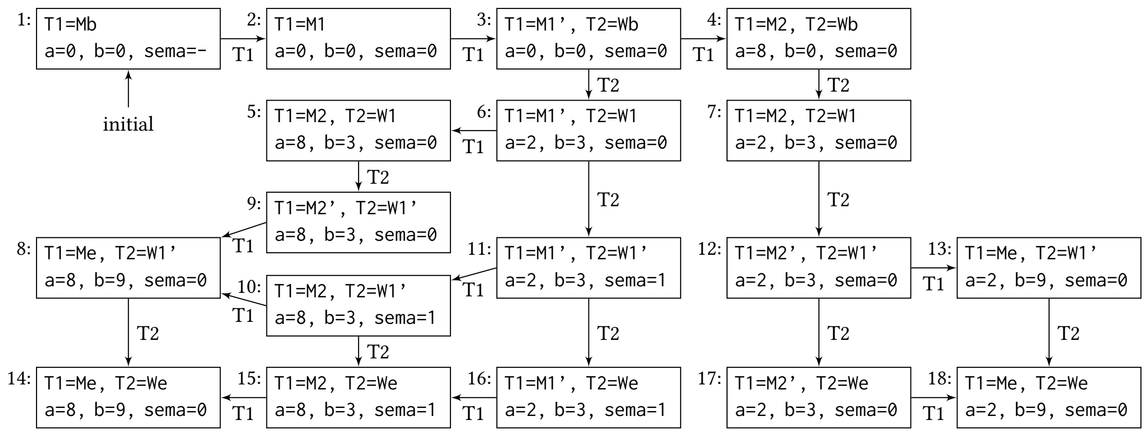

1: T1=Mb a=0, b=0, sema=-
initial
2: T1=M1 a=0, b=0, sema=0
3: T1=M1', T2=Wb a=0, b=0, sema=0
4: T1=M2, T2=Wb a=8, b=0, sema=0
5: T1=M2, T2=W1 a=8, b=3, sema=0
6: T1=M1', T2=W1 a=2, b=3, sema=0
7: T1=M2, T2=W1 a=2, b=3, sema=0
8: T1=Me, T2=W1' a=8, b=9, sema=0
9: T1=M2', T2=W1' a=8, b=3, sema=0
10: T1=M2, T2=W1' a=8, b=3, sema=1
11: T1=M1', T2=W1' a=2, b=3, sema=1
12: T1=M2', T2=W1' a=2, b=3, sema=0
13: T1=Me, T2=W1' a=2, b=9, sema=0
14: T1=Me, T2=We a=8, b=9, sema=0
15: T1=M2, T2=We a=8, b=3, sema=1
16: T1=M1', T2=We a=2, b=3, sema=1
17: T1=M2', T2=We a=2, b=3, sema=0
18: T1=Me, T2=We a=2, b=9, sema=0
T1
T2

Figure 3: The behavior of the program in Fig. 1 modeled as a transition system. The first line in each node list the running threads (Tx) and the operation each thread is about to execute: W*x* and M*x* denote that the corresponding fence is about to be executed. W*x*' and M*x*' denote that the operation after the fence is about to be executed. The second line contains the value of all variables in the program. Edges correspond to advancing the corresponding thread using a fence step (horizontal for T1 and vertical for T2).

when states 5 and 7 are reached. Finally, the graph has been slightly simplified for readability. Whenever thread T1 reaches fence M2, the implementation does not know if the thread has to wait for the semaphore to be signaled. As such, the thread is technically paused once at M2, and again by the semaphore if it has to wait. These situations would result in an additional state which is not shown in Fig. 3 to improve readability.

In the implementation in Progvis each node contains a list of threads that are ready to execute and a string representation of the entire program state. This includes a hierarchical representation of all (global and local) variables in the program as well as memory accesses performed by each thread, and the next operation to be executed by each thread. The model checker is then implemented as a graph search. First, a node corresponding to the initial state of the program is created. The model checker then finds a node with an unexplored edge (i.e., a node where a thread is ready to execute but no corresponding edge exists in the graph), and advances the program to that state through a series of fence steps. Then, a fence step corresponding to the unexplored edge is performed and the resulting state is recorded. If this state does not already exist in the graph, a new node is created. Finally, a new edge is added to the graph, and the process of finding unexplored edges is repeated. If an error is found at any time during this process it is reported and the user is presented with a visualization of the interleaving that caused the error. Otherwise, if all edges are successfully explored, the model checker reports success.

5 METHOD

In addition to our proposed improvements to Progvis, we also aim to expand the short-term evaluation made by the authors of Progvis [26]. In particular, we evaluate whether using Progvis aid students in learning concurrent programming by integrating Progvis into a course on concurrency and operating systems in the year 2022 and compare students' performance on the final exam to previous years.

5.1 The Course

The course on concurrency and operating systems is given during 10 weeks towards the end of the second year of a three-year bachelor program in computer science at Linköping University. The course consists of six lectures and a series of computer lab assignments. The lectures briefly introduce C programming (as students have previously mainly worked in C++), and then focus on concurrency as students are expected to be familiar with operating system theory from a previous course. The main focus of the course is on the lab assignments, where students work in pairs and implement a number of system calls in the educational operating system Pintos.[2]

As can be seen from Fig. 4, the lab assignments are expected to be solved during the first 8 weeks of the course. During this time, students have two or three 2-hour sessions scheduled in a computer lab with a TA and are thus able to ask questions and demonstrate their solutions. Week 9 is a self-study period where students have time to prepare for the exam in week 10. The content and order of the lab assignments has remained the same for the last four years (2018–2021), and were as follows:

Introduction to C: Familiarizes students with programming in C through two basic assignments.

Basic system calls: Introduces the mechanisms behind system calls by implementing two simple system calls.

File I/O: Implements the system calls `open`, `close`, `read` and `write` using an existing file system implementation. The focus is therefore on implementing a per-process table for storing file descriptors.

Stack initialization: Involves writing code to initialize the execution stack of newly created processes to allow passing command-line parameters to the `main` function.

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

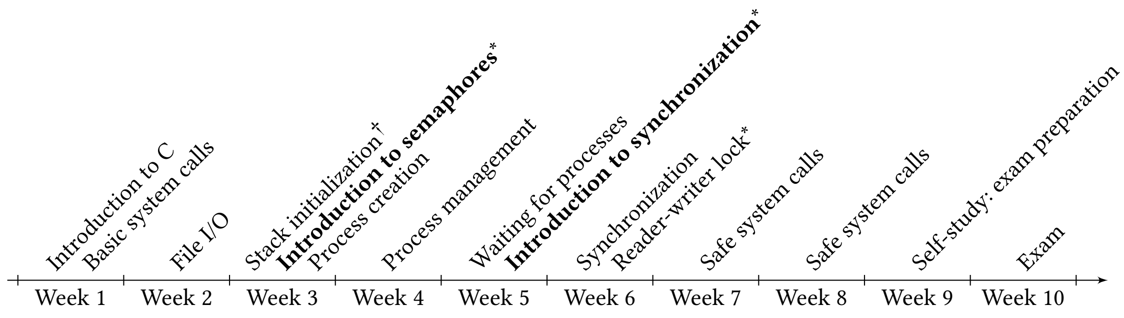


Figure 4: Overview of the structure of the course. The assignments in bold were added to integrate Progvis into the course, and the assignment marked with a dagger (†) was removed. It is possible to use Progvis in assignments marked with an asterisk (*).

Process creation: In Pintos, a new process is created by creating a new kernel thread and letting that thread load the executable. The original thread thus needs to wait for the new thread to complete in order for errors and process IDs to be propagated. Students typically solve this using a semaphore.

Process management: Implements a global data structure that stores all running processes for later use.

Waiting for processes: Uses the data structure from *process management* to implement the system call `wait` to allow waiting for processes to exit. Again, students typically solve this using a semaphore.

Synchronization: Ensures that no data races exist in the existing file system implementation, and in the table implemented in *process management*. This involves identifying critical sections and protecting them using locks.

Reader-writer lock: Implements a *reader-writer lock* to allow multiple processes to read from the same file concurrently, while ensuring mutual exclusions for processes writing to a file.

Safe system calls: Ensures that processes are not able to pass invalid data to system calls and crash the kernel. This mainly involves verifying that pointers refer to valid memory.

Even though the structure of the labs have been the same since 2018, some minor changes have been made. Most notably the course was given remotely the years 2020 and 2021 due to the ongoing pandemic. Additionally, the instructions for the assignments were re-structured in 2019. This did, however, only change the wording and the order in which information was presented, not the assignments themselves. Furthermore, Progvis was available to students (without our additions) in years 2020 and 2021. It was, however, not possible to use it to visualize the lab assignments since Pintos contains constructs that are not supported by Progvis. Thus, using Progvis requires creating a small, external program that illustrates the relevant parts from Pintos.

Usage of Progvis was integrated into the course in 2022 by replacing the assignment *stack initialization* with two assignments where students solve synchronization problems in a small program external to Pintos. This means that students are able to use Progvis to visualize the program, but also that they are able to focus on the task at hand since they do not have to familiarize themselves with the code in Pintos. To keep these assignments relevant, they were designed to resemble the upcoming assignment that would be solved inside the Pintos codebase. While it was possible to compile and test these programs with standard command-line tools, the assignment encouraged students to use Progvis. The new assignments were as follows:

Introduction to semaphores: Involves adding a semaphore to synchronize a program similar to the situation encountered in the *process creation* assignment. The program is similar to the second task from [26].

Introduction to synchronization: Involves using locks to synchronize a program similar to a part of the file system implementation in Pintos. It requires synchronizing both global variables and members of a data structure.

We also added a stand-alone program to allow students to test their implementation of the *reader-writer lock* assignment in Progvis. In this case, however, using Progvis was presented as an option rather than encouraging it to be used. This modification was done in 2021, but the improvements to Progvis (Section 4) were not available until 2022.

A final difference between the years is the gradual introduction of a bank of optional concurrency problems. In 2021, students had the ability to view and solve concurrency problems from an online bank of problems (using Progvis). Each problem simply asked students to find concurrency errors in a piece of code. Only five students used the system 2021. The next year, 2022, this system was improved using the model checker to allow automatically checking if the submitted solution was correct. Even though points for the final exam were awarded for using the system to a certain extent, only eight students participated in 2022. Due to the few participants in both years, we consider the details of this system to be out of scope for this paper.

5.2 Data Collection and Analysis

Two types of data were collected to evaluate Progvis: answers to a questionnaire, and students' performance on their final exam. The questionnaire was sent to students by e-mail after their final exam with the aim of assessing students' impressions of using Progvis during the lab assignments. As such, the questionnaire contained three questions, one for each assignment where Progvis could be used. Each question consisted of two parts. The first part asked if the student used Progvis to solve that lab. If the student answered yes, they were also asked to assess how helpful they found Progvis in helping them understand the assignment on a 4-point Likert scale. At the end of the questionnaire the students were also asked for general feedback about Progvis.

To investigate whether using Progvis aids students' learning of concurrent programming we collected student answers from the final exam from different years, and compared students' performance in 2022 to previous years. Since all exams are required to be published in Sweden it was not possible to use the same question for multiple years. However, the structure of the exam has remained the same since 2018, again with the exception of years 2020 and 2021 due to the remote teaching mandated by the pandemic. To compare students' performance on exams with a similar structure, we include the years 2018, 2019, and 2022 in our comparison. The two-year gap between 2019 and 2022 means that any differences might be due to other changes in prior courses. We therefore also include year 2021 in the comparison in spite of the remote exam. We do not, however, include year 2020, as the swift transition to remote teaching gave students little time to prepare for the new structure of the exam. All exams were conducted digitally, either in computer labs on campus, or remotely. Students were able to compile and test the code in the exam, but points were not deducted for solutions that did not compile. It was not possible to use Progvis during the exam.

Exams in years 2018, 2019 and 2022 all consisted of two theoretical questions and a number of practical questions. The practical questions involve fixing concurrency errors in a piece of code. At least one of these questions scaffold the process of finding and fixing any concurrency issues in the code similarly to our previous work [24]. The questions first provide an example of how the code may behave incorrectly and ask students to explain the reason for this behavior. After that, students are asked to highlight critical sections in the code, and then to use synchronization primitives to eliminate the concurrency issues. The remaining practical questions involve atomic operations and any topics that are not covered by the larger, scaffolded question(s). For this paper, we select practical questions that 1) scaffold the process of finding errors, and 2) involve some kind of data structure that needs synchronization. Requirement 2 is to allow analyzing the placement of the synchronization primitives similarly to the pilot study of Progvis [26].

The remote exam in 2021 consisted of two parts. The first part consisted of a single scaffolded, practical question as described above. The second part then gave students two possible solutions to the concurrency issues in part 1. One of them was correct and the other was faulty, but students were only told that they had different strengths and weaknesses. Students were then asked to pick one of the solutions and write a small essay outlining any problems in the selected solution. They were also asked to describe any changes needed to their submitted solution in order for it to be correct.

All collected answers were analyzed using the same process as in the pilot study of Progvis [26]: we examined whether or not each answer was correct (i.e., if the solution follows the specification, and has no data races), and whether the used synchronization primitives were declared at an appropriate location. In the questions selected for analysis all synchronization primitives would ideally be declared inside the available data structure (similarly to tasks 2–4 in [26]). For the answers to the question from 2021, we included any modifications described in part 2 when considering the correctness of the solution. Finally, we compared the number of correct answers and correctly declared synchronization primitives between 2022 and the prior years. Since these numbers represent a number of answers, they follow a Bernoulli distribution and we therefore use Boschloo's exact test [5] to compare them. To verify the assumption that the measurements follow a Bernoulli distribution, we compared the Boschloo tests to non-parametric Mann-Whitney U-tests [21], which do not make assumptions about the distribution of the data.

6 RESULTS

This section presents the results from integrating Progvis into the course described in Section 5.1. First we present the results from the questionnaire regarding students' impressions of using Progvis. Then we present the results from the analysis of the answers to the final exams in the different years.

6.1 The Questionnaire: Students' Impressions

Forty percent (31 of 77 students) responded to the questionnaire. The responses are presented in Fig. 5 and show that approximately two thirds of the respondents used Progvis for the two assignments which encouraged using Progvis, and only one third used Progvis for the *reader-writer lock* assignment where Progvis was only presented as an option. All students who used Progvis answered that they found it helpful to understand the assignment to at least some extent, and a majority answered either *fairly much* (option 3) or *very much* (option 4).

Seventeen students also provided additional, written feedback about Progvis. Even though the question were worded neutrally (i.e., do you have any additional feedback?) all but two answers were positive towards Progvis. The first non-positive answer stated that they were unsure how to use it for the labs as they often managed to solve the synchronization issues, but got stuck on other parts of the assignment. The second stated that they were unsure if they had the time to learn Progvis since it was not introduced in detail during lectures. Additionally, one of the students that were positive to Progvis stated that they used it to visualize some examples, but then opted to not use it during the lab assignments to prepare themselves for situations where visualization is impractical. Finally, one student explicitly highlighted the usefulness of the model checker.

Some answers also highlighted some minor issues that could be addressed. One student noted compatibility problems on some Linux platforms. Another student highlighted not being able to include multiple files, and thereby visualizing larger programs as an issue. Furthermore, one student noted that they had problems in creating their own examples to visualize, likely due to Progvis only supporting a subset of the C language. Finally, one student highlighted a potential improvement to the *introduction to synchronization* assignment to make it better illustrate some issues. We will discuss these comments further in Section 7.2.

6.2 Students' Performance

Using the criteria from Section 5.2, we selected a total of 7 questions (2 from 2022, and 5 from prior years) to collect data from. We use the notation Q<*year*>-<*number*> to refer to these questions (e.g., Q2022-1 is the first question from 2022). While we are unable to

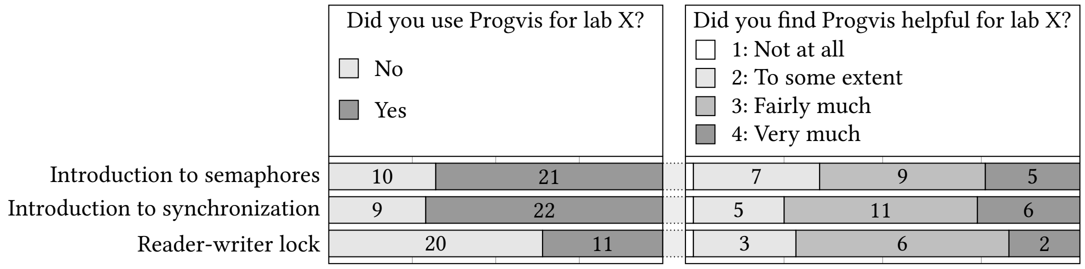


Figure 5: Summary of students who used Progvis to solve the new assignments (left), and to what extent these students found Progvis helpful in aiding their understanding of the assignment (right).

reproduce the questions in full due to space constraints, we provide a brief summary of the questions below, and include them in full in the supplementary materials for this paper.

Q2022-1 The data structure in this question represents a train track. Each track is divided into segments, and only one train is allowed to be present in each segment. The data structure also contains a function to output a visual representation of the track to standard output. The question asks to synchronize access to the segments, and to ensure that the print function outputs a consistent state of the data structure.

Q2022-2 The program in this question rasterizes a line-drawing into a bitmap. Multiple threads are used to speed up the rasterization process. A function for rasterizing a single scanline is provided. Since scanlines are independent of each other (which is explicitly stated) the student only needs to synchronize a shared data structure to ensure that each scanline is rasterized once, and that they are outputted to standard output in the correct order.

Q2021-1 The data structure in this question manages a pool of threads that handle requests in a hypothetical web-server. When the data structure is created, a specified number of threads are created. Work submitted to the data structure is then handled by one of these threads. When the data structure is destroyed, all threads shall be terminated. The question asks the student to synchronize all these operations.

Q2019-1 The program in this question simulates two hypothetical robots that mix drinks to customers. The program contains a data structure that represents a shared inventory of ingredients, and the question asks students to ensure that access to the inventory is properly synchronized.

Q2019-2 The data structure in this question is a *future*-object that allows threads to wait for some value that will become available at some point in the future. The question is similar to the fourth task from [26], but scaffolds the process of finding problems in further detail.

Q2019-3 The program in this question computes the value of a function $f(x)$ for all values $0 < x < n$ and stores them in an array. As in Q2022-2, this involves ensuring that each value is computed exactly once using a shared data structure. The order in which the elements are stored is, however, not important as they are stored in an array.

Q2018-1 The data structure in this question stores a number of strings. It allows sequential insertion and random removal of elements. The insertion fails if the data structure is full, while removal shall wait for an element to be added. The question asks students to synchronize these two operations.

Table 1 contains the results of analyzing the answers to these seven questions for correctness (i.e., if they work according to the specification and are free from data races). Some answers to Q2022-1 stated that they did not consider the print function to be a part of the question and therefore did not synchronize it, even though one of the examples explicitly highlighted a problem with the output. Since this interpretation might be widespread, the column marked with a dagger (†) ignores the print function entirely when assessing correctness.

As can be seen in Table 1, the number of correct answers varies between 10% (Q2022-1) and 69% (Q2019-1). This is true even for questions from the same exam. For example, 69% of students answered correctly to Q2019-1 while only 26% answered correctly to Q2019-2. The Boschloo tests show a number of differences between 2022 and prior years, but all of these are cases where students in 2022 performed worse compared to prior years. The exceptions to this result is the alternate correctness criteria for Q2022-1. All differences that the Boschloo test found significant, the Mann-Whitney U-test also found significant and vice versa.

Table 2 presents the number of solutions that incorrectly declared the synchronization primitives globally. The table shows that fewer students declared the synchronization primitives globally in 2022 (3% and 4% respectively) compared to all prior years. Furthermore, the Boschloo tests show that the results from 2022 are significantly lower than all prior questions except Q2019-2. As with the tests for the correctness, the cases found to be significant using the Boschloo test were also significant using the Mann-Whitney U-test and vice versa.

7 DISCUSSION

In this section we discuss the improvements to Progvis and the results in order to answer the research questions. Each of the three subsections below aims to answer RQ1–RQ3 respectively.

Table 1: Comparison of the number of correct answers between year 2022 and prior years. The question marked with a dagger excludes issues with the `print` function. Significant results are marked with an asterisk.

		Question	Q2022-1	Q2022-1†	Q2022-2
		n	67	67	67
Question	n	Correct	10%	63%	28%
Q2021-1	68	22%	$p = 0.0852$	$p < 0.0001^*$	$p = 0.4298$
Q2019-1	68	69%	$p < 0.0001^*$	$p = 0.4677$	$p < 0.0001^*$
Q2019-2	68	26%	$p = 0.0187^*$	$p < 0.0001^*$	$p = 0.8760$
Q2019-3	57	37%	$p = 0.0006^*$	$p = 0.0045^*$	$p = 0.3273$
Q2018-1	67	55%	$p < 0.0001^*$	$p = 0.4104$	$p = 0.0017^*$

Table 2: Comparison of the number of students who incorrectly placed synchronization primitives at a global scope between year 2022 and prior years. Significant results are marked with an asterisk.

		Question	Q2022-1	Q2022-2
		n	67	67
Question	n	Global	3%	4%
Q2021-1	68	22%	$p = 0.0010^*$	$p = 0.0033^*$
Q2019-1	68	16%	$p = 0.0117^*$	$p = 0.0386^*$
Q2019-2	68	9%	$p = 0.2293$	$p = 0.4149$
Q2019-3	57	26%	$p = 0.0002^*$	$p = 0.0007^*$
Q2018-1	67	19%	$p = 0.0027^*$	$p = 0.0093^*$

7.1 Improvements to Progvis

As described in Section 4 we implemented two improvements to Progvis with the aim of further improving its accuracy and usefulness. The first improvement improves the accuracy of the memory model by implementing a weaker memory model that allows Progvis to detect data races (Section 4.1). This is important as students might otherwise be led to believe that sequential consistency holds in general, which previous work [25] found evidence for. The proposed memory model allows modelling the behavior of common synchronization primitives and sequentially consistent atomics, while remaining simple to explain to students and simple to visualize in Progvis. It is, however, a simplification of the C memory model and it is therefore not able to model more advanced concepts such as relaxed atomics. We believe this is a good trade-off as Progvis is aimed at introductory courses on concurrency.

The second improvement was to implement a model checker (Section 4.2) to help students find errors in their programs. As it also provides an example of an interleaving that causes the error it lets students validate their understanding of the execution model, as mentioned by Lönnberg et al. [20]. For example, a student who believes that *acquiring a lock prevents all other threads from executing* [25] will likely not explore interleavings that break this assumption. In such cases, the model checker can help by providing an example that illustrates that the assumption is incorrect. While the model checker is useful, it is not a silver bullet. Similarly to unit testing, it needs a test program that suitably exercises the concurrent behavior of the code to test, or it might fail to find some errors. Furthermore, the model checker only assesses that the program works as intended, not the quality of the solution. It is therefore not a replacement for manual grading and feedback. Finally, it is worth mentioning that the model checker is only able to verify relatively small programs, otherwise the number of states become large enough for verification to take a substantial amount of time.

With the addition of these features, Progvis matches or exceeds the functionality of the existing tools mentioned in Section 3.2. In addition to providing an interactive visualization of concurrent execution like ConEE [22] and The Deadlock Empire, it also visualizes a rich data model. The improved memory model adds further accuracy to the visualization by detecting and reporting data races. To our knowledge, Progvis is the only tool that models this kind of undefined behavior. Furthermore, the addition of the model checker means that Progvis, similarly to ConEE, Eraser [6] and the Spin model checker [3], is able to automatically detect errors, and provide visuals that help students understand the cause of the error, similarly to Atropos [17].

7.2 Students' Impressions

Since the number of respondents to the questionnaire were fairly low (40%) the results should be interpreted with some care. This is especially true since respondents were self-selected, which means that the results might be biased towards students who were positive to Progvis. The answers to the questionnaire showed that approximately two thirds of students used progvis when encouraged to do so, and that all of these students found it to help their understanding of the assignments. For the last assignment, *reader-writer lock*, where Progvis could be used but was optional, only about one third of students used Progvis. Since this assignment appears late in the course it could mean that students were already comfortable enough with the material to solve it on their own. Another possibility is that, similar to one of the free-form answers, students were stressed towards the end of the course and did not want to spend time on anything but writing code that directly contributed to solving the assignment (even though Progvis might have saved time).

While the feedback from the students were generally positive, they highlight some areas that could be further improved. In particular, the worries about Progvis taking much time to learn could be addressed by slightly expanding the many visualizations in Progvis already performed during lectures to also show the simplicity of loading programs, and a slightly more thorough explanation of the model visualized by Progvis. This concern, alongside difficulties with creating examples and visualizing programs in multiple files, could also be addressed by providing better and easily accessible documentation. For example, it is possible to visualize programs spanning multiple files, but it is not apparent how to do so since include statements are currently ignored.

7.3 Students' Performance

Comparing the number of correct solutions between 2022 and prior years showed some significant differences, mainly that students 2022 performed worse than prior years. However, the large variation in the number of correct answers, even within the same exam, suggests that the questions were not equally difficult. For example, 69% of answers to Q2019-1 were correct, while only 26% of answers to Q2019-2 were correct. This can also be seen for 2022, where 63% of students solved a large part of Q2022-1 correctly, even though only 10% managed to solve the issue with the print function. Another cause of this variation are differences in prior courses. This is particularly relevant here, as students in 2022 took their introductory courses remotely during their pandemic, which likely has some effect (either positive or negative) on their learning. To minimize this difference, it is most relevant to compare years 2021 and 2022, which is made difficult due to the remote exam in 2021. Because of all these factors we find these results inconclusive.

In spite of the above-mentioned problems, the results show that students were significantly more likely to declare their synchronization primitives correctly (inside the data structure) in 2022 compared to prior years, with the exception of Q2019-2. Since more students declared their synchronization primitives globally in Q2019-1 and Q2019-3, this could be because the test program to Q2019-2 creates and uses multiple instances of the future object. This is only done in one other question, Q2022-1. As such, this could mean that students realize the problem with global synchronization primitives more often when an example is provided. However, this benefit seems to not be significant in 2022, since the difference between Q2022-1 and Q2022-2 was not significant. Finally, the improvement in 2022 could also be due to a modification in a prior course. We do, however, believe this to be unlikely since the difference between 2022 and 2021 was large (22% vs. 3% and 4%), and the difference was significant for all prior years (except Q2019-2).

As such, the results are a strong indication that Progvis does help students learn some aspect of concurrent programming. Since more students succeeded in correctly associating synchronization primitives with the protected data, their mental models of either the synchronization primitives, or the semantics of the programming language have improved. Both of these aspects were highlighted as important by Strömbäck et al. [25]. Furthermore, Fisler et al. [8] have shown that students struggle with concepts such as scope, mutation and aliasing, which are vital when working with concurrency. As such, these results could mean that students have improved their understanding of these particular concepts by using Progvis. Finally, Haglund et al. [9] argue that this type of concepts are important in other areas, such as abstraction. This means that Progvis might be useful in other contexts as well.

8 CONCLUSION

In this paper, we have addressed the shortcomings in Progvis we highlighted in our previous work [26] by implementing a memory model that more closely resembles the intricacies of the C memory model, and covers the concepts that are typically covered in introductory courses on concurrency. While the proposed model is not able to encompass all intricacies of the C memory model, it has the benefit of being easy to explain to students, and easy to visualize. We also implemented a model checker into Progvis, which allows it to automatically find concurrency errors in the visualized program that students might otherwise have missed, and provide an example to illustrate why the error occurs. As with many other automatic tools, this requires adequate tests to produce reliable results.

This paper also evaluates Progvis in a larger scale compared to [26]. The results from this evaluation suggest that approximately two thirds of students used Progvis when encouraged to do so, and that most of these students found that it helped them understand the assignment. The results regarding students' ability to correctly solve concurrency problems were inconclusive due to large variations in student performance. The results did, however, show that significantly fewer students incorrectly declared synchronization primitives in a global scope in the year 2022, which suggests that using Progvis helps students at least in this regard.

REFERENCES

[1] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking.* MIT Press, Cambridge, Massachusetts.

[2] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

[3] 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

[4] 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

[5] R. D. Boschloo. 1970. Raised conditional level of significance for the 2×2-table when testing the equality of two probabilities. *Statistica Neerlandica* 24, 1 (1970), 1–9.

[6] 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

[7] 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

[8] 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

[9] Pontus Haglund, Filip Strömbäck, and Linda Mannila. 2021. Understanding Students' Failure to use Functions as a Tool for Abstraction – An Analysis of Questionnaire Responses and Lab Assignments in a CS1 Python Course. *Informatics in Education* 20, 4 (2021), 583–614. https://doi.org/10.15388/infedu.2021.26

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

[11] 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.

[12] 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

[13] Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[14] 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

[15] 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

[16] 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

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

[18] Jan Lönnberg and Anders Berglund. 2007. Students' Understandings of Concurrent Programming. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88* (Koli National Park, Finland) *(Koli Calling '07)*. Australian Computer Society, Inc., Darlinghurst, Australia, 77–86. http://dl.acm.org/citation.cfm?id=2449323.2449332

[19] 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

[20] Jan Lönnberg, Lauri Malmi, and Anders Berglund. 2008. Helping Students Debug Concurrent Programs. In *Proceedings of the 8th International Conference on Computing Education Research* (Koli, Finland) *(Koli '08)*. ACM, New York, NY, USA, 76–79. https://doi.org/10.1145/1595356.1595369

[21] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50 – 60. https://doi.org/10.1214/aoms/1177730491

[22] 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

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

[24] 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

[25] 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

[26] 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