Automating Testing of Visual Observed Concurrency

Prasun Dewan
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599
dewan@cs.unc.edu

Andrew Wortas
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599
ajwortas@live.unc.edu

Zhizhou Liu
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599
yiwk321@cs.unc.edu

Samuel George
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599
sdgeorge@cs.unc.edu

Bowen Gu
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599
gubowen2@live.unc.edu

Hao Wang
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599
harrywh@live.unc.edu

***Abstract*—. Existing techniques for automating the testing of sequential programming assignments are fundamentally at odds with concurrent programming as they are oblivious to the algorithm used to implement the assignments. We have developed a framework that addresses this limitation for those object-based concurrent assignments whose user-interface (a) is implemented using the observer pattern and (b) makes apparent whether concurrency requirements are met. It has two components. The first component reduces the number of steps a human grader needs to take to interact with and score the user-interfaces of the submitted programs. The second component completely automates assessment by observing the events sent by the student-implemented observable objects. Both components are used to score the final submission and log interaction. The second component is also used to provide feedback during assignment implementation. Our experience shows that the framework is used extensively by students, leads to more partial credit, reduces grading time, and gives statistics about incremental student progress.**

Keywords—software testing, concurrency, education, producer-consumer, observer, simulations, synchronization, event database

I. Introduction

Program assessment is the problem of generating feedback for programs expected to meet the requirements of some assignment given in a formal or informal learning environment. In both environments, it can help trainees calibrate their performance, receive feedback, and converge to a final solution. In formal learning environments, it also scores the final solution. Therefore, it is attractive to explore automation of program evaluation to not only reduce instructor grading burden but to also create a collaborating "agent" for students and gather statistics about incremental student progress. Program evaluation is particularly important for concurrent programs, as they are notoriously difficult to write [1-3] and substantial instructor effort is required to evaluate the performance and correctness of these programs and identify potential problems.

Evaluation of student programs can be partially automated by grading management systems and completely automated by grading automation systems. Grading management systems require a human evaluator to inspect different components of each submission. They automate the effort required to (a) navigate among submissions of an assignment provided by different students, and (b) score a submission associated with a rubric. Grading automation systems automatically score the submissions.

Contemporary examples of these two kinds of systems are designed to evaluate the sequential but not concurrent aspects of assignments. The intuitive reason is that the use of concurrent abstractions affects the algorithm implemented by the program, and current assessment systems are oblivious to the algorithms used by the evaluated programs.

We have developed a new testing-based framework that makes assumptions about the domain to provide both a grading management and automation system for evaluating the concurrency requirements of assignments implemented in Java. We assume that the user-interface of a concurrent program (a) is implemented using the observer pattern and (b) makes apparent whether the concurrency requirements are met. We have used this framework in an undergraduate course on object-based and concurrent programming, offered in the summer of 2021, and recorded and analyzed various aspects of its use. We refer to this course as our target course.

The rest of the paper is organized as follows. Section II surveys previous work in this area. Section III describes the domain on which we focus using illustrative problems. Section IV and V describe and illustrate the components of our framework. Section VI gives our experience using various components of the framework. Section VII gives conclusions and directions for future work.

A recording of the conference talk on this paper is available here.

This work was funded in part by NSF awards OAC 1829752 and 1924059.

II. RELATED WORK

A. Gradescope Grading Management

Assignment evaluation is more than solution checking – it consists also of efficiently navigating among solutions (ideally with one click), remembering the requirements for an assignment, assigning scores based on which requirements are met, and possibly reassigning the scores based on new information. A grading management system automates these tasks without automating any solution checking. Gradescope has popularized the idea of a web-based implementation of such a system [4]. An instructor interactively associates components of a fixed-size (in terms of pages) assignment pdf with rubric templates - tables mapping requirements to scores - which are instantiated for each submission. Human graders can display each component with the associated rubric instance and simply click on the requirement to identify that it has been met. The tedious and error-prone task of (re)assigning and totaling scores in a rubric instance and displaying it to the students is automated.

Gradescope has recently added the ability to inspect a variable-sized collection of source code and other scrollable and collapsible files, and associate this collection with a single rubric. Through these capabilities it provides support for general assignments, making few assumptions about their nature.

To interactively run programs submitted to Gradescope, instructors must use an ssh interface, which provides no grading management and does not directly support creation of graphical user-interfaces. Thus, the system provides no special support for manually testing visualized concurrency. Our grading management system addresses this limitation.

B. Problem-Independent and Specification-based Testing

Testing of industrial-strength parallel programs has received much attention because of the difficulty of coding them correctly and efficiently. A comprehensive review of this research [3, 5-8] is provided in [1]. It can be divided into (a) techniques for testing that programs meet problem-specific declarative formal specifications [7, 9], and (b) problem-independent tools that help identify race conditions, deadlocks, and other problem-independent issues in multi-threaded programs [10-13]. Meeting problem-specific constraints implicitly ensures problem-independent issues (preventing these constraints from being met) do not occur. What is missing in this work are testing frameworks that evaluate problem-specific constraints without requiring formal declarative specifications of the behavior of programs. It is because of these limitations that we believe they have not been used, to the best of our knowledge, in educational environments

C. JUnit

JUnit is a popular Java-based framework for writing tests for student programs. Instructor or students write test methods and delineate them from tested methods through special annotations. JUnit provides an API to automatically find and execute a sequence of test methods sequentially. Some programming environments such as Eclipse provide a special user interface for triggering the execution of such a sequence and displaying the results and execution times. JUnit tests a single implementation of the tested code. Some educational systems such as Web-CAT [14, 15], ASSYST [16] and Gradescope automatically execute the same JUnit test code on each submission of an assignment.

JUnit is designed to provide functional testing of programs in which the relationships between input and output parameters are tested. Concurrency has to do with the algorithm used by the methods – the same functionality can be implemented by a sequential and parallel program. For example, a method can add a sequence of numbers either sequentially or in parallel – functional testing cannot tell the difference. Our concurrency extensions overcome this limitation by providing special support to JUnit tests to determine the nature of the algorithm used.

D. Concurrency Testing in the Classroom

We know of two uses of automatic tests for assessing concurrent student programs. In an invited talk in Edu-HPC-18, Lumsdaine [17] mentioned the implementation and use of automated tests for a concurrency course he taught at UW-Seattle, but also indicated that they were difficult to write and had errors, leading to student frustration. He did not describe the nature of these tests.

Sarkar's group [18] reported the use of Web-CAT to write and execute tests for a concurrency-based CS course at Rice University that uses a variant of Java developed at Rice – Habanero Java. To evaluate concurrency aspects, they looked at the speedup resulting from running their tests with an increasing number of cores. In addition, they developed a web UI for displaying the performances of different student submissions. In a later paper [19], the group reported that they abandoned Web-CAT for several reasons. They also reported problems using alternate tools such as Marmoset [20]. Therefore, they developed a web-based tool specific to their course and Habanero Java to handle the entire assessment process. Their custom grader was used extensively by students. A later project used it to assess assignments in an online version of their course [21].

Speedup is insufficient to assess concurrent problems in which threads do dissimilar tasks (such as producing and consuming information in a pipeline) or when increasing the number of cores allocated to a program is not possible (because a language such as Java does not provide a way to do so or because a student's computer does not have enough cores), or when possible, does not lead to perceptible performance improvement. Our work is targeted at a subset of these problems. If the program provides an entry point to do so, a test can change the number of threads, as illustrated in our dining philosopher implementations, and observe the variation in performance that results.

III. VISUAL OBSERVABLE CONCURRENCY

A fundamental assumption in our work is that the user-interface of the tested concurrent assignment provides information to evaluate its concurrency constraints. For example, an add program can print partial sums and the names of the threads that compute them to demonstrate its concurrent execution. In [22, 23], we describe a class of problems in which the user interface does not require special modifications to provide such information. These problems implement simulations of concurrent actions in the real world. The Olympics provide several such examples. Consider the

simulation of a relay race. It would use atomicity primitives to ensure that competitors do not share the same track. It would use coordination primitives to block and unblock (a) a single thread to simulate baton passing, and (b) multiple threads to simulate racers waiting for a whistle. Synchronized swimming would require continuous coordination. Correct and incorrect implementations of these simulations can be identified by running them one or more times using carefully chosen input.

In [22, 23], we describe several such simulations, implemented both as worked examples and student assignments. For example, one worked example involves two pilots, represented by two different threads, taking a shuttle along two different planned paths. In the correct implementation, the later pilot waits for the previous pilot to complete the planned task, while in the incorrect one, this atomicity constraint is violated, leading to the shuttle oscillating between the two planned paths.

A more well-known example is provided by the user-interface of our textual simulation of a variation of the classic Dining Philosopher problem, shown in Fig. 1. This simulation involves a user-specified number of philosophers using chopsticks placed between them - one chopstick between each

```
Please enter the number of philosophers/chopsticks
2
Hit return to quit or input time to eat the next course:
200
Hit return to quit or input time to eat the next course:
Philosopher 0.Fed:false->false
Philosopher 1.Fed:false->false
Chopstick 0.Used:false->true
Chopstick 1.Used:false->true
Philosopher 0.WithLeftChopstick:false->true
Philosopher 1.WithLeftChopstick:false->true
Chopstick 0.Used:true->false
Philosopher 0.WithLeftChopstick:true->false
Chopstick 0.Used:false->true
Philosopher 1.WithRightChopstick:false->true
Chopstick 1.Used:true->false
Philosopher 1.WithLeftChopstick:true->false
Chopstick 0.Used:true->false
Philosopher 1.WithRightChopstick:true->false
Philosopher 1.Fed:false->true
Chopstick 0.Used:false->true
Philosopher 0.WithLeftChopstick:false->true
Chopstick 1.Used:false->true
Philosopher 0.WithRightChopstick:false->true
Chopstick 0.Used:true->false
Philosopher 0.WithLeftChopstick:true->false
Chopstick 1.Used:true->false
Philosopher 0.WithRightChopstick:true->false
Philosopher 0.Fed:false->true
```

Fig. 1. Similation of Polling Multi-Course Dining Philosophers Visualizing Concurrency and Coordination

pair - to eat multiple courses. They think for different times before trying to pick the chopsticks, but once they have successfully picked both chopsticks, they take a fixed amount of time (input by the user) to eat the course, and then return the chopsticks to their resting places.

The user-interface shows transitions to the properties of the objects simulating the philosopher and chopsticks. A chopstick object has one Boolean property, *Used*, which indicates whether it has been picked up by one of the two adjacent philosophers or not. A philosopher object has three properties, *Fed*, indicating whether the philosopher has finished the current course or not, and *WithLeftChopstick* and *WithRightChopstick*, indicating whether the philosopher has successfully picked the left and right chopstick, respectively.

In this implementation, the actions of each philosopher are executed by a separate thread. Two courses, taking the same time, lead to different sequences of events, with the two philosophers picking chopsticks and finishing eating in different orders, which is an indication that the solution is indeed multi-threaded. Running the program multiple times provides further, but not, of course, conclusive, evidence of the concurrency.

In the implementation illustrated, if an attempt to pick a chopstick fails because it is in use, the philosopher thread that executes the action does not wait in a queue. Instead, it releases the other chopstick, if it has it, waits for some time, and then tries again, repeating these steps until success is achieved. Thus, this is a polling solution to the problem that does not lead to deadlocks. This is indicated by the actions of Philosopher 0, who released the first chopstick picked because the other chopstick was in use, and then picked the two chopsticks later when they were both available at the same time.

The main thread repeatedly prompts the user for the next course time. This thread does not coordinate with the philosopher threads to determine if they have successfully set the *Fed* properties of their respective *Philosopher* objects to true. This thread essentially represents a Butler serving food. The lack of coordination is exhibited also in the user-interface, as the prompt for the second course is given immediately after the time for the first course has been input rather than after the two philosophers have been successfully fed the current course. A graphical user-interface of the kind supported by our assignments would make these inferences more evident.

What is inferred by close human inspection of the user-interface should also be evident to test code if it is given the same information. In this example, the user-interface is textual, so the output of the application could be provided to this code. However, this approach requires the test code to parse the text, and more important, does not work for graphical simulations. In our target course, all assignments were graphical, and together simulated a variation of the bridge scene from the movie *Monty Python and the Holy Grail*. In one extension to this scene, the avatars representing the Knights marched to a beat set by the clapping of a guard (Fig. 2), which corresponds to our previous example of synchronized swimming. Whether the coordination among the threads animating the Knights and the guard was correct was evident from the user-interface. However, there was no textual representation of the actions to be given to tests.

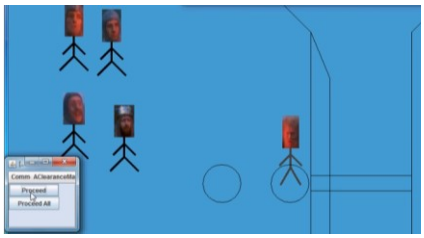

Fig. 2. Holy Grail Bridge Scene:

To reduce parsing overhead and accommodate graphical simulations, we make another assumption about the nature of tested programs, which is consistent with the fact that our target course was on object-oriented programming. We assume that all simulated objects are

observable [24]. An observable object allows arbitrary observers to be registered with it, and on every write to its state, announces the nature of the write to each registered observer by calling a notification method in it. We will refer to the state change as a notification or an event. This is a pattern familiar to anyone who has been explicitly taught it or has been exposed to user-interface toolkits, which allow, for instance, application objects to be notified of clicks on observable buttons.

The assumptions about visualization of concurrency and observability of state changes are the basis, respectively, for the two components of our framework – the grading management system and the automatic testing system.

IV. Active Grading Management System

While our grading management system was developed independently of Gradescope, it offers similar features to automate rubric management and navigate among student submissions. Fig. 3 shows parts of it being used to manually grade the concurrency features (4-7) of an assignment, which also included non-concurrency features (1-3).

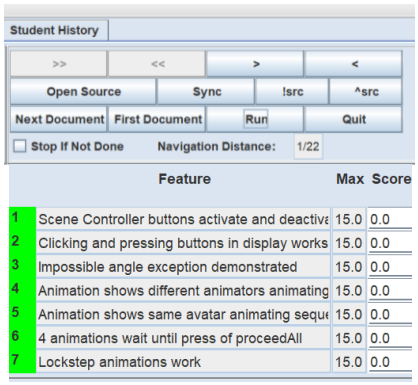

Fig. 3. Interactive Submisson Exection

Unlike Gradescope, our system is specialized for program grading and is not web-based. This allows us to make it "active" by providing a *Run* command, which can be invoked by a human grader to execute the graded submission and interact with it. Fig. 2 shows the result of invoking this command from the user interface of Fig. 3. This command is, of course, crucial to grade visualized concurrency.

Sometimes the human grader needs to run the code through the debugger, modify it, and upload it back to explain the problem to the student. The *!src* command copies the code from the student submission directory to a specified directory – typically the *src* directory of a previously configured Eclipse project – and the *^src* command writes the code back to the submission directory. In both cases, previous contents of the target directory are replaced. The system assumes the directory structure of the Sakai Learning Management System (used in our university), which allows the student directory to be uploaded back to Sakai (as feedback) after downloading it.

V. Programming Observing Test Cases

As mentioned earlier, if concurrency is visualized and all visible state changes are observable, then it should be possible to write test cases that automatically check consistency constraints. We refer to such tests as observing concurrency test cases. Below, we derive a thread architecture and event-based generic abstractions for reducing the complexity of such test cases and the effort required to write them. We assume JUnit test cases, though the concepts are more general.

A. Event-Database and Upper and Lower Halves of Tests

The writer of the test code must not only address the nature of multithreading in the application code, but also deal with multithreading issues in the test code, because the test observer code is expected to be called by multiple application threads. As mentioned earlier, concurrent code is difficult to write [1-3].

Therefore, in the concurrent test cases implemented by us, we have followed a common thread-architecture that provides isolation between the parts of the test code executed by the testing thread and the ones invoked by application threads. It is based on an event database. Fig. 4 shows its nature.

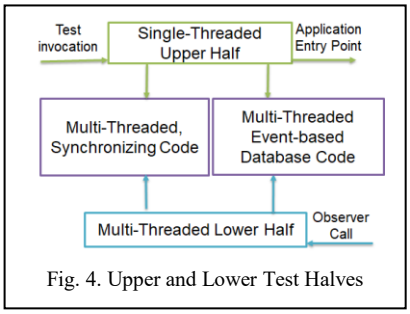

Fig. 4. Upper and Lower Test Halves

The test code consists of two halves, the upper and lower halves, which are executed by the testing thread (e.g. one testing if the dining philosopher solution is deadlock-free) and application threads (such as philosopher threads), respectively. The lower-half consists of the code called by application observables (such as the objects simulating the philosophers and chopsticks) and stores information about the notified events (such as a chopstick being picked or a philosopher being fed) in an event database. When test-specific synchronizing conditions are satisfied (such as all philosophers being fed), it unblocks the upper-half code waiting for these conditions, which then reads the information in the event database. The event database and synchronizing code, together, form a bridge between the two halves, and can be accessed by both halves and the corresponding threads.

A test implemented using this architecture resembles a traditional producer-consumer implementation with the test thread executing the upper-half code being the consumer and the application threads invoking the lower-half code being the producers. The difference is that consumers are unblocked, not when a buffer becomes non-empty, but when application-specific conditions are met such as an event indicating that all philosophers have been fed in the current round.

B. Lower-Half Event Database Management System

Once we gained experience with this thread architecture, our next step, to reduce test complexity, and more important, effort, was to develop a "test-independent" generic implementation of the lower half and bridge code. Such a step would be akin to moving from application-specific databases to "generic" database management systems (DBMS) such as a relational DMBS. Therefore, we refer to this abstraction as the event database management system. The terms "test-independent" and "generic" are in quotes as automation is provided by making assumptions, which reduces flexibility.

As a first-step effort in this direction, our goal was to create an event DBMS that satisfied the requirements of our assignments and multiple versions of the dining philosopher

problem. The fact that the dining philosopher problem is a classic is a strong indication that it is representative of a large class of concurrency problems.

The DBMS assumes all notifications are instances of the standard Java event, a property change. Such an event has four components – the (a) source, which is the observable object that was written, (b) property, which is the name of the property of the observable that was written, (c) old value, which is the value of the property before the announced write was made, and the (d) new value, which is the current value of the property. The last four fields in Fig. 5 give examples of these four components.

When property events arrive, our DBMS wraps them in concurrent property events, and stores some of these events in an ordered sequence, which can be reset. A concurrent event adds three additional components to the event it wraps – the (a) event relative time, which is the difference in times of event announcement and DBMS reset, (b) sequence number, which indicates how many events occurred before it since the last reset,

```
16:[0,Thread Philosopher 1,Philosopher 1,Fed,false,false]
17:[1,Thread Philosopher 0,Philosopher 0,Fed,false,false]
228:[2,Thread Philosopher 0,Chopstick 0,Used,false,true]
228:[3,Thread Philosopher 1,Chopstick 1,Used,false,true]
228:[4,Thread Philosopher 0,Philosopher 0,WithLeftChopstick,false,true]
228:[5,Thread Philosopher 1,Philosopher 1,WithLeftChopstick,false,true]
228:[6,Thread Philosopher 0,Chopstick 0,Used,true,false]
228:[7,Thread Philosopher 0,Philosopher 0,WithLeftChopstick,true,false]
229:[8,Thread Philosopher 1,Chopstick 0,Used,false,true]

                                ...
851:[21,Thread Philosopher 0,Chopstick 1,Used,true,false]
851:[22,Thread Philosopher 0,Philosopher 0,WithRightChopstick,true,false]
851:[23,Thread Philosopher 0,Philosopher 0,Fed,false,true]
```

Fig. 5. Concurrent Property Change Objects Stored in Event DBMS

and the (c) thread that announced the event. The first three fields in Fig. 5 give examples of these components, respectively. Thus, the third entry in the figure corresponds to an event announced by Thread *Philosopher 0* at relative time *228*, with sequence number *2,* which announces a write of the value true to property *Used* of observable *Chopstick 0* whose old value was false.

Application-specific processing is performed by executing application-specific implementations of the Selector interface: shown in Fig. 6. Fig. 7 shows the DBMS interface. It is an extension of the standard Java interface, *PropertyChangeListener*, so that it can listen to property changes. It allows registering of two kinds of selectors: ignore and wait selectors. Each registered ignore and wait selector is invoked each time an event is received. The event is not stored if the select method of any of the ignore selectors returns true. An upper half thread can execute *selectorBasedWait()* to block until the specified timeout or the select method of any of the wait selectors returns true. The DBMS also provides the upper half with operations to determine all threads that have been notified since the last reset and those that were created after the reset.

```
public interface Selector<SelectedType> {
  boolean selects(SelectedType anObject);
}
```

Fig. 6. Selector Interface

```
public interface ConcurrentPropertyChangeSupport
              extends PropertyChangeListener {
 ConcurrentPropertyChange getLastReceivedPropertyChange();
 void resetConcurrentEvents();
 void addIgnoreSelector(
   Selector<ConcurrentPropertyChangeSupport> aSelector);
 void addWaitSelector(
   Selector<ConcurrentPropertyChangeSupport> aSelector);
 void selectorBasedWait (long aTimeOut);
 ConcurrentPropertyChange[] getConcurrentPropertyChanges();
 Thread[] getNotifyingNewThreads();
 Thread[] getNotifyingThreads();
}
```

Fig. 7. Event DBMS Interface

D. Two Generic Selectors

To reduce the overhead of writing selectors, we have written two important predefined parameterized selectors. The first, an event matching selector, assumes its select method is passed a concurrent property change. It returns true if the source, old value, and new values of the event matches a regular expression passed to the constructor of the selector, whose header is shown below

```
ParameterizedPropertyChangeSelector(
  Object[] aParameters)
```

Thus, if the constructor has been passed the regular expression array: [".*", ".*", "Chop.*", ".Used, "false", "true"] then events 2, 3, and 8 in Fig. 5 are matched by this selector. Such a selector is used to determine, for instance, if a non-atomic change to the *Used* property of a *Chopstick* results in the old and new announced values being the same, which can happen if a context switch occurs between the time a thread checks the property value and makes and announces the change.

The second selector we provide is a thread-based history matching selector, whose constructor is shown below.

```
ConcurrentPropertyChangeThreadMatchesSelector (
Object[] aParameters, int numThreads, int numMatches,
String threadPattern, long minDelay)
```

Its select method takes an instance of the event DBMS as an argument. The method returns true if at least *numThreads* threads whose names match *threadPattern* have announced events separated by at least *minDelay* that have at least *numMatches* of parameters with the events stored so far in the database passed as an argument. It uses the selector above to determine if an event matches *aParameters*. To illustrate its use, assume a DBMS that stores the events shown in Fig. 5 is passed as an argument to its select method and its constructor parameters are as follows:

numThreads = 2; numMatches = 1;
threadPattern = ".*; minDelay = 0;
aParameters = [".*", "Phil.*: "Fed", "false", "true"]

Then the select method will return true when event 23 is received as two threads have announced a change matching parameters. We use this selector to unblock the upper half of the dining philosopher test when all philosophers have been fed.

We also use this selector in our lockstep animation test code to store events separated by a minimum delay. The reason is that multiple events fired by the same avatar movement give no additional information for testing. What is of interest is not how many events are fired on each beat (guard clap) by an avatar thread but the sequence of events fired by different avatar threads on each beat. Fig. 8 shows a subsequence of events when the minimum delay for each thread is 0. Fig. 9 shows a subsequence of stored events when this delay is 10. As we see

from Fig. 8, a thread fires several events on each beat at the same time. In Fig. 9, only one of these events is stored for each thread.

D. Partitioning, Matching, Interleaving Operations

Our framework also provides three kinds of generic operations on event lists we have found useful in our test cases. These can be invoked during (a) online event processing by the lower-half threads or (b) offline processing by the upper-half threads after they have been unblocked.

```
15518:[0,Thread-3,grail.shapes.RotatingLine@47a41338,Height,-24,-29]
15518:[1,Thread-3,grail.shapes.RotatingLine@47a41338,Width,-16,-5]
15518:[2,Thread-3,grail.shapes.RotatingLine@4c879e1f,Height,-26,-29]
15518:[3,Thread-3,grail.shapes.RotatingLine@4c879e1f,Width,13,1]
```

Fig. 8. Part of Stored Event Sequence with Min Thread Delay 0

```
18222:[0,Thread-3,grail.shapes.RotatingLine@118e6e22,Height,-24,-29]
18479:[1,Thread-3,grail.shapes.RotatingLine@118e6e22,Height,-29,-24]
18479:[2,Thread-1,grail.shapes.ImageShape@477dc525,X,30,50]
18480:[3,Thread-2,grail.shapes.ImageShape@734b6086,X,100,120]
```

Fig. 9. Part of Stored Sequence with Min Thread Delay 10

Sequence partitioning operations split the input sequence by thread, source and time. Given the sequence of events in Fig. 5, Fig. 10, 11, and 12 show one of the splits produced by a partition by thread, source and time, respectively. Splits are useful for not only writing testing code but also understanding the behavior of the concurrent programs by both a student and a human grader. For instance, the Fig. 10 thread split shows that philosopher 0's first attempts to eat occurs at event 0 but is not successful until the last event because the other philosopher was able to get both chopsticks first. The Fig. 11 source split shows that, because of polling, the chopstick is picked three times rather than the optimal number of two. The Fig. 12 time partition shows that philosopher 0 is able to get both chopsticks at this time.

```
17:[1,Thread Philosopher 0,Philosopher 0,Fed,false,false]
228:[4,Thread Philosopher 0,Philosopher 0,WithLeftChopstick,false,true]
228:[7,Thread Philosopher 0,Philosopher 0,WithLeftChopstick,true,false]
634:[16,Thread Philosopher 0,Philosopher 0,WithLeftChopstick,false,true]
634:[18,Thread Philosopher 0,Philosopher 0,WithRightChopstick,false,true]
851:[20,Thread Philosopher 0,Philosopher 0,WithLeftChopstick,true,false]
851:[22,Thread Philosopher 0,Philosopher 0,WithRightChopstick,true,false]
851:[23,Thread Philosopher 0,Philosopher 0,Fed,false,true]
```

Fig. 10. Thread Philosopher 0 Partition

```
228:[2,Thread Philosopher 0,Chopstick 0,Used,false,true]
228:[6,Thread Philosopher 0,Chopstick 0,Used,true,false]
229:[8,Thread Philosopher 1,Chopstick 0,Used,false,true]
432:[12,Thread Philosopher 1,Chopstick 0,Used,true,false]
634:[15,Thread Philosopher 0,Chopstick 0,Used,false,true]
851:[19,Thread Philosopher 0,Chopstick 0,Used,true,false]
```

Fig. 11. Source Chopstick 0 Partition

```
634:[15,Thread Philosopher 0,Chopstick 0,Used,false,true]
634:[16,Thread Philosopher 0,Philosopher 0,WithLeftChopstick,false,true]
634:[17,Thread Philosopher 0,Chopstick 1,Used,false,true]
634:[18,Thread Philosopher 0,Philosopher 0,WithRightChopstick,false,true]
```

Fig. 12 Time 634 Partition

We saw earlier the ability to match a single event with a regular expressions array. A sequence matching operation matches a sequence of, possibly *non-consecutive*, events with a sequence of regular expressions arrays, returning the number of matches that occur. Thus, the regular-expression array sequence of Fig. 13 matches all thread partitions created from Fig. 5 (such as Fig. 10) once. An interleaving operation takes as input N event sequences and determines if they interleave or are executed serially.

E. Anatomy of a Concurrency Test

The test we have implemented for the polling dining philosopher solution illustrates the usefulness of the event DBMS and the predefined operations based on it. It creates an instance of the DBMS and registers it as an observer of all chopsticks and philosopher objects by calling an entry point in the tested code. The test then registers with the DBMS an online selector that returns true when all philosophers have been fed once. It next calls two entry points in the application code that set the number of philosophers and pause time, respectively, and then waits for the selector to return true or a timeout to occur. In the latter case, it gives an error message. Otherwise, it checks the stored event sequence for the following constraints: (1) Did some interleaving occur? (2) Was the number of newly created notifying threads the same as the number of philosophers? (3) Did the expected thread sequence of Fig. 13 occur in each thread partition? (4) Were the old and new values of each notification different? If any of these checks fails, it gives an appropriate error message. Otherwise, it declares success.

```
public static final String[][] EXPECTED_THREAD_CHANGES = {
   {".*", "Philosopher.*", "Fed", ".*", "false"},

   {".*", "Chopstick.*", "Used", "false", "true"},
   {".*", "Philosopher.*", "With.*Chopstick", "false", "true"},
   {".*", "Chopstick.*", "Used", "false", "true"},
   {".*", "Philosopher.*", "With.*Chopstick", "false", "true"},

   {".*", "Chopstick.*", "Used", "true", "false"},
   {".*", "Philosopher.*", "With.*Chopstick", "true", "false"},
   {".*", "Chopstick.*", "Used", "true", "false"},
   {".*", "Philosopher.*", "With.*Chopstick", "true", "false"},

   {".*", "Philosopher.*", "Fed", "false", "true"},

   }
```

Fig. 13. Sequence Matching

VI. EXPERIENCE

A. Programming Tests

The first author implemented the following tests for the concurrency-covering assignment (A4) of our target course: (a) *AsyncAvatar*: Checks that separate threads are created for moving the Knights in the bridge scene (Fig. 2). (b) *SyncAvatar*: Checks that two threads that move the same Knight do so atomically. (c) *WaitingAvatar*: Checks that threads for moving the Knights do not start the animation until the user presses the *ProceedAll* button (Fig. 2). (d) *LockstepAvatar:* Checks that the Knights march to the beat set by the Guard's clapping. Each test was first implemented without the event DBMS abstractions and then (after the course terminated) with these abstractions. In both cases, *AsyncAvatar* and *SyncAvatar* had four subclasses, one for each Knight (e.g. *AsyncArthur* and *AsyncLancelot*), but these were trivial checks not concerned with events and thus not considered here. Both sets of checks were equivalent in that they produced the same result and error messages on the same events, used the same principles to remove code duplication, were formatted similarly, and had the same or equivalent comments.

Table I shows the lines of code written with and without the abstractions. As expected, the abstractions reduced the code sizes as they required writing of only the upper half. *AsyncAvatar* was much larger than the other classes because, in both cases, it was their superclass and implemented common

code inherited by the latter. What is most striking is the small amount of code required to check lockstep movement with abstractions. The reason was that with the abstraction, the test asked the DBMS to store for each thread only events that occurred at different beats, and then asked the event sequence matching operation to determine how many sequences of Guard followed by Knight events occurred.

TABLE I. TEST LINES OF CODE

Test	Without abstraction	With Abstraction
AsyncAvatar	300	195
SyncAvatar	116	47
WaitingAvatar	129	38
LockstepAvatar	198	38

The first author also used the abstractions to implement several different versions of the dining philosopher problem. (i) Sequential: A single thread executes the operations of all philosophers, making them eat serially. (ii) Shared: A separate thread executes the operations of each philosopher and a chopstick can be used simultaneously by both adjacent philosophers. (ii) Exclusive: Same as Shared but a chopstick cannot be shared and there is no mutual exclusion. (iv) Locked: Same as Exclusive except the operation to check if a chopstick is used is atomic and a thread waits until the necessary chopsticks are available, leading possibly to deadlock. (v) Polling: Same as Locked except that a thread does not wait. Instead, it releases the chopstick and keeps trying to get both chopsticks together until they are not in use, waiting for a time period between successive attempts. (vi) NoDeadlock: Same as Locked except that deadlock is prevented. (vii) ButlerCoordination: The prompt for the next course does not appear until all philosophers have eaten the previous course. Again, inheritance was used extensively to share code among the tests. The number lines of code for these seven tests were 165, 72, 17, 43, 15, 7, and 10, respectively. What is remarkable about these data is that the six concurrency checks added relatively little code to the sequential check – again because the lower half did not have to be written. Appendix I gives GitHub links to the source code of our checks.

The fact that the same abstractions were used in all checks shows their generality. However, as the two classes of applications are very different, techniques for using the abstractions were also very different. On the other hand, techniques for writing our tests for the Holy Grail assignments should easily transfer to all other concurrency assignments described in [22, 23] as their concurrency requirements are similar. The reported lines of code in Table I give an accurate indication of the relative ease of programming the tests. None of these tests was straightforward, even though an isolating architecture and high-level abstractions designed and implemented by their author were used for some of them. This is consistent with our experience that both concurrent code and tests are inherently difficult to implement.

B. Automatic vs Manual Testing

In our target course, we used both automatic and manual testing of concurrency and other constraints. Manual checking was done after automatic testing. It tried to find obvious false positives and negatives given by the tests, and ensured students had written code demonstrating that met requirements. Assignment 4, the assignment that covered concurrency, also covered exceptions and assertions. 31 Students submitted it to Gradescope for fully automatic grading and 27 of them submitted it to Sakai for manual grading.

Fig. 3 shows the rubric used for manual testing of Assignment 4. As it was a summer-course assignment, it had the size of three semester-course assignments. A single learning assistant (LA) graded all submissions. He gave extra credit for particularly nice demonstrations and examined source code when the demonstrations did not work. Our grading management system logged all grader interactions. Assuming a particular student would be graded without a break, we found that the average time to manually grade an assignment was about 7 minutes. This is a remarkably small time given the number of and kind of tasks performed by the LA. It is likely low because concurrency was visualized, our framework provided rubric and navigation support, the LA knew that automatic testing would probably catch his mistakes, and the vast majority of students had made sure they passed the tests before submission. For a small class whose assignment set is a one-off, it might be less time-consuming to grade only manually. As we point out, saving instructor grading time is not the only benefit of tests,

The average (standard deviation) of the concurrency scores given by the auto and human grader were 83.8%(30) and 76.4%(38) when considering only the 27 assignments graded both automatically and manually. This is somewhat surprising as both gradings were checking similar things. There are two reasons for the lower manual score. First, our automatic checks gave extensive partial credit based on the events observed, while manual checks could only look at the results shown on the display. Second, some students did not demonstrate features they had implemented, which were found by automatic checks, The second reason is an argument for adding manual checking to automatic grading, as, arguably, students understand better their solution if they demonstrate it working.

C. Automatic Testing: Student Experience

Automatic tests can be an extra collaborator for programmers, giving information about the requirements they are expected to meet, determining which of them they have met so far, and sometimes, based on the error messages, even giving direction on how to meet the pending requirements. On the other hand, they can be buggy, give misleading or confusing error messages, impose unintuitive requirements about entry points, and not identify the requirements they are testing, which is probably why Lumsdaine's talk [17] mentioned they led to student frustration. To reduce the chance of such frustrations, students were told they did not have to worry about running automatic tests – if they had implemented the assignment requirements correctly, regardless of test scores, they would get full points from manual grading. Every student still relied on our tests to ensure they met the requirements.

This does not imply there were no negative consequences. Students were told that if the tests gave unexpected messages, they would get prompt attention in Piazza forum posts and office hours. Since this was a summer course, there were scheduled

Zoom office hours 9 to 5 every weekday. Before an office hour visit, students were also asked to add a comment in a special office hours Piazza thread about the issue they would discuss.

We analyzed data from office hour issues on Piazza, regular Piazza posts, and Zoom audio transcripts of visits relevant to Assignment 4. We classified each of these items into the following categories – not about concurrency, about concurrency but not about tests, and about concurrency and tests. Table II shows the results. Assuming each item categorized as concurrency and tests had to do with meeting the constraints of tests or understanding test messages rather than how to meet concurrency requirements, the last column shows that there were far more questions about concurrency than concurrency tests. To the best of our knowledge, none of our tests had errors, though some of them did have misleading messages which we fixed when identified. Appendices II, III, and IV give the raw data reported in this table.

TABLE II. PIAZZA POSTS AND ZOOM TRANSCRPTS

Assignment 4	Total	Non Conc.	Conc.	Conc. & Tests.
OH Issues on Piazza	91	62	20	9
Zoom Transcripts	24	18	4	2
Piazza Posts	36	30	3	3

D. *Test-based Awareness*

Like our grading management system, our tests, when run on the computer of the students (as opposed to Gradescope) logged timestamped test executions. Based on these logs, we have developed an algorithm to determine how many times a test was attempted before it reached its final score [22], and how much time was spent during these attempts (assuming that a five-minute pause is a break). This algorithm, in turn, can be used to give test-based visual awareness of student activities while attempting the solution. Fig. 13 shows an example of such awareness created from 15 submitted logs.

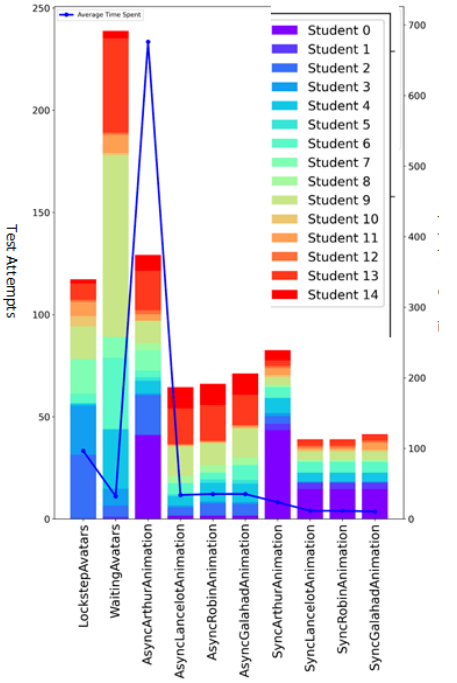

Fig. 13. Test-based Awareness

The stacked bars represent the number of attempts by each student to complete the given test. Each color represents a different student. The blue line represents the average time spent by students (in minutes) trying to complete the test.

How such awareness is used is, of course, a matter of future research but this example shows some possibilities. The figure confirms the intuition that once a student is able to asynchronously animate one Knight, then the work required to animate other Knights should be low, if code is reused properly. We see that the number of attempts and testing time associated with *AsyncArthurAnimation* is much more than what is associated with animation of other Knights such as Lancelot and Galahad. This test appeared earlier than others in the testing user interface and was therefore executed before others. The high number of attempts on *WaitingAvatars* is consistent with the fact that thread coordination is difficult. The relatively low time on it may show that a few changes whose effect is visualized can quickly correct the problem.

The figure also shows that there is not a strong correlation between the number of attempts and the amount of time spent trying to complete a given test. Most tests were worked on between 5 and 30 minutes except for the *LockstepAvatars* and *AsyncArthurAnimation* tests. The attempt data also illustrates that a small collection of students is often responsible for a very large portion of attempts on a given test. Student 9, for example, was responsible for almost half the attempts on the *WaitingAvatars* test. Looking at the distribution of attempts for a single student also demonstrates that most tests require few attempts, but a few tests may prove difficult. For example, Student 0 had fewer than 50 attempts on all tests except for the *AsyncArthurAnimation* and *SyncArthurAnimation* tests, which accounted for over 100 attempts each. This information, if gathered incrementally, could be used to offer help to those who are late to seek it.

VII. CONCLUSIONS

This paper makes several contributions. We survey and critique current efforts on testing student programs based on the extent to which they handle concurrency. We present a framework for manually and automatically testing a subset of concurrent programs that visualize concurrency and make all visualized events observable by observer objects. The framework consists of a novel grading management system that allows interactive testing and debugging of student programs, a producer-consumer architecture for writing observer-based concurrency tests, a generic event-based DBMS for automating the producer component, and the generic interleaving, partition, and match operations on data stored in the DBMS. We give our experience using the framework based on lines of code required to write tests, the time required to do manual grading, the scores assigned by manual and automatic grading, the contents of Piazza post and office visits. We also show the possibilities of creating new forms of visualization based on data logged by tests, which is another argument for writing them.

The main future work is to extend and adapt these ideas to procedural programming languages such as C and numeric assignments that do not inherently visualize concurrency. How to report and use test-based experiences is another exciting future direction. This paper motivates and provides a basis for such work.

ACKNOWLEDGMENT

This work was funded in part by NSF awards OAC 1829752 and 1924059. The in-depth comments of the reviewers improved the presentation of the research.

REFERENCES

[1] Brito, M.A.S., K.R. Felizardo, P.S.L. Souza, and S.R.S. Souza, Concurrent Software Testing: A Systematic Review, in Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging. 2011.

[2] McDowell, C.E. and D.P. Helmbold, Debugging concurrent programs. ACM Computing Surveys (CSUR), 1989. 21(4): p. 593-622.

[3] Chen, J. and S. MacDonald, Towards a better collaboration of static and dynamic analyses for testing concurrent programs, in Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging. 2008, ACM: Seattle, Washington. p. 1-9.

[4] Arjun Singh, S.K., Kevin Gutowski, Pieter Abbeel. Gradescope: A Fast, Flexible, and Fair System for Scalable Assessment of Handwritten Work. in Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale. 2017. ACM.

[5] Adams, J., R. Brown, and E. Shoop, Patterns and Exemplars: Compelling Strategies for Teaching Parallel and Distributed Computing to CS Undergraduates, in Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum. 2013, IEEE Computer Society. p. 1244-1251.

[6] Ayewah, N., D. Hovemeyer, J.D. Morgenthaler, J. Penix, and W. Pugh, Using Static Analysis to Find Bugs. IEEE Softw., 2008. 25(5): p. 22-29.

[7] Fraser, G., F. Wotawa, and P.E. Ammann, Testing with model checkers: a survey. Softw. Test. Verif. Reliab., 2009. 19(3): p. 215-261.

[8] Howden, W.E., Reliability of the Path Analysis Testing Strategy. IEEE Trans. Softw. Eng., 1976. 2(3): p. 208-215.

[9] Yoon, Y. and B.A. Myers. Capturing and analyzing low-level events from the code editor. in Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools. 2011. New York.

[10] Banerjee, U., B. Bliss, Z. Ma, and P. Petersen. Unraveling data race detection in the intel thread checker. in Proceedings of STMCS '06. 2006.

[11] Corporation, I. Intel® Thread Checker 3.1 for Windows*. 2018; Available from: https://www.polyhedron.com/threading_tools/reseller_productpage_thread_checker.html.

[12] Sack, P., B.E. Bliss, Z. Ma, P. Petersen, and J. Torrellas, Accurate and efficient filtering for the Intel thread checker race detector, in Proceedings of the 1st workshop on Architectural and system support for improving software dependability. 2006, ACM. p. 34-41.

[13] Btrace, B. BTrace User's Guide. 2014; Available from: https://kenai.com/projects/btrace/pages/UserGuide.

[14] Edwards, S.H. and M.A. Perez-Quinones, Web-CAT: automatically grading programming assignments. SIGCSE Bull., 2008. 40(3).

[15] Edwards, S.H., Z. Shams, M. Cogswell, and R. C. Senkbeil, ,. Running students' software tests against each others' code: New life for an old "gimmick". in Proceedings of the 43rd ACM SIGCSE. 2012.

[16] Jackson, D. and M. Usher. Grading student programs using ASSYST. in Proceedings of the 28th SIGCSE technical symposium. 1997.

[17] Lumsdaine, A. The only constant is change: Maintaining an up-to-date HPC curriculum. in Proc. of EduHiPC-18 worksahop in IEEE HPC. 2018.

[18] Aziz, M., H. Chi, A. Tibrewal, M. Grossman, and V. Sarkar, Auto-grading for parallel programs, in Proceedings of the Workshop on Education for High-Performance Computing. 2015, ACM: Austin, Texas. p. 1-8.

[19] Grossman, M., M. Aziz, H. Chi, A. Tibrewal, S. Imam, and V. Sarkar, Pedagogy and tools for teaching parallel computing at the sophomore undergraduate level. J. Parallel Distrib. Comput., 2017. 105(C): p. 18-30.

[20] Spacco, J., D. Hovemeyer, W. Pugh, F. Emad, J.K. Hollingsworth, and N. Padua-Perez. Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses. in Proc. ITiCSE. 2006.

[21] Sarkar, V., M. Grossman, Z. Budimlic, and S. Imam, Preparing an Online Java Parallel Computing Course. 7th NSF/TCPP Workshop on Parallel and Distributed Computing Education (EduPar-17). 2017.

[22] Dewan, P., S. George, A. Wortas, and J. Do, Techniques and tools for visually introducing freshmen to object-based thread abstractions. Journal of Parallel and Distributed Computing, 2021. 157.

[23] Dewan, P. Visually Introducing Freshmen to Low-Level Java Abstractions for Creating, Synchronizing and Coordinating Threads. in Proc. of EduHiPC-19 worksshop in IEEE HiPC. 2019. Hyderabad: IEEE.

[24] Gamma, E., R. Helm, R. Johnson, and J. Vlissedes, Design Patterns, Elements of Object-Oriented Software. 1995, Reading, MA.: Addison Wesley, 1995.

REPRODUCIBILITY APPENDIX I:EVENT DBMS/CHECKS

Our implementation of the event DBMS and associated generic operations is available from the following GitHub directory:
https://github.com/pdewan/GraderBasics/tree/master/src/gradingTools/shared/testcases/concurrency/propertyChanges

The checks we wrote using the event DBM for the various versions of the dining philosopher problem are available here:
https://github.com/pdewan/DemoCourseLocalBasicChecks/tree/master/src/gradingTools/comp999/assignment2

The concurrency checks we wrote using the event DBMS for the target class are available in the following GitHub files:

https://github.com/pdewan/Comp401LocalChecks/blob/master/src/gradingTools/comp301ss21/assignment4/async/AbstractionAsyncArthurAnimationTestCase.java

https://github.com/pdewan/Comp401LocalChecks/blob/master/src/gradingTools/comp301ss21/assignment4/sync/AbstractionSyncArthurAnimationTestCase.java

https://github.com/pdewan/Comp401LocalChecks/blob/master/src/gradingTools/comp301ss21/assignment4/coordination/AbstractionLockstepAvatarsAnimationTestCase.java

https://github.com/pdewan/Comp401LocalChecks/tree/master/src/gradingTools/comp301ss21/assignment4/coordination

The concurrency checks we wrote without using the event DBMS for the target class are available in these files:

https://github.com/pdewan/Comp401LocalChecks/blob/master/src/gradingTools/comp401f16/assignment10/async/testcases/AsyncArthurAnimationTestCase.java

https://github.com/pdewan/Comp401LocalChecks/blob/master/src/gradingTools/comp401f16/assignment11/sync/testcases/SyncArthurAnimationTestCase.java

https://github.com/pdewan/Comp401LocalChecks/blob/master/src/gradingTools/comp401f16/assignment12/waitnotify/testcases/LockstepAvatarsAnimationTestCase.java

https://github.com/pdewan/Comp401LocalChecks/blob/master/src/gradingTools/comp401f16/assignment12/waitnotify/testcases/WaitingAvatarsAnimationTestCase.java

REPRODUCIBILITY APPENDIX II:OFFICE HOUR VISIT ISSUES

The following table shows each office hour issue posted on Piazza and its classification into non concurrency (N), Concurrency but not Testing (C), and Concurrency and Testing (C & T)

Office hour Piazza Posts	**C/T**
How to run the debugger properly to see the expandable threads	N
syntax errors	N
discuss the demoing of the view class	N
how the RestrictedLine connects to the new Legs class	N
how to account for the left and right lines and their subsequent angle range	N
Observing Bridge-Scene View	N
parts 3 and 4 of A4	C
failing some controller checks + throw the exceptions for part 2 of A4	N
button issue + localchecks for the buttons and syncanimations NOT passing	C&T
assertions are not getting call correctly	N
failing all the SceneControllerButton checks	N
discuss part 1 of A4	N
discuss syncAnimation and AsyncAnimation	C&T
discuss TokensRun tests	N
help with the async/sync methods	C
discuss SceneControllerButtonDynamics	N
A4 part 1's last section with the JButtons	N
logistical questions around part 1 of A4	N
how to implement part 1 of A4	N
not able to get the ImageShapes for each avatar to display in the scenes	N
implementing the button part (part 1 of A4)	N
unsure of what type of preconditions that should be looking for + when the notification of event changes should be firing	N
Object Editor and the Keyboard presses	N
discuss an issue with waiting avatars	C
not passing the SceneControllerButtonDynamics test + keyboard presses in the object editor	N
failing a scenecontrollerbuttondynamics test and impossible angle exception test	N
failing the scenecontrollerbuttondynamics test + don't understand how broadcastingclearancemanagers work	N
how to properly register the scene controller as a listener	N
discuss the use of the Coordinating Animators in the bridge scene class for the lockstep animation methods	C
gradescope grade does not match localcheck grade	N
clarify some parts of bridgescenecontroller + some null pointers getting in bridgescene	N
discuss exceptions	N
get help on the Synch method	C
check if the student's project has been sent out successfully	N
get help on the async/sync method	C
failing bridge scene controller tests	N
unable to get the avatars to speak	N
ask about preconditions and button listeners in part 1 of A4	N
the button controllers in part 1 of A4	N
exceptions after creating legs and restricted line classes	N
failing a few syncanimation and controller checks	C&T
discuss how the animation commands should be called within (or outside of) the asynchronous methods	C
ask about the synchronous and coordinated methods in part 4 of A4	C
animating the scene using the buttons	N
discuss part 4 of A4	C
mouseListener stopped working	N
not passing some checks regard to buttons	N
failing the bridgescenebuttoncontroller test	N
failing the button dynamics test	N
failing the scene controller dynamics test	N
finding a new error with the test	N
get help with Coordinated Animation	C&T
A4 scene and button dynamics do not work on gradescope	N
passing localchecks for A4 part 1 (AssertingBridgeSceneDynamics) but not in gradescope	N
failing A4 SceneControllerButtonDynamics test	N
discuss the SceneControllerButtonDynamics test in SceneControllerButtons of A4	N

discuss child thread not found error for the sync and async animations	C&T
how to implement abstract classes in part 1 of A4	N
understanding of the extended console view and the implementation of buttons	N
get some more clarification on the buttons part of A4 + clear up some implementation concerns about the factory method for the Part 2 exceptions	N
get a better idea of how to implement part 1 of A4	N
clear up some wording issues of part 2 of A4	N
clarify some parts of BridgeSceneDyanamics	N
failing style checks tests + get some help with part 4 of A4	C
get some help with part 4 of A4	C
failing Asynchronous animations test	C&T
questions about gradescope test	N
get help with Synch method and Buttons Dynamics	C
discuss an issue with waiting avatars	C
mismatch in checks between Gradescope and Localchecks	N
questions about exceptions	N
discuss exceptions and Asynchronous animation	C
failing SceneControllerButtons and AssertingBridgeSceneDynamics tests + ask about part 4 of A4	C&T
questions about manual grading of A4	N
getting errors in sync and async animations	C
discuss SceneControllerButtonDynamics	N
get help with async animations	C
talk about Observing Bridge-Scene View	N
discuss some concepts of paint() and repaint()	N
creating the buttons in part 1 of A4	N
resolve some issues with part 1 and part 2 of A4 + get some help understanding part 3	C
get a better understanding of part 3 for A4	C
talk about part 2 of A4 and creating the exception class	N
discuss Part 2 of A4	N
mouseListener and keyListener are not working properly	N
approach method and say method not working in localchecks	N
discuss how to debug A4 for the LocalCheck errors such as "Child thread 1 not found" and "Child thread not found."	C&T
add buttons to the JPanel	N
figuring out how the lockstep methods work	C
discuss Asynchronous Animation and controller calls	C
discuss waitingAvatars and LockstepAvatars	C&T

REPRODUCIBILITY APPENDIX III: ZOOM TRANSCRIPTS

The following table shows the summary of each Zoon transcript during office hours and its classification into non concurrency (N), Concurrency but not Testing (C), and Concurrency and Testing (C & T)

Office hour transcript summary	**C/T**
Debugging in practice 4.1	N
Exceptions	N
Questions on how to implement restricted legs and legs throwing exceptions	N
The head is in the correct position, but the body is not right; wires water only see like property check that that's for x's and y's (either rotatable by or locatable)	N
Null pointer exception	N
Lots of things wrong on local checks	N
Confused on how to go about the threads part	C
Part four in part one	C
This tango is caught by rotating line or restricted line	N
Button issue	N
Exceptions	N
Mouse clicking no longer work	N
Buttons not focus	N
Scene controller dynamics; On the key events seems it's not firing	N
Scene controller, unable to access anything when you call the button presses	N
Scene controller, unable to access anything when you call the button presses	N
Passing everything on grade scope, but issues with my walk step methods	C
Controller factory method	N
Messages for different things (extends io exception)	N
Avatar moving	C
Grades group of the local checks (bridge team dynamics)	N
Sync controller buttons (New value false does not equal approach button enabled status)	C&T
Waiting Avatar; about scene button; about synchronous; it says that lockstep	C&T
Explain synchronize keyword	N

REPRODUCIBILITY APPENDIX IV: PIAZZA POST SUBJECTS

The following table shows the subject lines of each Piazza post classification into non concurrency (N), Concurrency but not Testing (C), and Concurrency and Testing (C & T)

Piazza Posts	**C/T**
Assignment 4	N
Assignment 4 Due Date	N
Early Deadline	N
A4 Part 1 OE not making grey if precondition is false?	N
A4 Part 1 Dynamically Enabled Buttons for Calling Asserting Methods	N
checkstyle	N
Clarification with Assignment 4 Instructions	N
Failing SceneControllerRegistersAsActionListener	N
Failing SyncAnimation tests, unsure why from console trace	N
Delayed Events Error with implemented sleep call	C&T
JButton problem	N
Indentation Issue	N
Animation for speaking not loading properly	N
AssertingBridgeSceneDynamics failing	N
Steps and Claps	C
Lockstep test failed	N
Typo in grader	N
scene controller button tests	N
Exception Thrown and Grader Test	N
A4 part lockstep animator issue	N
Lockstep Animators	C
A4 gradescope	N
Style Checks Partially Passing	N
Gradescope not match with Checkstyle	N
A4 sakai submission	N
GradeScope grades not match with localcheck	N
SceneControllerISAPropertyChangeListener not passing	N
BridgeScene Dynamics checks- failing preconditions	N
Table Demonstration	N
Waiting Test on Local Checks not matching Gradescope test	N
Asynch/Synch/Wait Animations and the Use of synchronize KeyWord	C&T
LocalChecks not matching gradescope for SceneDynamics and ButtonDynamics	C&T
A4 SceneControllerButtonDynamics Error	N
AssertingBridgeSceneDynamics is giving me a null point exception	N
AssertingBridgeSceneDynamics	N
A4 Exception in thread... EmptyStackException	C