Infrastructure for Writing Fork-Join Tests

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

ABSTRACT

We have developed a software infrastructure for testing multi-threaded programs that implement the fork-join concurrency model. The infrastructure employs several key ideas: The student solutions use print statements to trace the execution of the fork-join phases. The test writer provides a high-level specification of the problem-specific aspects of the traces, which is used by the infrastructure to handle the problem-independent and low-level details of processing the traces. During performance testing, trace output is disabled automatically. During functionality testing, fine-grained feedback is provided to identify the correct and incorrect implementation of the various fork-join phases. Tests written using our infrastructure have been used in an instructor-training workshop as an instructor agent clarifying requirements and checking in-progress work. The size of the code to check the concurrency correctness of final and intermediate results was far smaller than the code to check the serial correctness of such results.

CCS CONCEPTS

• Parallel Computing Methodologies • Software Notation and Tools

KEYWORDS

Software Testing, Concurrency, Education, Fork, Join, Awareness, Intelligent Tutor Systems

1 Introduction

Most of the research on HPC (High-Performance Computing) has focused on identifying concurrency concepts, pedagogical models, worked examples, and assignments in various kinds of undergraduate courses. Our work addresses a different but related topic - techniques for automatically testing multi-threaded programming assignments.

Such tests can, of course, automate grading, making education more scalable and, hence efficient. Relatedly, they can provide quicker feedback than manual grading, which is important in short-duration training sessions - such as the instructor-training sessions supported by NSF's Center for Parallel and Distributed Computing Curriculum Development and Educational Resources (CDER)[1].

More important, trainees can run tests on in-progress code to clarify requirements, determine if they are on the right track, and if not, how to get back on track. Thus, the tests can take the form of an always-available instructor agent who clarifies requirements, checks partial work, and gives directions for the next phase of work. Such help is particularly important for multi-threaded programs, which are notoriously difficult to write and debug.

Tests run on in-progress work can give valuable feedback also to instructors. The logged results of these tests can provide instructors with awareness of unseen partial work, which can be used to manually or automatically infer if the assignment is too easy or difficult, or difficult only for a subset of identified students. These inferences can, in turn, be used by the instructor to modify the current or future assignment, and provide unsolicited help to students who are in apparent difficulty or have taken the wrong path. Such monitoring is more important in HPC education because of the relative infancy of this area.

Traditional software testing, however, is at odds with the notion of using concurrency to improve performance. The reason is that such testing gives feedback based on the relationship between the input and output of the tested code, and performance improvements do not change this relationship. Two approaches have been used so far to address these limitations of traditional testing.

The first is to use performance rather than input/output relationships to test concurrent code [2]. While such an approach could offer the advantage of scalable quick grading, it cannot provide feedback that pinpoints problems in the failed code.

The second is to assume that multi-threading is used, not for high-performance computing, but to visually create concurrent animations [3]. These animation actions are observed by testing code to check them for correctness. This approach, of course, works only for assignments that create such visual animations. Moreover, as there is no common model followed by such animations, low-level testing details cannot be automated.

We have developed a software infrastructure to address these limitations of existing work. An overview of an earlier version of

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM 979-8-4007-0785-8/23/11…$15.00
https://doi.org/10.1145/3624062.3624098

the infrastructure, and how it was used for training has been published previously [4]. In this paper, we focus in-depth on the infrastructure, describing it from the perspective of a test writer. Section 2 lists the set of requirements it was designed to meet. Section 3 identifies the key concepts to meet these requirements. Section 4 describes various aspects of the infrastructure that implements these concepts. Section 5 evaluates our infrastructure. Finally, Section 6 gives conclusions and directions for future work.

2 Requirements

As mentioned in the introduction, two approaches to concurrency testing have been used before – performance testing and functionality testing of observable animation events.

The former is suitable for programs that meet two conditions. First, their threading level can be increased dynamically. Second, increasing the threading level when idle cores are available improves performance. It has the advantage that no problem-specific testing code has to be written as no problem-specific requirement is checked as long as these two conditions are met.

The latter is suitable for a different class of concurrent programs – those that use threads to create concurrent animations in the user-interface using animation events that can be observed by arbitrary observer objects. We refer to these as observable animations. The level of multi-threading does not have to be changeable. This approach requires low-level problem-specific testing code to be written. This code can identify to what degree a solution meets each requirement and thus, in comparison to the former, can help the programmer better pinpoint problems.

Based on this discussion, we can identify our requirements.

- The infrastructure should support both performance checking and functional testing of observable animations so that it subsumes rather than replaces the features of the state-of-the-art.
- It should allow functionality testing of programs that use multi-threading to increase performance rather than implement concurrent animations.
- It should provide high-level support for writing code to test these programs. Ideally, the test code should be responsible only for specifying the problem requirements - how these requirements are checked should be the responsibility of the underlying testing infrastructure.
- Such higher-level support, ideally, should also be layered above lower-level support implementing observable concurrent animations,
- A functionality-testing approach that does not pinpoint the problems in an erroneous solution is no better than a performance-testing approach. Therefore, we require that the infrastructure provide fine-grained checking and reporting of errors – in particular, it should check the correctness of final and intermediate results.

3 Key Ideas

In the discussion below, we motivate and *highlight* our key ideas.

To provide high-level support for testing the functionality of concurrent programs, *the testing infrastructure must be based on a high-level model of concurrent programming* so that it can automate testing of those aspects that are common to programs that follow the model, leaving problem-specific aspects to the test code.

We provide testing support for the *fork-join model* defined as follows: The main or root thread forks zero or more multiple worker threads to do different parts of a problem and makes a blocking call to join the worker threads. Each worker thread executes a loop with zero or more iterations to compute zero or more results and then makes these results accessible to the root thread. When all worker threads have terminated, the join call unblocks the root thread, which can then print the combined results of the worker threads. This is a practical model as it covers many worked examples and assignments used in undergraduate courses.

As mentioned in the introduction, a major hurdle to testing HPC parallel programs is that parallelization does not change the user-interface. To illustrate, consider Fig. 1, which shows the Hello World program from an online chapter introducing C++ concurrency [5].

```
void hello() {
    std::cout<<"Hello Concurrent World\n";
}
int main() {
     std::thread t (hello);
     t.join();
}
```

Fig. 1. C++ Program Producing Concurrency-Unaware Output

In this program, the thread that executes the main method forks a new child thread, t, and then makes the blocking join call. The child thread prints "Hello Concurrent World", and then terminates, which causes the main thread to unblock from the join call and terminate. This output does not indicate the presence of multi-threading – the main method could have directly printed it without forking and joining a child thread. Thus, a testing program cannot tell, by looking at only the output, that a fork and join occurred. The reason is that the output is **concurrency-unaware** – it gives no indication that multiple threads were involved in its creation.

A possible solution is suggested by an OMP variation [6] of this example shown in Fig. 2. The figure has an OMP pragma to create multiple threads to execute the print call, which displays the text "Hello World" along with the id of the thread that executes this call. As a result, the output is **concurrency-aware** – it shows that concurrent threads were involved in its creation. In this example, the reason for creating concurrency-aware output was pedagogical – the person running this code is made aware of multithreading and the role of each thread.

Concurrency-aware output can also be used to create test code that checks that multithreading occurred. In the above example, the test code can parse the output to determine the number of different threads created. Therefore, another important concept in our design is the *ability to associate the output of a testable concurrent program with the thread that produced the output* (without necessarily displaying the thread information in the output).

```
#pragma omp parallel
{
   printf("Hello World… from thread =  %id\n",
          omp_get_thread_num());
}
```

(a) OMP Concurrent Progam

```
Hello World… from thread = 1
Hello World… from thread = 0
Hello World… from thread = 4
…
```

(b) Concurrency-Aware Output

Fig. 2. OMP Concurrent Progam with Concurrency-Aware Output

Consider a variation of the two programs above in which the forked threads do not directly produce their output. Instead, they write it to some data structure shared with the main thread, which then combines and prints the results from all threads to the console. This is how a large variety of concurrent fork-join programs work. In this case, the output is printed by the root thread, making it concurrency-unaware. Fig. 3 shows the output of a program that determines the number of primes in a set of random numbers. The root thread prints the set of random numbers it computes, asks different worker threads to determine the number of primes in different subsets of this set, and after unblocking, prints the sum of the computed numbers. Both the input (random numbers) and final output (number of primes) are printed by the root thread.

To address this problem, *we require the threads in the tested concurrent programs to produce intermediate output tracing the steps they perform to process their part of the problem*. The exact nature of the steps is problem-specific and determined by the test writer. As with the final output, our design allows testing code processing the output to determine which thread created it. Fig. 4 shows part of the intermediate output produced by one of the worker threads in the prime example. As we see here, the id of the worker thread is different from that of the root thread.

```
Thread 23->Random Numbers:
[509, 578, 796, 129, 272, 594, 714
Thread 23->Total Num Primes:1
```

Fig. 3. Root Thread Printing its Input and Final Output

```
Thread 24->Index:0
Thread 24->Number:509
Thread 24->Is Prime:true
```

Fig. 4. Example Intermediate Output of Worker Thread

Not all solutions to a problem – concurrent or serial – will produce the same sequence or even set of prints. For example, different solutions to the prime number example of Fig. 3 and 4 will work on different sets of random numbers, and thus, produce different outputs. Thus, *the testing program must extract relationships between the prints rather than their exact content.*

These relationships have to do with state changes the parallel program makes to **logical variables** – these are conceptual variables associated with all expected solutions to the problem and may not translate directly to internal variables. *The testing program should work in terms of these logical state changes directly without having to parse the output*. This principle implies that a testable program should use a standard method for printing logical variables and the testing infrastructure should parse the output for the testing program. Fig. 3 and 4 show the nature of such parsable output. In Fig. 3, the root thread prints the values of two logical variables: the set of random numbers and the total number of primes. In Fig. 4, a worker thread working on the 0^{th} prime number prints the value of three logical variables: the index of the number in the input array, the number itself, and whether it is a prime or not. All solutions to the problem must use the same names for these logical variables, which become part of the assignment requirement.

To support the existing technique of performance-based testing, *we require the tested programs to provide a main argument that allows the number of threads in the program to be varied* – the exact form of this argument is decided by the associated test.

Producing intermediate output for functionality testing is at odds with performance-based testing of parallel programs. Multi-threading has an impact on performance only when the number of iterative steps performed by the worker thread is large. To ensure that the loads of worker threads are balanced, each iterative step would have to produce some output (Fig. 4) so that the number of outputs/steps can be compared. A program written for functionality testing would be artificially slowed down and unnecessarily clutter output (possibly using all the heap space if output is stored) when used for performance testing. *Our solution is a mechanism to dynamically turn off all prints for performance testing*. To layer our fork-join infrastructure support on top of existing work on testing observable animations, *we make the intercepted prints observable by arbitrary observer objects*.

4 Testing Infrastructure

We have implemented these key ideas in a testing infrastructure written in Java that expects the testing and tested programs to also be written in Java. It is layered on top of our previous Java infrastructure for testing observable concurrent animations, which in turn, is layered on top of the Java Junit testing infrastructure. In this section, we overview the main components of our infrastructure: (i) the user-interface provided to programmers of the tested code; (ii) the programming-interface provided to programmers of the tested code; (iii) the programming-interface provided to programmers of the testing code; and (iv) and how our new infrastructure support is layered on top of our existing layers.

4.1 User-Interface of Instructor Agent

Programming environments such as Eclipse and IntelliJ provide facilities to allow users to run JUnit tests interactively. We have developed an additional user-interface for such interactive testing that adds two features to JUnit. First, it is independent of the programming environment and can be created from the command line. Second, it displays the score assigned to each test by the testing program along with the error message.

JUnit provides programmers of tested code the ability to group tests into suites. To create our user-interface, the user simply runs the suite created for the problem being tested. Fig. 5 shows the user-interface created when the suite for the primed number problem is

run. This suite consists of two tests, one for functionality and another for performance. The figure shows the result of double-clicking on the functionality test to run it. The user-interface displays a score of 32 out of 40 for this test along with a message (not shown completely) indicating which requirements were met and not met.

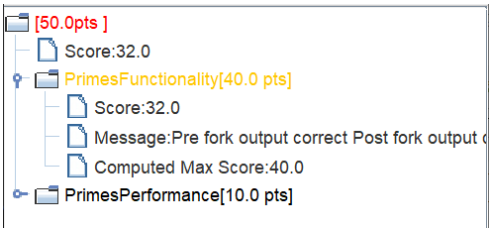

Fig. 5. Plugin Independent Testing User-Interface

This user-interface is optional in that students who are confident that they have met all requirements can simply submit their solution to Gradescope for grading. It is necessary only for students wishing to use the tests to iteratively refine their solution.

JUnit has been used earlier [7] to test the correctness of parts of a complete concurrency assignment such as a shared counter object. In our work, we use JUnit to test the complete assignment by always calling the main method, and letting it run to full completion before analyzing its output. As a result, unlike the previous work, we do not need to directly catch exceptions and other intermediate errors created by the tested parts – these are expected to be manifested as incorrect traced final output.

4.2 Programming Interface for Tested Program

Our infrastructure provides tested programs with a special method, `printProperty(String, Object)`, to output changes to logical variables in a standard form. Fig. 6 shows its use in the loop executed by a worker thread to identify prime numbers in the input array. The method takes two arguments, the name of the logical variable and its current value. It prints the current thread id along with its arguments. Thus, the three calls to this method in Fig. 6 (lines 9, 11, and 16) output the three lines, respectively, shown in Fig. 4. Logical variables in our implementation correspond to JavaBean properties, so we use the term *property* and logical variable interchangeably

In this example, problem-independent code prints the index of the list of items processed, along with the number at that index, and problem-specific code traces the computation done with this number. Tracing the computation performed by each iteration allows the intermediate results produced by the iteration to be checked by testing code, as we see later.

A tested program can use a Java print method to print an object O of type T directly. Our infrastructure intercepts such a print, does not change its output, but internally stores it as the setting of a logical variable named T whose value is O. Our implementation of the fork-join Hello World uses a Java print call:

```
System.out.println("Hello Concurrent World");
```

resulting in the output:

```
Hello Concurrent World
```

This call internally results in a trace of a logical variable named "String" being set to "Hello Concurrent World". Even though no thread id is output, internally, a thread object is kept with this trace. A corollary of this discussion is that a test program that tries to print the wrong thread id cannot fool the infrastructure as it internally keeps the object associated with the printing thread.

Both the tested and testing programs can invoke the infrastructure-provided method, `setHideRedirectedPrints(boolean)`. It is used to disable and enable the intercepted prints when doing performance and functionality testing, respectively. A disabled print produces no output and makes no changes to the trace kept by the infrastructure.

```
1  public static void fillPrimeNumbers(
2    int[] aNumbers,
3    int aStartIndex, int aStopIndex) {
4    int aNumberOfPrimeNumbers = 0;
5    for (int index = aStartIndex;
6         index < aStopIndex;
7         index++) {
8      // Start of typical iteration code
9      printProperty("Index", index);
10     int aNumber = aNumbers[index];
11     printProperty("Number", aNumber);
12     // End of typical iteration code
13
14     // Start of problem-specific iteration code
15     boolean isPrime = isPrime(aNumber);
16     printProperty("Is Prime", isPrime);
17     if (isPrime) {aNumberOfPrimeNumbers++;}
18     // End of problem-specific iteration code
19   }
```

Fig. 6. Typical Use of printProperty

4.3 Programming Interface for Testing Program

Recall that our goal is to make the testing program responsible for specifying the "what" of testing, leaving the "how" to the infrastructure. Fig. 7 illustrates how we achieve this goal for performance testing. It gives the full code of the test program written for a solution to the prime number problem that takes two main arguments: the number of random numbers to be generated and the number of threads to be created to process these numbers.

In our implementation, each testing program is a subclass of a predefined abstract infrastructure class. A performance-testing program is a subclass of the predefined class `AbstractConcurrencyPerformanceChecker` (lines 2 and 3, Fig. 7). The testing program overrides concrete methods in the class to specify the testing requirements (lines 10-23, Fig. 7). It uses the predefined annotation MaxValue to specify the score assigned to the test (line 1, Fig. 7). In this paper, the annotation, *@Override*, is not used to label overridden methods for space reasons. All methods given here are overriding methods.

The overridden methods in this example are *parameter methods* as they specify infrastructure-defined parameters of testing. Such a method has the form:

protected T p() { **return** v}

It tells the infrastructure that the infrastructure-defined parameter of type T has value v. To do performance testing, the tested program must be executed with different arguments specifying low and high numbers of forked threads. These are specified through

the string-array parameters `lowThreadArgs` (lines 13-16, Fig. 7) and `highThreadArgs` (lines 17-20, Fig. 7), respectively. The name of the testable program is given by the parameter `mainClassIdentifier` (lines 10-12, Fig. 7).

```
1  @MaxValue(10)
2  public class PrimesPerformance extends
3    AbstractConcurrencyPerformanceChecker {
4  final String TESTED_CLASS_NAME =
5     "ConcurrentPrimeNumbers";
6  final String NUM_RANDOMS = "100";
7  final double MINIMUM_SPEEDUP = 1.5;
8  final String MIN_THREADS = "1";
9  final String MAX_THREADS = "4";
10 protected String mainClassIdentifier() {
11     return TESTED_CLASS_NAME;
12 }
13 protected String[] lowThreadArgs() {
14     return new String[] {
15         NUM_RANDOMS, MIN_THREADS};
16 }
17 protected String[] highThreadArgs() {
18     return new String[] {
19         NUM_RANDOMS, MAX_THREADS};
20 }
21 protected double expectedMinimumSpeedup() {
22     return MINIMUM_SPEEDUP;
23 }
```

Fig. 7. Prime Performance Tester

The code in Fig. 7 is a formalization of the performance and testing requirements of the expected solution. The former requirements indicate that the solution must vary the number of random numbers and threads based on its performance and provide a speedup of at least 1.5. The latter indicate that the program must have the standard name, `ConcurrentPrimeNumbers`. The testing requirements would not be needed if the solution were not to be automatically assessed, so they represent the testing overhead on the programmer. Arguably, this overhead is small for performance.

Our infrastructure processes this specification as follows: It runs the named program with `lowThreadArgs` and `highThreadArgs` a default number of times (10), which can be changed by overriding another parameter method. Based on the total times taken in these two cases, it computes the speedup, and gives full points if it is more than MINIMUM_SPEEDUP and 0 pts if it does not. In general, when it deducts points, it gives the reason for it – in this case, it indicates the difference between the expected and actual speedups. Thus, we see here how the testing program and infrastructure are responsible for the "what" and "how" of testing. These two tasks are relatively simple for performance testing.

They are more complicated for functional testing. This is because the testing program must specify the nature of a correct trace. Such a trace has three aspects: *static syntax*, *dynamic syntax*, and *semantics*.

Static syntax determines the syntax of print lines, which, in turn, is determined by the names and types of the logical variables to be printed by the tested program. For example, in the traces of Fig. 3 and 4, it specifies the names of the logical variables and the types of values that follow these names.

Dynamic syntax determines the order and number of each kind of logical variable traced. For the prime number problem, it specifies the number of triples of Fig. 4 output in the iterative phase, and the order of the triples relative to other traces of the program.

Semantics can be broken into serial and concurrency semantics. The former check whether the expected final result is computed (e.g. number of primes) and would be relevant even if the problem was solved using a single thread. The latter check multi-thread requirements (e.g. correct number of interleaved threads were forked). To help pinpoint problems, our tests check intermediate results for both kinds of correctness (e.g. does each iteration compute the correct prime; were the number of primes computed by different threads summed up correctly without race conditions). Thus, both kinds of semantics can be subdivided into *final* and *intermediate* semantics.

Based on the discussion above, a trace corresponds to a programming language (associated with syntax and semantics) and a testing program corresponds to a programming language translator. It is, of course, possible to use general-purpose program translation tools to check traces, but they can be expected to be more cumbersome to use than necessary because of their generality.

Therefore, our infrastructure offers more specialized and hence higher-level support that leverages three important properties of our traces: (a) they consist of prints of logical variables of different types, which can be processed by regular expressions rather than grammars; (b) their order is determined implicitly by the phases of the fork-join model, so no explicit ordering constructs are required; (c) only the iteration phase requires a dynamic number of prints, which depends on the number of expected iterations specified by the testing program.

For static and dynamic syntax, our infrastructure provides parameters to specify (a) the names and types of logical variables to be output in each of the predefined fork-join phases, (b) the total number of iterations to be processed, and (c) the number of threads to be forked to process these iterations. For semantics, it provides callbacks defined through overridden methods in the testing program.

Fig 8. Illustrates these concepts for the iteration phase using our running example. Lines 1 to 4 declare an array that specifies the names and types of the properties to be output in each iteration, that is, the static syntax of this phase. This array is returned by the parameter method, `iterationPropertyNamesAndType`.

The dynamic syntax is specified by the parameter method `totalIterations`, which returns the total number of iterations performed by all threads, together, and hence the total number of times the three iteration logical variables are to be output. In this problem, one iteration is performed for each random number, so the total number of iterations is the same as the total number of random numbers to be generated.

The iteration semantics is specified by the semantics check method, `iterationEventsMessage`. It is called for each set of trace outputs in an iteration. Its first argument is the thread that did the output and the second is a Map giving the names and values of the iteration properties traced in the iteration. A semantics check method can perform arbitrary checks based on its parameters and the global state created by previous invocations of semantics check methods. In this example, the code checks if the value of the "Number" property output in this iteration is consistent with the "Is Prime" property output in this iteration. To do so, it calls the custom function `isPrime(int)`, which is expected to correctly determine if a number is prime or not.

```
1  final Object[][] ITERATION_PROPERTIES = {
2      {"Index", Number.class},
3      {"Number", Number.class},
4      {"Is Prime", Boolean.class}
5  }
6  final int NUM_THREADS = 4;
7  final int NUM_RANDOMS = 7;
8  protected Object[][] iterationPropertyNamesAndType(){
9      return ITERATION_PROPERTIES;
10 }
11 protected int totalIterations() {
12     return NUM_RANDOMS;
13 }
14 protected String iterationEventsMessage(
15   Thread aThread,
16   Map<String, Object> aNameValuePairs){
17     boolean isPrime =
18       (boolean) aNameValuePairs.get("Is Prime");
19     if (isPrime) {
20         numPrimesFoundByCurrentThread++;
21     }
22     int aNumber =
23        (Integer) aNameValuePairs.get("Number");
24     boolean isActualPrime = isPrime(aNumber);
25     if (isPrime != isActualPrime) {
26        return "Is Prime output as " +
27        isPrime + " for number " + aNumber +
28        " but should be " + isActualPrime;
29     }
30     return null;
31 }
32 protected int numExpectedForkedThreads() {
33     return NUM_THREADS;
34 }
```

Fig. 8. Prime Iteration Functionality Tester

A test program can similarly specify the properties and semantics checks for the three other fork-join phases: pre-fork, post-iteration, and post-join. The appendix gives the complete code of the program for testing the functionality of our prime example.

Fig. 9 gives part of the traces of a correct implementation of the prime example, that is, one whose traces are found correct by the test program. It is embellished with comments to delineate the various fork-join phases. In addition to the three iteration properties discussed above, to be printed by each forked worker thread, the complete implementation specifies:

(a) one pre-fork property, "Random Numbers", to be printed by the main root thread, which is the array of random numbers created by the tested program (lines 2-3, Fig 9.);

(b) one post-join property, "Total Num Primes", again to be printed by the main thread, which is the number of primes in this array found by the tested program (line 39, Fig. 9),

(c) one post-iteration property, "Num Primes", to be printed by each forked worker thread, which is the number of primes it found in its iterations (lines 28, 30, 34, 37, Fig. 9).

Because of interleaving, the iteration and post-iteration phases of the threads are mixed in the output. The program correctly processes the required number of random numbers, 7, specified in the testing program (lines 11-12, Fig. 8). Four worker threads are involved in the fork phase (iteration + post-iteration), which is the number expected by the test program (lines 32-34, Fig. 8). These threads interleave their output, so have to address any synchronization conditions this concurrency causes. Moreover, the load is as balanced as it can be: each thread except Thread 27 processes two numbers, with Thread 27 processing one number (This cannot be verified in Fig. 9 as some of the iterations have been cropped out). Each iteration of a thread correctly determines if the indexed number is a prime. The number of primes reported by each thread in its post-iteration phase is consistent with the primes it found, which is verified by the semantic post-iteration check given in the appendix. In addition, the total number of primes reported by the root thread (line 39, Fig. 9) in the post-join phase is the sum of the number of primes reported by each thread, which is verified by the post-join check given in the appendix. Hence, the test gives full points out of the maximum (line 1, Fig. 8), that is, 100 % (line 41, Fig. 9). This trace demonstrates to both the test program and the end-user viewing it that each phase is correct. Arguably, to the end-user, it also illustrates how concurrency in general and fork-join in particular work, and thus serves a useful pedagogical purpose even if automating assessment was not a goal.

```
1  //start pre-fork
2  Thread 23->Random Numbers:
3  [166, 634, 795, 571, 894, 509, 51]
4  //start fork
5  //start, Thread 24 iteration
6  Thread 24->Index:0
7  Thread 24->Number:166
8  Thread 24->Is Prime:false
9  //start, Thread 26 iteration
10 Thread 26->Index:4
11 //start, Thread 25 iteration
12 Thread 25->Index:2
   [illegible]
27 //start, Thread 25 post-iteration
28 Thread 25->Num Primes:1
29 //start, Thread 27 post-iteration
30 Thread 27->Num Primes:0
31 Thread 26->Index:5
32 Thread 26->Number:509
33 //start, Thread 24 post-iteration
34 Thread 24->Num Primes:0
35 Thread 26->Is Prime:true
36 //start, Thread 26 post-iteration
37 Thread 26->Num Primes:1
38 //start post-join
39 Thread 23->Total Num Primes:2
40 //test output
41 Test Result:100.0% 30.0,30.0.
```

Fig. 9. Traces Showing Correct Concurrency at Work

Traces are even more useful if they can be used to manually or automatically identify mistakes. Figures 10 and 11 identify potential mistakes identified automatically.

In Fig. 10 (which also has cropped parts), the trace has two semantic problems. First, the execution of the threads is serialized in the order of their thread number, thereby avoiding the synchronization problems that arise in combining their results, which the test program is expected to overcome. Second, the load is imbalanced - each thread except Thread 24 performs one iteration (lines 24-27, Fig. 10), with Thread 24 performing four (lines 3-15, Fig, 10). Both mistakes are pointed out by running the test code (lines 26-39, Fig. 10). The test run also indicates all aspects of the test program that are correct (lines 30-35, Fig. 10). Thus, it helps pinpoint the problems with the code. Based on the requirements correctly and incorrectly met, a score of 80% is assigned.

The trace syntax is correct in Fig. 10. The trace in Fig. 11 (again partly cropped) has two different syntax errors. First, the pre-fork property is named "Randoms" (line 1, Fig. 11) rather than "Random Numbers", as indicated by the first error message (lines 23-25, Fig. 11). In addition, the fork output (consisting of iteration and post-iteration output) does not match the 25 regular expressions expected for 7 random numbers (3 iteration outputs for each of the 7 random numbers plus 1 post-iteration output for each of the 4 threads) – due to a loop error it has only 17 outputs. This error is pointed out in lines 28-32 Fig. 11. Because of these syntax errors,

no semantic checks are run, and the tested program earns a score of 10% (line 21, Fig. 11).

```
1  Thread 23->Random Numbers:
2  [509, 578, 796, 129, 272, 594, 714]
3  Thread 24->Index:0
4  Thread 24->Number:509
5  Thread 24->Is Prime:true
6  Thread 24->Index:1
7  Thread 24->Number:578
8  Thread 24->Is Prime:false
9  Thread 24->Index:2
10 Thread 24->Number:796
11 Thread 24->Is Prime:false
12 Thread 24->Index:3
13 Thread 24->Number:129
14 Thread 24->Is Prime:false
15 Thread 24->Num Primes:1
16 Thread 25->Index:4
23 Thread 26->Num Primes:0
24 Thread 27->Index:6
25 Thread 27->Number:714
26 Thread 27->Is Prime:false
27 Thread 27->Num Primes:0
28 Thread 23->Total Num Primes:1
29 Test Result:80.0%,24.0,30.0
30 Pre fork output correct
31 Post fork output correct
32 Post join output correct
33 Number of forked threads correct
34 Pre fork events correct
35 Correct number of iterations
36 Imbalanced thread load:
37 Max thread iterations:4 -
38 min thread iterations = 3
39 No interleaving during fork
40 Post join events correct
```

Fig. 10. Trace with Semantic Errors

```
1  Thread 24->Randoms:
2  928, 85, 384, 616,704, 658, 77]
3  Thread 25->Index:0
4  Thread 25->Number:928
5  Thread 27->Index:6
6  Thread 27->Number:77
19 Thread 26->Num Primes:0
20 Thread 24->Total Num Primes:0
21 Test Result:10.0%,3.0,30.0
22 Event tests will not be run until output fixed
23 Pre fork output did not completely match the 1
24 regular expressions in:
25 [.*Thread.*->Random Numbers:.*\[.*\].*]
26 Post fork output did not completely match the 25
27 regular expressions in:
28 [.*Thread.*->Index:.*\d.*.*,
29  .*Thread.*->Number:.*\d.*.*,
30  .*Thread.*->Is Prime:.*(true|false).*,
31  // remaining expressions omitted
32  ...]
33 Post join syntax correct
```

Fig. 11. Trace with Syntax Errors

Not all examples will have properties and semantic checks associated with each of the fork-join phases. This is shown in Fig. 12(a), which has the complete test code for a Java version of the C++ fork-join Hello World example of Fig. 1. Recall that in this example, the root thread forks a single worker thread to print the greeting. There is no pre-fork or post-fork output. The iteration and post-iteration steps are not distinguished during the fork as there is no loop - only one step. No data are manipulated by the program, so there is no property to be printed. The exact text printed for the greeting does not matter, so there is no semantics check.

There are only two checks overall that need to be performed – a single thread was forked and it printed something. Hence, the test program consists of three parameter methods, giving the name of the test program (lines 4-6, Fig. 12(a))), the number of expected threads (lines 7-9, Fig. 12(a))), and the credit allocated to forking exactly one thread (lines 10-12, Fig. 12(a)). The infrastructure-provided defaults for allocating credit do not work, hence the overridden parameter method. `threadCountCredit`. It allocates 80% for having the right number of threads, leaving the other 20% for creating one or more threads. Fig. 12(b) shows the result of running this test on a program in which the root thread prints the greeting directly, without forking any worker thread. The exact problem is identified in an error message (line 3).

The test code for the primes problem (see appendix) is larger and more complicated than the one for the Hello World program for two reasons: First, it checks both serial and concurrency semantics. Second, it checks all intermediate results. A version of the test code that checks only concurrency correctness would require only three parameter methods: (a) One to specify the name of the tested program; (b) another to specify the two arguments of the program; and (c) the final one to specify the number of threads.

As with performance testing, we see in function testing that the testing program is concerned with only the "what" of traces – (a) the arguments of the program, (b) the names and types of the properties to be printed in each phase, (c) the relationships among the property values, (d) the total number of iterations, (d) the number of threads to be forked to perform these iterations, and (e) optionally, the partial credit associated with the syntax and semantics of each phase.

```
1  @MaxValue(10)
2  public class HelloForkJoinTest extends
3    AbstractForkJoinChecker {
4      protected String mainClassIdentifier() {
5          return "HelloForkJoin";
6      }
7      protected int numExpectedForkedThreads()
8          return 1;
9      }
10     protected double threadCountCredit () {
11         return 0.8;
12     }
13 }
```

(a) Hello World Test Program

```
1  Hello Concurrent World
2  Test Result:0.0%,0.0,10.0
3  Num threads created 0 != num expected threads 1
```

(b) Serial Program Test Results

Fig. 12. Concurrent Hello World Test

The infrastructure is responsible for the "how" of trace processing – (a) invoking the program, (b) collecting the traces, (c) checking the syntax and semantics of each phase based on the test program specifications, (d) checking that the correct number of threads were forked and their load was balanced, (e) checking that their prints were interleaved, (f) allocating default credit to each independent aspect of the trace, and (g) outputting error messages.

4.4 Infrastructure Layers

Both the function and performance testing abstract classes in our infrastructure use a common program-execution layer to run a program with specified input and arguments, and collect its output. In our previous work on testing of concurrent animations, we built, above this program-execution layer, a higher event-database layer. This layer (a) observed all events announced by observable objects in the tested program, (b) stored each event in a database along with the Thread object that announced it, and (c) allowed a testing

program to determine how many threads made the announcements (during a selected event range), and whether the announcements of these threads were interleaved.

Our implementation for the fork-join model leverages the event-database layer. The testing abstract classes replace the predefined Java `System.out` (used to print output on the console) with a custom observable object that (a) asks the predefined Java object to print objects when such printing is enabled, and (b) converts the print to an event observable by the database. The function testing abstract class retrieves the traced output to do syntax checking and the stored print events in the database to do semantic checking, calling overriding methods in the testing programs to do so.

5 Evaluation

As discussed in Section 4 above, our infrastructure meets all of our qualitative requirements listed in Section 2. We evaluate it quantitatively below based on our experience using it to write tests. In particular, we compare the efforts required to check (a) serial and concurrency requirements, and (b) final and intermediate results

We used this infrastructure to write programs for testing concurrency code written by fourteen participants in a test-based session in an NSF-supported workshop on training faculty members in concurrency [4]. The session involved three problems. One of them was a slight simplification of the primes example here in which a fixed rather than variable number of threads were used to find primes in a list with a variable number of random numbers. The second computed the value of PI concurrently using the Monte Carlo method, again using a fixed number of threads and a variable number of iterations. These two exercises, given to the attendees, were designed by the author's collaborators at Tennessee Tech University. To demonstrate Java concurrency primitives, the author developed a worked example that used a fixed number of threads to concurrently find odd numbers in a list with a variable number of random numbers. For functional correctness, the total number of iterations performed by all threads together is small (we used 27 in our workshop examples), allowing the tests to finish quickly.

Table 1 compares the relative effort required to write testing code that checks serial and concurrency requirements. It performs the comparison based on the number of lines in the test programs after comments and imports were removed from them. The numbers in parentheses give the number of lines of code that had to be written to check intermediate results. Thus, the entry 78 (14) in the top-left cell indicates that 78 lines of code were written to check serial requirements and 14 of them were used to pinpoint problems with intermediate results that would be computed in both the serial and concurrency cases. The PI computation is such that the only way to check final serial correctness is to check the correctness of intermediate serial results – hence 0 lines of code are assigned to serial intermediate.

If intermediate results related to concurrency are *not* to be checked, only three lines of code need to be written – to specify the number of expected forked threads (Fig. 12(a), lines 7-9). Even when intermediate results are checked, the code required for concurrency checks is small in absolute and relative terms. This is a strong case for traces, as, without them, functional concurrency correctness and serial intermediate results could not be checked. With them, our infrastructure requires very few lines of code to be written, as the specification of the concurrency requirements is small compared to the specification of serial ones.

TABLE I. TEST CODE SIZE

Problem	Serial (Intermediate)	Concurrency (Intermediate)
Odd	78 (14)	25 (22)
Prime	86 (14)	25 (22)
PI	95 (0)	21 (18)

5 Conclusions and Future Work

Our contributions include not only the Java-based infrastructure we have developed for Java procedural programming but also the requirements and key insights we have identified. Future work is required to translate these concepts to other languages such as C and Python and declarative programming such as in C/OpenMP and Java/Pyjama. It would also be useful to automatically generate these traces by instrumenting compiled code, thereby reducing testing requirements students must follow while writing their code. More experience is also needed with our current implementation for training both students and teachers. Tracing additional classes of concurrent programs is another avenue for future work. It would also be useful to incorporate techniques for influencing thread scheduling to catch synchronization bugs [8].

This paper provides a basis for carrying out such work.

ACKNOWLEDGMENTS

This work was funded in part by NSF award OAC 1924059.

REFERENCES

[1] Prasad, S.K., A. Gupta, A. Rosenberg, A. Sussman, and C. Weems. CDER Center | NSF/IEEE-TCPP Curriculum Initiative," NSF/IEEE-TCPP Curriculum Initiative. 2017; Available from: https://grid.cs.gsu.edu/~tcpp/curriculum/?q=node/21183.

[2] 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:. p. 1-8.

[3] Dewan, P., A. Wortas, Z. Liu, S. George, B. Gu, and H. Wang. Automating Testing of Visual Observed Concurrency. in 2021 IEEE/ACM Ninth Workshop on Education for High Performance Computing (EduHPC). 2021. IEEE.

[4] Dewan, P., A. Worley, S. George, F. Yanaga, A. Wortas, J. Juschuk, M. Rogers, and S.K. Ghafoor, Hands-On, Instructor-Light, Checked and Tracked Training of Trainers in Java Fork-Join Abstractions., in HiPCW. 2022, IEEE. p. 28-35.

[5] Williams, A., Chapter 1. Hello, world of concurrency in C++!, in C++ Concurrency in Action 2022, Manning (https://livebook.manning.com/book/c-plus-plus-concurrency-in-action/chapter-1/1).

[6] Akbar, B. OpenMP | Hello World program. 2023; Available from: https://www.geeksforgeeks.org/openmp-hello-world-program/#.

[7] Ricken, M. and R. Cartwright. Test-First Java Concurrency for the Classroom. in Proceedings of ACM SIGCSE. 2010.

[8] Musuvathi, M., S. Qadeer, and T. Ball, Chess: A systematic testing tool for concurrent software. 2007, Technical Report MSR-TR-2007-149.

ARTIFACT DESCRIPTION (AD) APPENDIX

This commented test code example explains how the described artifact - our testing infrastructure - works and should be used. Some of the lines have been reformatted from the original code and some, as described below, have been omitted. Thus, a count of the lines here will not correspond to the count reported in Table I.

```
package gradingTools.javaThreads.primes.execution;
// imports and annotations omitted
MaxValue(40)
public class PrimesFunctionality extends AbstractForkJoinChecker {
    // Start tested-program invocation data
    final String TESTED_CLASS_NAME = "ConcurrentPrimeNumbers";
    final int NUM_THREADS = 4;
    final int NUM_RANDOMS = 7;
    // End tested-program invocation data
    // Start tested-program invocation overridden methods
    protected String mainClassIdentifier() { return TESTED_CLASS_NAME ;}
    protected int totalIterations() {return NUM_RANDOMS; }// one iteration for each random number
    protected int numExpectedForkedThreads() {return NUM_THREADS;} }// used for concurrency correctness
    protected String[] args() {
      return new String[] {
          Integer.toString(totalIterations()),Integer.toString(numExpectedForkedThreads())};
    }
    // End tested-program invocation overridden methods
    // Start syntax data
    // The public constants are exported to test programs so they can use them in printProperty calls
    public static final String RANDOM_NUMBERS = "Random Numbers";
    public static final String INDEX = "Index";
    public static final String NUMBER = "Number";
    public static final String IS_PRIME = "Is Prime";
    public static final String NUM_PRIMES = "Num Primes";
    public static final String TOTAL_NUM_PRIMES = "Total Num Primes";

    final Object[][] PRE_FORK_PROPERTIES = {    {RANDOM_NUMBERS, Array.class}};
    final Object[][] ITERATION_PROPERTIES = {
            {INDEX, Number.class},
            {NUMBER, Number.class},
            {IS_PRIME, Boolean.class}
    };
    // End syntax data
    final Object[][] POST_ITERATION_PROPERTIES = {{NUM_PRIMES, Number.class},};
    final Object[][] POST_JOIN_PROPERTIES = {{TOTAL_NUM_PRIMES, Number.class},};
    // End syntac data
    // Start syntax overridden methods
    protected Object[][] preForkPropertyNamesAndType() {
       return PRE_FORK_PROPERTIES;
    }
    protected Object[][] iterationPropertyNamesAndType() {return ITERATION_PROPERTIES;}
    protected Object[][] postIterationPropertyNamesAndType() {return POST_ITERATION_PROPERTIES;}
    protected Object[][] postJoinPropertyNamesAndType() {return POST_JOIN_PROPERTIES;}
    // End syntax overridden methods

    // Start semantics data
    Object[] randomNumbers;
    int numPrimesInRandomNumbers, numPrimesFoundByCurrentThread; sumPrimesFoundByAllThreads;
    // End semantics data
    // Start semantics overridden methods
    // Each of these semantics methods should return null if there is no error, otherwise
    // it should return an error message printed by the infrastructure.
    /**
     * This is the first semantic method invoked. The first argument is the root thread and the Map
     * the properties output by it before forking.  */
```

```
    protected  String preForkEventsMessage(Thread aThread, Map<String, Object> aNameValuePairs) {
        randomNumbers = (Object[]) aNameValuePairs.get(RANDOM_NUMBERS);
        return null;
    }
    /** This method is invoked as each iteration of a thread is processed */
    protected  String iterationEventsMessage(Thread aThread, Map<String, Object> aNameValuePairs) {
        int anIndex = (int) aNameValuePairs.get(INDEX);
        int aNumber = (int) aNameValuePairs.get(NUMBER);
        int anExpectedNumber = (int) randomNumbers[anIndex];
        if (aNumber != anExpectedNumber) {
            return "Number " + aNumber + " output at index " + anIndex + "!= expected number " +
               anExpectedNumber;
        }
        boolean isPrime = (boolean) aNameValuePairs.get(IS_PRIME);
        if (isPrime) {
            numPrimesFoundByCurrentThread++;
        }
        // check if the number is actually a prime
        boolean isActualPrime = isPrime(aNumber); // implementation of isPrime not given
        if (isPrime != isActualPrime) {
            return "Is Prime output as " + isPrime + " for number " + aNumber + " but should be " +
               isActualPrime;
        }
        return null;
    }
    /**
     * This method is invoked after all iteration of a thread have been processed, and before the iteration
     * properties of the next thread have been processed. Threads are ordered arbitrarily. Even though the
     * property outputs are expected to be interleaved, the testing infrastructure does not interleave the
     * execution of the two iteration, allowing the testing code to finish processing all properties
     * output by a thread, before it processes those output by another thread
     */
    protected String postIterationEventsMessage(Thread aThread, Map<String, Object> aNameValuePairs) {
        int aNumNumbersComputed = (int) aNameValuePairs.get(NUM_PRIMES);
        if (aNumNumbersComputed != numPrimesFoundByCurrentThread) {
          return "Thread " + aThread.getId() + " found " + numPrimesFoundByCurrentThread +
               " but computed " +  aNumNumbersComputed;
        }
        sumPrimesFoundByAllThreads += aNumNumbersComputed;
        numPrimesFoundByCurrentThread = 0;// reset the variable for the next thread
        return null;
    }
    protected  String postJoinEventsMessage(Thread aThread, Map<String, Object> aNameValuePairs) {
        int aComputedFinalNumbers = (int) aNameValuePairs.get(TOTAL_NUM_PRIMES);
        if (aComputedFinalNumbers != sumPrimesFoundByAllThreads) {
            return "Num primes with dispatching thread " + aComputedFinalNumbers + " != " +
                       "sum of primes found by each thread " + sumPrimesFoundByAllThreads;
        }
        int aNumActualPrimes = 0;
        for (Object aRandom:randomNumbers) {
            if (isPrime((int) aRandom)) aNumActualPrimes++;
        }
        if (aComputedFinalNumbers != aNumActualPrimes) {
            return "Num computed primes " + aComputedFinalNumbers + " != " + "actual primes " +
                       aNumActualPrimes;
        }
        return null;
    }
    // End semantic checking methods
    // Implementation of isPrime() omitted
}
```