An Experimental Evaluation of Tools for Grading Concurrent Programming Exercises*

Manuel Barros[0009−0006−7855−5235], Maria Ramos[0009−0007−4256−5295], Alexandre Gomes[0009−0005−6131−7647], Alcino Cunha[0000−0002−2714−8027], José Pereira[0000−0002−3341−9217], and Paulo Sérgio Almeida[0000−0001−7000−0485]

INESC TEC & U. Minho
{manuel.q.barros,maria.j.ramos}@inesctec.pt, pg46950@alunos.uminho.pt,
{alcino,jop,psa}@di.uminho.pt

Abstract. Automatic grading based on unit tests is a key feature of massive open online courses (MOOC) on programming, as it allows instant feedback to students and enables courses to scale up. This technique works well for sequential programs, by checking outputs against a sample of inputs, but unfortunately it is not adequate for detecting races and deadlocks, which precludes its use for concurrent programming, a key subject in parallel and distributed computing courses. In this paper we provide a hands-on evaluation of verification and testing tools for concurrent programs, collecting a precise set of requirements, and describing to what extent they can or can not be used for this purpose. Our conclusion is that automatic grading of concurrent programming exercises remains an open challenge.

Keywords: Concurrent programming · Testing · Verification · e-Learning.

1 Introduction

Learning concurrent programming is hard. For students coming from a sequential programming background, writing programs with multiple threads that share data is confusing and difficult to get right. Concurrent programming gives rise to new classes of bugs, such as deadlocks [24] and race conditions [34].

When learning to program, students are often presented with a system such as *Codeboard* [1], with a variety of coding exercises, accompanied by unit tests. Students can submit solutions and find out how they perform based on the number of tests passed. Unit testing proves to be a valuable technique for autograding simple sequential programs. It lets students experiment and get instant feedback, exposing problems and guiding them to correct solutions.

Unlike for sequential code, identifying incorrect concurrent code is not trivial. Even for sequential programs testing is non-exhaustive, but it is relatively easy to define a set of unit tests that almost always detects incorrect solutions. However,

* This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.

unit tests are much less reliable when checking concurrent programs, where errors are often subtle and may manifest into visible bugs only after many program executions. This non-determinism may lead students to falsely assume that their programs are correct when tests show no anomalies.

Given the benefits of autograders for learning sequential programming, we would like to deploy similar learning strategies for concurrent programming. For that purpose, we would need tools that automatically detect common concurrency anomalies, like race conditions and deadlocks, and provide accurate and helpful feedback. As unit testing is generally inappropriate for concurrent code, we turn to more sophisticated techniques, such as automatic race detectors.

The main goal of this work is to evaluate existing tools that perform automatic analysis of multi-threaded Java code. Focusing on Java, instead of a more verification friendly pseudo-code specification language such as Promela [25] or PlusCal [29], enables a systems approach that considers the interaction of concurrency with all the complications of the language and platform. For instance, reference aliasing and exceptions in the Java language can contribute to races (by exposing supposedly encapsulated state) and deadlocks (by preventing locks from being released), respectively.

In particular, we want to evaluate the possibility of using these tools in an educational context, to automatically grade and give prompt feedback on student submissions to concurrent programming exercises regarding races and deadlocks. Although, being race and deadlock free is not the same as being optimal, as solutions might, for instance, overly restrict concurrency to meet tests. The possibility that an inadequate solution is accepted by automatic testing also exists in sequential programming exercises but can be mitigated by judicious problem statements. The same care should be taken when devising concurrency problems.

The rest of this paper is structured as follows. Section 2 establishes the scope of this experiment by describing the requirements for automatic grading. Section 3 introduces a set of typical concurrent programming problems and corresponding solutions containing various errors. Then, Section 4 introduces currently available tools that can be repurposed for automatically grading these solutions. Section 5 presents the results of the experiments and Section 6 uses them to compare the tools. Finally, Section 7 concludes the paper by summarizing the main lessons learned and proposing future work.

2 Scope

Since we are interested in tools for an educational setting, our requirements are different than those regarding tools suitable for production settings. Our main use case is finding concurrency bugs in submissions to proposed exercises. Normally, students are given a specification in the form of some interface methods and are asked to write thread-safe Java classes that implement them. Our analysis targets will be small, self-contained programs, with just a few Java classes.

We are not interested in tools that require thoughtfully annotating or modifying parts of the code. The analysis of hundreds of submissions needs to be

automatic and require at most a one-time set-up, such as writing a script or defining some sort of specification. Tools that require the tester to manually analyze each submission fall outside of our scope, but tools that require simple annotations can be considered, as long as annotating code can be automated.

We will take into consideration the rates of false-positives/negatives. A false-negative is a result where a tool does not report a bug when it exists and a false-positive is when a tool reports a bug that does not exist. When automatic analyzers are being used to give students feedback on potential concurrency problems, it is important to have a small rate of false-negatives, so students are not mislead into thinking that there is nothing wrong with their code, which could cause them to develop bad programming habits. But when grading students assignments, we are mostly concerned with the false-positive rate, as we cannot afford to falsely classify a program as incorrect when a student grade depends on it.

For the purpose of this evaluation, we are interested in two types of bugs: data races and deadlocks. A data race occurs when two threads access the same data item without using some synchronization mechanism, one access being a write. A deadlock corresponds to a situation where every member of a set of threads is waiting for some action to be taken by another thread of the same set. In most deadlock scenarios, a thread is waiting for a signal to be sent or a lock to be released. In this situation, every thread is prevented from making progress. It is important to note that data races and deadlocks are a subset of the possible concurrency anomalies. There are even subtler concurrency bugs that silently (with no crash or stall) violate the intended semantics of a program, but we choose to focus on these since they often occur when first learning concurrent programming.

3 Dataset

Our dataset for evaluating analysis tools mainly consists of two collections of concurrent programs. The first one contains different implementations of a thread-safe `Bank` class, mostly written by students, with operations such as fetching the balance of an account or transferring money between accounts. The other one is made up of different implementations of a `BoundedBuffer`. Each program was manually revised and labeled based on which bugs (deadlocks and/or data races) were present. For each of the two datasets we created tests that exercised the different methods of the supposedly thread-safe classes under workloads of multiple threads. This was necessary for the evaluation of the dynamic checking tools. Both collections of programs are available in an online repository.[1]

Additionally, we use the `JBench` [23] benchmark to evaluate data race detection tools. It consists of a collection of programs containing data races, specifically curated for evaluating data race detection tools. Although `JBench` will be part of our dataset, we will mostly use the `Bank` and `BoundedBuffer` for qualitatively evaluating the tools we explore, since these examples have been carefully

[1] https://github.com/mj-ramos/FORTE2023

```
class Bank {
    private static class Account {
        private int balance;
        private ReentrantLock account_lock;

        int balance();
        boolean deposit(int value);
        boolean withdraw(int value);
    }

    private Map<Integer, Account> accounts;
    private int nextId;
    private ReentrantLock bank_lock;

    int createAccount(int balance);
    int closeAccount(int id);
    int balance(int id);
    boolean deposit(int id, int value);
    boolean withdraw(int id, int value);
    boolean transfer(int from, int to, int value);
    int totalBalance(int[] ids);
}
```

Fig. 1. Bank class variables and methods.

revised and better resemble the kind of concurrent programming exercises presented to students. Also, it is easier to manually evaluate each tool by running it on a collection of small programs sharing the same interface and whose semantics we understand clearly. `JBench` will mostly serve as an extra benchmark to assess how each tool performs when ran with no special configuration against an existing set of programs known to contain data races.

3.1 The `Bank` dataset

Part of our dataset is composed of submissions to an exercise that asked students to implement a `Bank` class that manages a set of client accounts, as shown in Figure 1. This problem statement is aimed at applying multiple locks and the two-phase locking protocol when traversing and modifying collections, a key learning outcome for concurrent programming. It is useful also to demonstrate the usefulness and applicability of different locking primitives.

A correct implementation should be thread-safe: it should work correctly when used by multiple threads concurrently. A correct implementation should allow concurrency, but the result of running multiple concurrent operations must correspond to the outcome of some sequential execution. This implies, for example, that a thread cannot observe the state of the bank in the middle of a transfer.

Besides the simplistic solution that uses a single lock and overly restricts concurrency, the assignment then suggests two different implementations. The first one uses `ReentrantLock` instances for protecting accesses to each account and accesses to the collection of accounts. Alternatively, `ReentrantReadWriteLock` can be used to synchronize accesses to the collection of accounts (the main point of contention), allowing more concurrency. Writing such a thread-safe class, that maximizes concurrency by applying two-phase locking and using `ReentrantReadWriteLock`, is not trivial. There is a lot of room for introducing concurrency bugs.

In addition to bugs caused by unprotected accesses to the collection of accounts, or to the accounts themselves, there is a particularly subtle class of problems that can lead to deadlocks. It corresponds to the scenario where two threads attempt to acquire the same two locks but in reverse orders. This may cause the two threads to block indefinitely waiting for each other's lock to be released. This type of bug can arise in implementations that use `ReentrantReadWriteLock` to protect accesses to the collection, particularly in methods involving multiple accounts (`transfer` or `totalBalance`).

In total, the `Bank` dataset is composed of seventeen different implementations. Eight of them contain data races and five contain deadlocks. Ten of them are actual student submissions. The remaining seven variants were inspired by typical student errors collected over many years of teaching concurrent programming.

3.2 The `BoundedBuffer` dataset

The *bounded buffer* is a classical problem in concurrent programming and a construct that can be used as a building block for inter-thread synchronization and communication. It is thus widely available in various forms, such as operating system *pipes* and in concurrent programming libraries. It is also a good example of how synchronization primitives such as semaphores or monitors can be used.

The `BoundedBuffer` dataset was developed specifically for the purpose of this study based on well-known pitfalls and the experience of authors teaching concurrent programming. The first step was writing a correct implementation, as shown in Figure 2, using a single `ReentrantLock` and two `Conditions` for concurrency control.

The two main methods are `pop` and `push`. The first one is responsible for removing an element from the array. The lock is acquired and an element is removed if the array is not empty, signaling a producer that is waiting for space to be freed and freeing the lock. If the array is empty, then the consumer will free the lock and wait for a producer to add an element to the array. As soon as the consumer is signaled by a producer, it acquires the lock, checking again if the array is empty and repeating the process. The `push` method works in the same way, but for the addition of an item instead.

The second step was identifying possible ways in which bugs could arise. For example, data races can occur when the array is accessed outside the scope of a lock. There are also ways in which deadlocks can arise, such as when using a single `Condition` and `signal`, or signaling only if the buffer is empty or full.

```
class BoundedBuffer<T> {
    private final int max;
    private List<T> queue = new ArrayList<>();
    private Lock lock = new ReentrantLock();
    private Condition notFull = lock.newCondition();
    private Condition notEmpty = lock.newCondition();

    public BoundedBuffer(int size) { this.max = size; }

    public T pop() throws InterruptedException {
        try {
            lock.lock();
            while (queue.isEmpty()) notEmpty.await();
            notFull.signal();
            return queue.remove(0);
        } finally {
            lock.unlock();
        }
    }

    public void push(T value) throws InterruptedException {
        try {
            lock.lock();
            while (queue.size()>=max) notFull.await();
            notEmpty.signal();
            queue.add(value);
        } finally {
            lock.unlock();
        }
    }
}
```

Fig. 2. Correct implementation of the Bounded Buffer class.

Not retesting predicates (i.e., using `if` instead of `while`) can lead to exceptions, either due to spurious wake-ups (which may or may not happen), or the usage of `signalAll`. Table 1 shows different ways in which a bounded buffer implementation can be wrong.

The third step was to create multiple permutations of the identified possible mistakes, with eleven classes being created using this method. Two other correct classes were created by changing the way in which the condition in the `push` and `pop` methods is checked and the way the threads are signaled. Together with the original class, we ended up with a total of fourteen classes in our dataset.

Table 1. Concurrency bugs in a Bounded Buffer implementation.

Method	Buggy code	Outcome
`pop`	remove after lock release	data race, deadlock
`pop`	remove before lock acquisition	data race
`push`	insert after lock release	data race, deadlock
`push`	insert before lock acquisition	data race
`pop`, `push`	only one `Condition` and `signal`	deadlock
`pop`, `push`	`signal` only if empty/full	deadlock
`pop`, `push`	test with `if` (instead of `while`)	exception
`pop`, `push`	test with `if` and `signalAll`	exception

3.3 JBench

JBench is a collection of multi-threaded programs specifically targeted at benchmarking data race detectors. It is composed of several small programs and some real-world applications, each with a number of known data races. Every program in this repository has been manually verified by its creators, and every data race has been properly documented.

The types of programs contained in JBench vary in size and nature. It contains 6 real-world examples, consisting of actual applications, and 42 small, academic programs. Since our targets are small programs with few classes, we excluded the real-world programs from our dataset. Among the remaining programs, there are some whose format better resembles the kind of programs students are asked to develop (thread-safe implementations of some interface) and others that mainly consist of a main method that launches several threads that exercise some sequence of statements that possibly leads to data races.

4 Tools

Our method for finding tools for automatic analysis of concurrent programs mainly consisted in scanning through scientific papers related to this topic. As a starting point, we analyzed a survey of tools [32], to find potential candidates for testing. Additionally, most papers we analysed were found either by searching through Google Scholar, using terms like "deadlock detection" and "data race detection", or by following references in relevant works.

Our experience shows that, despite the large amount of published work on automatic analysis of concurrent programs, there are not many publicly available tools with active support that can easily be downloaded and ran on current Java code. Many articles describe tools or algorithms for detecting concurrency anomalies but do not present a publicly available artifact [30,15,18,39,20,22,38,42,31,17,37]. In other cases, where a link for the tool was provided, the tool was no longer available [3,11,35,12,40,27,19]. Some tools were developed for now deprecated Java versions [28,33,21] and others do not work with the package `java.util.concurrent` [8,5].

```
src/RacyDict.java:20: warning: Thread Safety Violation
  Read/Write race. Non-private method ‘RacyDict.get(...)‘
      reads without synchronization from container ‘this.dict
      ‘ via call to ‘Map.get(...)‘. Potentially races with
      write in method ‘RacyDict.put(...)‘.
 Reporting because another access to the same memory occurs
     on a background thread, although this access may not.
  18.
  19.        public String get(String key) {
  20. >          return dict.get(key);
  21.        }
  22.    }

Found 1 issue
                          Issue Type(ISSUED_TYPE_ID): #
  Thread Safety Violation(THREAD_SAFETY_VIOLATION): 1
```

Fig. 3. RacerD output report example.

Excluding all the tools that we were unable to run for the reasons mentioned above resulted in a set of four tools: Infer, Java Pathfinder, RV-Predict, and MultithreadedTC. We believe these four tools are representative of the current panorama of mature tools for data race and deadlock detection for Java programs.

4.1 Infer

Facebook's Infer [16] is a static analysis tool for Java, C, and Objective-C, able to find several types of bugs in a program. For this evaluation, we explored its ability to detect data races and deadlocks.

Infer checks Java programs for the existence of data races using a static race detector called RacerD [14]. RacerD looks for potential data races between non-private methods of a Java class when run in parallel with one another. RacerD starts from non-private methods but will recursively analyze each method call it encounters. Given a program, RacerD will run this analysis for classes annotated with `@ThreadSafe` and classes that it can infer are intended to run in a concurrent context (classes that use locks, for example).

The generated reports provide a detailed description of the data races found, intended to give a clear understanding of their location, as well as the original method calls that gave rise to each bug. For example, when ran against a supposedly thread-safe class called `RacyDict`, containing a `get` method that reads from a `Map` without synchronization, RacerD outputs the report on Figure 3.

RacerD was designed to be used at Facebook. It is suited for large code bases that evolve rapidly over time. Its design favours speed, scalability, low friction (meaning it demands little effort from the tester) and effective signaling

(meaning it only reports high-confidence bugs, aiming at reducing false-positives and always providing meaningful reports). The detection of some data races is purposely compromised in order to achieve these goals. As such, false-negatives are possible.

Infer is also able to analyze a program in search of starvation issues. One of the issues it tries to find is the possibility of deadlocks. To trigger this analysis, Infer must be run with the `-starvation` flag. It will report a deadlock if there is the possibility of two threads attempting to acquire two locks in reverse orders.

4.2 Java Pathfinder

Java Pathfinder (JPF) core [6] is a Java Virtual Machine that executes the system under test (SUT) checking for properties such as unhandled exceptions, deadlocks and user-defined assertions. JPF is a model checker, therefore it does not simply test the SUT. Instead, it explores all potential execution paths, identifying points in programs which represent execution choices, from which the program could proceed differently. Execution choices can be due to different scheduling sequences or random values, and JPF allows the user to control which choices are explored if some of them are known to be uninteresting.

One of the main problems of model checking is the state space explosion. In order to mitigate this problem, JPF uses Partial Order Reduction (POR) to minimize context switches between threads that do not result in interesting new program states. This is done without prior analysis or annotation of the program, only by examining which instructions can have inter-thread effects.

To extract information from JPF while exploring all execution paths, JPF uses event-driven programming, through listeners for handling events. In particular, to allow data race detection, we need to extend JPF with a listener called `PreciseRaceDetector`.

One important aspect of this tool is the fact that it is highly extensible. Besides allowing users to create their own listeners or choice generators, is is possible to define publishers, to produce different output formats, or bytecode factories, to provide alternative execution semantics of bytecode instructions. In our evaluation, we only extended it with simple listeners.

4.3 RV-Predict

RV-Predict [10] is a dynamic data race detector that is both sound and maximal. Dynamic means that it executes the program in order to extract an execution trace to analyze. Sound means that it only reports races that are real (no false-positives). And maximal means that it finds all the races that can be found by any other sound race detector analyzing the same execution trace.

RV-Predict works by analyzing a sequentially consistent execution trace and changing the order of events to create a new set of traces, which are then analyzed in order to find traces that are consistent and manifest a data race [26,41].

4.4 MultithreadedTC

MultithreadedTC [36] is a framework that allows exercising specific interleavings of threads in Java applications. Threads are scheduled unpredictably by operating systems, making failures in concurrent applications non-deterministic, since they might not occur every time the application is run. The idea behind this framework is to make it possible to have deterministic and reproducible tests for concurrent code, despite some critical interleavings being hard to exercise because of the presence of blocking and timing.

To coordinate threads, MultithreadedTC uses a clock that runs in a separate thread. The clock advances when all threads are blocked and at least one is waiting for a tick. This simple mechanism makes it possible to delay operations within a thread without using functions like `Thread.sleep()`, which make the test timing dependent. These delays are crucial to define specific interleavings and allow the detection of deadlocks. When threads are blocked and none of them are waiting for a tick or a timeout, the test is declared to be in deadlock.

One of the main features of MultithreadedTC is its integration with JUnit [9]. JUnit assertions can be used to verify, for example, the current clock tick.

To fully utilize the potential of MultithreadedTC to verify a concurrent program, the source code needs to be changed. The task of specifying concrete thread interleavings requires threads to be programmed as individual methods, like `void thread1() {...}` and extra code to be written that defines the desired interleaving. This is not suitable for our case study, since we want the testing to be automated. Automation may be possible to achieve, but given the variety of ways in which students can program, adding lines of code to define interleavings might not be that simple.

It is possible to use MultithreadedTC without defining interleavings, by simply running multiple tests systematically and, through JUnit, define assertions for the desired results. However, this type of testing gives no guarantees about the results – they are non-deterministic. There is also the possibility of those assertions failing for reasons other than concurrency anomalies. Since every test will give different results, we decided to exclude MultithreadedTC from the quantitative results presented in the next section.

5 Results

Tables 2 and 3 present the quantitative results obtained by running each tool against our dataset, excluding MultithreadedTC, as mentioned above. It is important to note that no tool ever reported a false-positive. Since every program without concurrency bugs was always correctly labeled as bug-free, results only highlight the ratio of files reported to have bugs over the actual number of programs with bugs. Table 4 shows the average run time of each tool for each dataset.

We used Infer v1.1.0, RV-Predict v2.1.3, MultithreadedTC v1.01 with JUnit v4.13.2, and the latest version of JPF from GitHub [7]. MultithreadedTC was tested with jdk-11 and the rest of them with jdk-8.

Table 2. Ratio of programs with data races reported over programs with data races. Elements marked with * mean that some data races might not have been caught because of other problems in the programs.

	Tool		
Dataset	Infer	Pathfinder	RV-Predict
Bank	5/8	5*/8	8/8
Bounded buffer	9/9	0/9	9/9
JBench	15/30	30*/42	26*/42

Table 3. Ratio of programs with deadlocks reported over programs with deadlocks. Elements marked with * mean that some deadlocks might not have been caught because of other problems in the programs.

	Tool		
Dataset	Infer	Pathfinder	RV-Predict
Bank	0/5	4*/5	N/A
Bounded buffer	0/10	9/10	N/A

5.1 Infer

In the `Bank` dataset, Infer's RacerD misses three of the faulty programs, revealing two limitations of RacerD. The first limitation, already known [14], caused RacerD to miss races in two programs, caused by wrong usage of `ReentrantReadWriteLock`: `closeAccount` acquires a read lock instead of a write lock, even though it updates the map. The other missed race reveals another limitation of RacerD: it fails to detect races caused by unsynchronized accesses to the `balance` field of an `Account`. Further testing revealed that RacerD is unable to detect data races on fields of objects stored in a container in the class being tested.

Since Infer depends on a working compilation command to work, for testing RacerD we reduced the JBench dataset to only those examples with a valid compilation command. This excludes 12 from the 42 small programs in JBench. From the remaining 30 examples, RacerD detects data races in 15 of them.

It is worth mentioning that RacerD is fast. It takes around 600 ms per `Bank` example (each of these examples has around 200 lines of code), and around 160 ms per `BoundedBuffer` example (each example has around 40 lines of code).

Infer's documentation mentions that it can detect deadlocks when run with the `starvation` flag activated. However, it did not detect any of the deadlocks present in our dataset. For the `Bank` in particular, it failed to detect any problems in examples containing the following issues:

Table 4. Average run time in seconds for each tool and each dataset. For JPF we also show the number of examples for which the analysis was interrupted after the chosen 4m time limit.

	Tool		
Dataset	Infer	Pathfinder	RV-Predict
Bank	0.6	95.9 (2)	4.6
Bounded buffer	0.2	240.0 (14)	9.8
JBench	1.3	87.2 (14)	9.3

```
public int totalBalance(int[] ids) {
    int total = 0;
    Account1[] acs =new Account1[ids.length];
    rlock.lock();
    try {
        for (int i : ids) {
            Account1 c = map.get(i);
            if (c == null)
                return 0;
            acs[i]=c;
        }
        ...
}
```

Fig. 4. Common error in the `Bank` example

1. Methods that can cause two concurrent threads to try to acquire the same two locks in reverse order.
2. Methods that do not call `unlock()` on a previously acquired `Lock` object.

5.2 Java Pathfinder

In many tests, JPF was not able to detect data races or deadlocks because the programs had other bugs, causing exceptions to be thrown. This happened frequently in the JBench examples, especially in the more complex ones.

There is a very common error in the bank examples, in the `totalBalance` method, where many students confuse array indexes with account identifiers, as shown in the code fragment in Figure 4. If the array `ids` is `[2,3,4]`, an index out of bounds exception is thrown at line 10. JPF reports this exception and stops the execution, as expected. After correcting this error, JPF successfully found the deadlock related to the out-of-order acquisition of the locks. This is why the tables above may mislead the reader into thinking that JPF is not capable of detecting some concurrency anomalies. For the deadlock cases in the bank

samples, we corrected the code that was interfering with the deadlock detection, and all deadlocks were found.

Regarding data race detection, even with examples that were correctly implemented, apart from the concurrency problems, JPF could not find all the anomalies. One interesting outcome is that JPF never explicitly reported a data race in the bounded buffer examples, although in many examples it is possible to see the exception of index out of bounds being thrown, that, in the case of the bounded buffer, is potentially caused by data races.

JPF found all deadlocks but one: the deadlock that was already explained in Section 3.2 and is related to the use of `signal()` instead of `signalAll()`.

It is important to mention that, in many cases, JPF takes a long time to analyze the code, and we decided to impose a time limit of 4 minutes. In nine of the JBench samples the analysis timed out. The test class used for the `Bounded Buffer` with the RV-predict tool could not be used with JPF, because it cannot handle many threads, even with the partial order reduction enabled. For this reason, in almost all of the tests in this example, JPF timed out, even in those using only four threads.

5.3 RV-Predict

RV-Predict is exclusively a data race detector, and therefore not applicable to deadlock detection. In theory, RV-Predict does not report any false-positives, thus only the examples where data races could occur were tested.

RV-Predict was able to detect all of the races in `Bank` and `Bounded Buffer` datasets and being rather quick too. In the JBench dataset, RV-Predict did not detect races in some of the more complex programs: increasing the window of detection, that determines the largest distance between events where RV-Predict is going to try and find races, would make it possible to detect more races, but the prediction phase would take much longer to finish. In some examples RV-Predict could not find any trace that executed correctly – in these cases RV-Predict did not detect any races. When RV-Predict finishes the prediction phase, it will always run one trace of the execution. If a deadlock is encountered, or an exception is thrown, then the execution of the trace will not finish correctly, but if a log directory is specified, all of the races encountered will be logged.

6 Discussion

Infer is easy to use and requires a minimal set-up to make it work. Although its `starvation` analysis failed to detect any deadlock in our dataset, its race detection component, RacerD, showed several desirable qualities for our use case, namely:

– No need to manually inspect and modify the code under test. In some cases, some classes might need to be marked with the `ThreadSafe` annotation, but that can be automated for multiple files.

- Can easily be used to automatically check several programs and generate an aggregated report with the bugs found.
- Its reports provide detailed and understandable explanations regarding the source of the detected races (this is a really desirable feature for a tool meant to be used by students).
- Absence of false-positives, meaning that it could be safely used in automatic evaluation.
- The analysis is fast, which makes the tool viable to use on a large number of examples at once.

Another important feature is that Infer is already present in some e-Learning platforms, namely *Codeboard*, and can be used out of the box to provide automatic feedback on solutions to concurrent programming exercises.

RacerD shines when it comes to ease of use, but there were some data races it was unable to identify. RacerD highlights how our evaluation criteria might differ from those one would use to choose a tool to employ in an industrial setting. This tool sacrifices soundness for scalability. Since our use case involves small programs, we would rather have a less efficient tool, but that does not have false-negatives.

MultithreadedTC is an interesting tool for testing concurrent code, but it has some downsides. For instance, there is no specific mechanism to detect data races. What can be done to hopefully find data races is to define some assertions that express what are the expected results and run the code multiple times so that possibily many interleavings are exercised. One of the problems with this approach is the false-negatives because it is possible that the scenario which leads to wrong results does not come up. In fact, even for deadlock testing, when following this strategy, this can happen. Another downside is the lack of information given when a deadlock is found.

One important aspect of this framework is the fact that the testing methodology follows a *white box approach* [13]. This means that one needs to know the internal design of the application to be able to do more sophisticated tests. To define a specific interleaving, it is sometimes necessary to add lines of code inside a method. This can be seen as a downside since it is essential to have knowledge of how classes are coded, which is particularly undesirable in our use case of testing students' submissions. It is possible to use this tool without the methods that allow the definition of specific interleavings, as we did in some tests we performed, but this is not using the tool to its full potential and has the problem of non-determinism in the results, as described above.

Configuring JPF was not an easy task. Also, in early experiments, it never detected any deadlocks. Later, with the help of another user [2], we were able to find out that JPF was not really adapted to work with `ReentrantReadWriteLocks`. Luckily, someone already had encountered this problem and we managed to fix it by modifying some of the source code of JPF [4].

One downside of JPF is the amount of output generated when the property that allows finding multiple errors (`search.multiple_errors`) is enabled, since all the paths of execution that exhibit a particular data race will be reported as

an error. At the end of the execution, the output can have thousands of lines pointing the occurrence of the same data race. Another aspect related to the output is the poor information provided regarding detected deadlocks. In some cases, where the same example had two different deadlocks, it was difficult to understand if both deadlocks were being reported. For this reason, we had to perform tests separately to understand if a specific deadlock was detected.

Although JPF presented good results in detecting deadlocks, testing was extremely slow, even considering simple programs with a reduced number of threads. This makes it somewhat unsuitable for the purpose of our study, which is to find a tool that students can use to test their programs. It is not reasonable to expect that a student will wait more than 20 minutes for the result of a test of an implementation of a simple exercise.

RV-Predict shows great promise. It is very easy to install and use, and only requires writing a main test program. This main program does not even need to be complicated, in fact, the more simplistic the better since RV-Predict will take care of finding the traces where a data race could occur and still be fast. The only downside of RV-Predict is testing complex programs. A program where multiple methods need to be tested for data races means that the prediction phase will take much longer to complete since multiple combinations of states need to be tested. This could be circumvented by dividing the main test into smaller tests, which will accelerate the prediction process but, in turn, less data races might be detected.

7 Conclusions

The main goal of this work was to evaluate existing tools for automatic detection of concurrency bugs in Java programs, to determine their potential for being used in automatic grading of concurrency programming exercises. We focused on analysing tools for detecting data races and deadlocks.

One of our main takeaways is that, despite the amount of academic research on automatic analysis of concurrent code, there are not many mature and readily available tools for detecting concurrency problems in Java programs. In particular, we found no tool that excelled in detecting both data races and deadlocks.

Regarding data race detection, we were most impressed by Infer's RacerD and RV-Predict. Infer does static analysis, while presenting no false-positives and reporting errors in a comprehensible and educational fashion. However, it is unable to catch some problems that might arise in students' code. RV-Predict requires constructing a small testing program to exercise the code under evaluation (since it is does dynamic analysis) but performed very well with our dataset. We believe that RV-Predict is a good fit for a tool to be used in an educational setting for automatic detection of data races in students' code.

As for deadlock detection, we struggled to find a tool that completely satisfied our criteria. We had the most success with Java Pathfinder. It found most of the deadlocks in our dataset, however its run time makes it unfit to be used to evaluate hundreds of students' submissions. We can thus conclude that the sub-

ject of automatic deadlock detection for small Java programs is still a promising field of investigation.

One interesting future step would be to conduct a more precise analysis of the results obtained with each tool. For each type of bug (data race or deadlock) and for each program in our dataset, we only tested if a given tool labelled that program as erroneous or bug-free. This leaves room for undetected false-negatives (in case the analysis tool detects only a portion of the existing bugs) as well as false-positives (in case the tool reports more bugs than there really exist). Conducting such analysis is not trivial, since the way static tools and dynamic tools report errors is fundamentally different (for example, a single race reported by RacerD might be exercised more than once in some test method used by a dynamic race detector like RV-Predict, which would cause it to report more than one race).

References

1. Codeboard, https://codeboard.io/, acessed on 2023-04-30
2. Deadlock not being detected, https://groups.google.com/g/java-pathfinder/c/rzkaeuNDZCY, accessed on 2023-01-04
3. Dl-Check, https://github.com/devexperts/dlcheck, accessed on 2023-01-02
4. Google groups thread: java.lang.error: java.lang.nosuchfieldexception: tid, https://groups.google.com/g/java-pathfinder/c/t1n73xdyrFI, accessed on 2023-04-29
5. JaDA, http://jada.cs.unibo.it/demo.html, accessed on 2023-04-30
6. Java Pathfinder, https://github.com/javapathfinder/jpf-core, accessed on 2023-04-02
7. Java Pathfinder (master branch), https://github.com/javapathfinder/jpf-core/tree/45a4450cd0bd1193df5419f7c9d9b89807d00db6, accessed on 2023-01-04
8. JCarder, http://www.jcarder.org/download.html, accessed on 2023-04-30
9. JUnit, https://junit.org/junit5/, accessed on 2023-04-30
10. RV-Predict, https://runtimeverification.com/predict/, accessed on 2023-04-30
11. ThreadSafe, http://www.contemplateltd.com/, accessed on 2022-12-27
12. Visual Threads, http://www.unix.digital.com/visualthreads/index.html, accessed on 2023-04-30
13. White box testing techniques, tools and advantages – a quick guide (2022), https://www.xenonstack.com/insights/what-is-white-box-testing, accessed on 2023-01-03
14. Blackshear, S., Gorogiannis, N., O'Hearn, P., Sergey, I.: RacerD: compositional static race detection. Proceedings of the ACM Conference on Programming Languages **2**(OOPSLA), 1–28 (2018). https://doi.org/10.1145/3276514
15. Cai, Y., Wu, S., Chan, W.: ConLock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In: Proceedings of the 36th International Conference on Software Engineering. pp. 491–502. ACM (2014). https://doi.org/10.1145/2568225.2568312
16. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proceedings of the 7th International Symposium on NASA Formal Methods. LNCS, vol. 9058, pp. 3–11. Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1

17. Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: A Race-Aware Java Runtime. Commun. ACM **53**, 85–92 (11 2010). https://doi.org/10.1145/1839676.1839698
18. Engler, D., Ashcraft, K.: RacerX: Effective, static detection of race conditions and deadlocks. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles. pp. 237–252. ACM (2003). https://doi.org/10.1145/945445.945468
19. Erickson, J., Musuvathi, M., Burckhardt, S., Olynyk, K.: Effective Data-Race Detection for the Kernel. pp. 151–162 (01 2010)
20. Eslamimehr, M., Palsberg, J.: Sherlock: Scalable deadlock detection for concurrent programs. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. p. 353–365. FSE 2014, Association for Computing Machinery (2014). https://doi.org/10.1145/2635868.2635918
21. Flanagan, C., Leino, K., Lillibridge, M., Nelson, G., Saxe, J., Stata, R.: Extended static checking for Java. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation. p. 234–245. ACM (2002). https://doi.org/10.1145/512529.512558
22. Flanagan, C., Freund, S.N.: FastTrack: Efficient and Precise Dynamic Race Detection. SIGPLAN Not. **44**(6), 121–133 (jun 2009). https://doi.org/10.1145/1543135.1542490
23. Gao, J., Yang, X., Jiang, Y., Liu, H., Ying, W., Zhang, X.: Jbench: A dataset of data races for concurrency testing. In: Proceedinsg of the 15th IEEE/ACM 15th International Conference on Mining Software Repositories. pp. 6–9. ACM (2018). https://doi.org/10.1145/3196398.3196451
24. Holt, R.: Some deadlock properties of computer systems. ACM Comput. Surv. **4**(3), 179–196 (1972). https://doi.org/10.1145/356603.356607
25. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2011)
26. Huang, J., Meredith, P., Roşu, G.: Maximal sound predictive race detection with control flow abstraction. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 337–348. ACM (2014). https://doi.org/10.1145/2666356.2594315
27. Huang, J., Zhang, C.: Persuasive prediction of concurrency access anomalies. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. p. 144–154. ISSTA '11, Association for Computing Machinery (2011). https://doi.org/10.1145/2001420.2001438
28. Joshi, P., Naik, M., Park, C., Sen, K.: CalFuzzer: An extensible active testing framework for concurrent programs. In: Proceedings of the 21st International Conference on Computer Aided Verification. LNCS, vol. 5643, pp. 675–681. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_54
29. Lamport, L.: The PlusCal algorithm language. In: Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing. LNCS, vol. 5684, pp. 36–60. Springer (2009). https://doi.org/10.1007/978-3-642-03466-4_2
30. Luo, Q., Zhang, S., Zhao, J., Hu, M.: A lightweight and portable approach to making concurrent failures reproducible. In: Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering. LNCS, vol. 6013, pp. 323–337. Springer (2010). https://doi.org/10.1007/978-3-642-12029-9_23
31. Marino, D., Musuvathi, M., Narayanasamy, S.: LiteRace: Effective sampling for lightweight data-race detection. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 134–143. PLDI '09, Association for Computing Machinery (2009). https://doi.org/10.1145/1542476.1542491

32. Melo, S., Souza, S., Silva, R., Souza, P.: Concurrent software testing in practice: A catalog of tools. In: Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation. pp. 31–40. ACM (2015). https://doi.org/10.1145/2804322.2804328
33. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for java. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 308–319. ACM (2006). https://doi.org/10.1145/1133981.1134018
34. Netzer, R., Miller, B.: What are race conditions? Some issues and formalizations. ACM Letters on Programming Languages and Systems **1**(1), 74–88 (1992). https://doi.org/10.1145/130616.130623
35. Nir-Buchbinder, Y., Ur, S.: ConTest listeners: a concurrency-oriented infrastructure for Java test and heal tools. In: Pezzè, M. (ed.) Proceedings of the 4th International Workshop on Software Quality Assurance, SOQUA 2007, in conjunction with the 6th ESEC/FSE joint meeting. pp. 9–16. ACM (2007). https://doi.org/10.1145/1295074.1295077
36. Pugh, W., Ayewah, N.: Unit testing concurrent software. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering. p. 513–516. ASE '07, Association for Computing Machinery (2007). https://doi.org/10.1145/1321631.1321722
37. Said, M., Wang, C., Yang, Z., Sakallah, K.: Generating Data Race Witnesses by an SMT-based Analysis. vol. 6617 (04 2011). https://doi.org/10.1007/978-3-642-20398-5_23
38. Samak, M., Ramanathan, M.K.: Trace driven dynamic deadlock detection and reproduction. SIGPLAN Not. **49**(8), 29–42 (feb 2014). https://doi.org/10.1145/2692916.2555262
39. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. **15**(4), 391–411 (1997). https://doi.org/10.1145/265924.265927
40. Sen, K., Agha, G.: A race-detection and flipping algorithm for automated testing of multi-threaded programs. In: Bin, E., Ziv, A., Ur, S. (eds.) Hardware and Software, Verification and Testing. pp. 166–182. Springer Berlin Heidelberg (2007)
41. Şerbănuţă, T., Chen, F., Roşu, G.: Maximal causal models for sequentially consistent systems. In: Proceedings of the 3rd International Conference on Runtime Verification. LNCS, vol. 7687, pp. 136–150. Springer (2013). https://doi.org/10.1007/978-3-642-35632-2_16
42. Zhai, K., Xu, B., Chan, W., Tse, T.: CARISMA: A context-sensitive approach to race-condition sample-instance selection for multithreaded applications. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. p. 221–231. ACM (2012). https://doi.org/10.1145/2338965.2336780