Deadlocks as Runtime Exceptions

Rafael Brandao Lobo and Fernando Castor

Center of Informatics, Federal
University of Pernambuco, Brazil
rbl,castor@cin.ufpe.br
`http://www.springer.com/lncs`

Abstract. ...

Keywords: ...

2

Introduction

There are two main types of deadlocks: resources deadlocks and communication deadlocks.

Resource deadlocks: a set of threads is holding some resources and is waiting for the resources which have already been held by other threads in the set. Communication deadlocks: some threads wait for some messages or signals from other threads which are paused and unable to send the required messages/signals or have already sent them before a waiting thread starts to wait for it.

In this work we will focus on resource deadlocks where locks are resources.

. . .

3

Bug Reports Study

In a previous study [1], some interesting pattern about concurrency bugs was found: 30 out ot 31 deadlock cases involved at most two resources, where only one case was triggered by three threads waiting for three resources circularly. After that we also suspected that deadlocks between 2 threads and 2 resources are more common than other resource deadlocks, so we've decided to investigate this observation further with a bigger sample of real-world deadlock bug reports from open-source projects.

In this chapter, we'll present how we have conducted our bug reports study. We can split it in three parts: how we've collected the sample, how we've classified each bug in that sample, and what results we have found.

1 Sample Collection

We've chosen three open source projects which used Java as main programming language and made use of concurrent programming: Lucene, Eclipse and OpenJDK.

Lucene[1] is a text search engine library that can be used along many applications, where concurrent programming was used to deliver high performance. Eclipse[2] is one of the most popular IDE for java developers. OpenJDK[3] is an open-source implementation of the Java Platform. These three projects share a few similarities: they're written in Java; they have vast bug report repositories and tools to search for bug reports; development culture of reviews inside bug reports by discussing solutions to fix the problem. In particular, this last point is very important since we need to analyse bug reports and infer their classification.

We have initially searched in each repository for the keyword *deadlock*, and we've collected 541 bug reports in total. Each project had a different bug repository, so we've changed slightly the query parameters to find relevant bug reports.

In Lucene, we've searched for bugs matching the word "deadlock" anywhere in the bug report (i.e. in summary or in comments), in module "lucene-core" with issue type as "bug", where status was "closed". From this search, we've found 27 bugs[4].

[1] Lucene: http://lucene.apache.org/

[2] Eclipse: https://eclipse.org/

[3] OpenJDK: http://openjdk.java.net/

[4] http://goo.gl/DhVI3t

In Eclipse, we've searched for "deadlock" in summary, where resolution was set to "fixed" and status was set to "resolved". From this search, we've found 406 bugs[5].

In OpenJDK, we've searched the word "deadlock" in summary, in module "JDK" with issue type as "bug", where resolution was set to "fixed", status was "resolved". From this search, we've found 108 bugs[6].

Assuming normal distribution in bug reports population, we've calculated the minimal size of the sample where we could achieve 95% of confidence level and 5% sampling error (also setting response distribution to 50% to find the biggest sample size[7]). We've found minimal size as 225, so we've created a random sample of 225 bugs out of all bugs we've found. In appendix we provide the code we've used to calculate the sample size.

2 Data Labeling

We've merged all bug reports of our random sample in one single table. The first field was the name of the bug, each name was composed by a prefix that could be either *LUCENE*, *ECLIPSE* or *JDK*, followed by the bug number inside its own repository. Then we've added a category label field which could be "A", "B", "C" or "D". The category was added after manual inspections based on criterias that we will present next. Furthermore we've added other fields in this table such as *"TYPE"*, *" of THREADS"*, *" of RESOURCES"* and *"NOTES"*, they'll be described next.

The final version of this table[8] also contains *"TIME (HOURS)"* and *"COMMENTS"*, but these were extracted automatically from the bug reports data and we did not use them for this analysis.

2.1 Field "CATEGORY"

This is one of the most important fields, as we want to be able to identify what kind of deadlock this bug represents, or if it's not even a real deadlock. We have four different values for this field, and they must be one of the following:

A: We are confident this a resource deadlock. We should be able to provide a short explanation of how the bug occurs, which or how many threads are involved and how many locks are involved in this bug.

B: We are confident this is not a resource deadlock, so it must be a communication deadlock. It might be a lost notify/signal bug. We should be able to identify if this is a lost notify/signal or have clear evidence this is not a resource deadlock (adding a note whenever possible).

C: We are confident this is a false-positive for "deadlock" search. The term was used as a synonym of "hang" or "infinite loop", or to refer to another

[5] http://goo.gl/qQnrEm

[6] http://goo.gl/xYFfsO

[7] More about 'response distribution' in http://www.raosoft.com/samplesize.html

[8] http://goo.gl/zNsIGz

deadlock bug. In some cases, it is possible that a bug refers to another bug which was fixing a deadlock, so the initial bug may not be deadlock-related and just fix a regression for another bug (which could be deadlock-related). In other words, this is not a deadlock bug at all.

D: We are not confident whether this is a resource deadlock or a communication deadlock, or even if this is a false-positive for deadlock. There's not enough information in the bug report, or the information is just inconclusive. Since we are not experts on any of these repositories, it's hard to classify for sure in another category.

Category A will be only assigned when there's a clear comment in the bug explaining what threads and which resources are involved or other evidences can clarify without doubt how many threads and lock resources are involved. In a few cases, the explanation is not fully clear but the attachment provides a clear thread dump showing which threads are involved and which locks each one is holding and waiting for, so we can also use this information to make a final decision.

Category B can be classified by also looking into source code changes when we are almost clear about its category: if the patch changes areas of the code where a notifyAll is added or moved, then it is most likely a category B indeed. Sometimes it is just a semantic deadlock where one threads is in an infinite loop waiting for others to finish and other threads are stuck waiting to acquire a lock the first thread already acquired; in this case, we also understand as a communication deadlock: the "message" which the first thread have been waiting is whether the other threads have finished.

Category C is often easy to classify since the bug often explains another kind of bug and then cites the term "deadlock" as a synonym for "hang". As stated previously, if this bug only refers to another bug (such as a regression) that mentions deadlock or fixes a deadlock, then this bug might not be a deadlock by itself, just a fix of another previous fix, which would also fall into this category.

Category D is for all other bugs which could not be classified as either A, B or C.

Reviewing Protocol: In order to minimize error on our classification, we've created a protocol that every reviewer should follow, which basically describes how data should be analysed for a certain bug. For example, sometimes a bug points to another one as a duplicate, those links should be used if the initial bug is not clear enough. In order to organize how the review is executed, we should roughly follow these steps:

1. Look at bug title and bug main description (usually the first comment). Sometimes the reporter have an idea of how the bug occurs and which threads are involved, so this is a big help.

2. Look at further comments and see if someone understood this bug completely. Someone must have provided a reasonable explanation of how this bug occurs. If the category is already clear, then finish these steps; otherwise proceed.

3. If available, look at the patches (specially the final patch) and what changes have been made. If uncertain about this bug being in category B and the patch either moves or adds a notifyAll call, then it most likely is a category B bug. If this is not the case, then proceed.

4. If available, look at the related bugs or duplicates. It's often to find an initial bug that is unclear but which points out to a duplicate that have been largely discussed and is clear. Restart from step 1 for each of those related bugs. If a category was not assigned yet, then proceed.

5. See other attachments if available, like text files with thread dumps or stack traces. If they provide enough information to clarify which category it is, then assign a category to it, otherwise proceed.

6. Classify this bug in the category D.

2.2 Fields "# of THREADS" and "# of RESOURCES"

Whenever possible, the reviewer should state the number of threads and resources involved, even if this is in the category B. If it's unknown how many resources but it is clear how many threads are involved, then only one of them should be filled and the other field should remain blank.

2.3 Field "TYPE"

This field is just an annotation and it should be used to specify what kinds of resources a certain bug use. For example if there are two threads and they're in a circular deadlock, then this field should be locks/synchronized, or if you are sure that explicit locks were used for both, then just locks is enough, or if only synchronized blocks/methods are involved, then just synchronized.

The symbol + indicates a separation between threads, so for example "locks + wait" means that one thread holds a lock and the other waits". As this may be confusing, an easy replacement would be to use the "notes" field instead and write down what was found about this bug.

2.4 Field "NOTES"

This field was encouraged to be used specially to remind other reviewers in the future of how the conclusion was made for cases where it was tricky to choose the category.

3 Results Analysis

As we want to understand how many resource deadlock bugs did involve 2 threads and 2 resources, we discard bugs in B and C category because they're not resource deadlocks. What we have left are the bugs we could not determine its category. In the worse case scenario, all bugs in category D should be resource deadlocks which would involve something different than 2 threads and 2 resources, given by the following equation:

$$bugs\_ratio = \frac{bugs(A, threads = 2, resources = 2)}{bugs(A) + bugs(D)} \ . \tag{1}$$

In that equation, *bugs(...)* returns the number of bug reports that matches the parameters. Thus *bugs_ratio* represents the worse case scenario. However if we want to look at the best case scenario, then all bugs classified in D category must also be classic resource deadlocks.

Table 1. Bug Reports Classification.

Category	Number of Bugs
A	101
B	32
C	23
D	69

The numbers we've found are 54.7% in the worse case and 95.29% on the best case. In category A, we've found 93 bugs of classic deadlocks. That means **from all resource deadlocks we've found, 92.07% of them were classic deadlocks**, which corroborates with the finding of study conducted by Lu, Shan, et al [1]. Also, **75.93% of deadlock bugs were classified as resource deadlocks** which shows how popular resource deadlocks are in comparison to communication deadlocks.

We can also look at bugs in category D differently and assume that their distribution will follow the proportions of bugs we've found for A, B and C. We can do the same for bugs inside category A.

When we reclassify all bugs in D in other categories, we'll have the following: 45 new bugs in A where 41 bugs should also be classic deadlocks; 14 new bugs in B; and 10 new bugs in C. We can see the updated values in the following table.

Table 2. Bug Reports Categories Proportionally Distributed.

Category	Number of Bugs
A	146
B	46
C	33

$$bugs\_ratio = \frac{[bugs(A, threads = 2, resources = 2) = 134]}{[bugs(A) = 146] + [bugs(D) = 0]} \ . \tag{2}$$

Running the same equation again with the new values, **we estimate that 91.7% of resource deadlocks should be classic deadlocks if we could predict bug categories in D**.

4 Findings

As we've highlighted in the previous section, classic deadlocks are by far the most popular type of resource deadlocks; also, resource deadlocks are more popular than communication deadlocks. This gives us evidence that if we can solve the problem of deadlock detection for the classical case, that is, between 2 threads and 2 locks, we can cover most of the bugs.

We believe that giving developers a signal that something is wrong in the code (i.e. an exception) is much more powerful than showing nothing (as it happens today). Exceptions provide an easy framework to reason about potential issues in the code and makes easier to handle bugs once they were detected. And even if the bug is not handled, it still gives a signal to developers that something is wrong and should be fixed.

In the next chapter, we will present our deadlock detection algorithm we've built that will throw an exception when a classical deadlock happens. Following that chapter, we will show an experiment we did with students to test whether these exceptions are indeed helpful to find problems in the code.

4

Protocols

In this chapter, we present the deadlock detection algorithm divided in three parts. In the first part, an overview of the protocol is described and we also present proof that this protocol is sufficient to detect deadlocks between 2 threads and 2 locks (in short, we will call it *2-deadlock*). We further change the protocol to guarantee that exception is raised on both threads. Finally we show pseudocode of the actual implementation we developed on this research.

5 Protocol: Deadlock Detection

We have modified the default implementation of Java's *ReentrantLock* to allow efficient runtime 2-deadlock detection. We take advantage of the current algorithm and some of its guarantees to avoid the need to introduce extra synchronization mechanisms or costly atomic operations.

1. Each lock has a pointer for a thread which is the current owner or null when there's no thread owning that lock.
2. Each lock has an integer to represent its current state: 0 means the lock is free and no thread owning it (the *unlocked* state), 1 means there's a thread owning the lock (the *locked* state). For simplicity, we are only interested on these two states and its change holds the most complexity, but in the implementation of *ReentrantLock* each time the thread owner acquires the same lock, this state would be incremented, and decremented each time the thread releases it.
3. Each thread has a thread-local list of pointers of locks they are currently owning.
4. Each lock has a waiting queue of threads that are waiting to acquire it. Whenever a thread try to obtain a lock when it's already acquired, the thread will add itself on the waiting queue before parking. Upon the event of releasing the lock, the owner of that lock will look for the first thread in the waiting queue and unpark it.
5. When a thread wants to acquire a lock, it will swap the current state to *locked* if the current state is *unlocked* atomically.
   (a) If the thread fails, it must be because the lock is already owned by some other thread, then it will add itself on the waiting queue for that lock. Finally, the thread will park.
   (b) Otherwise, the thread will set itself as the current owner of that lock and also add this lock to its thread-local list of pointers of locks it owns.

6. When a thread is about to release a lock, the current owner pointer of that lock is set to null and that lock is also removed from the thread-local list of owned locks. Finally, the lock state is changed to *unlocked*.
7. Before parking, a thread will check whether there's deadlock. When the current thread is unable to acquire its desired lock, it must be because another thread is owning it already. It is possible to know who is the owner of any lock, so the current thread identifies the owner of its desired lock as the conflicting thread. Then the current thread will search on each lock of its thread-local list of owned locks if the conflicting thread is waiting on it.
   (a) If positive, then we have a circular dependency (current thread is stuck waiting its desired lock and the conflicting thread is stuck waiting for a lock the current thread owns) thus a deadlock exception will be raised.
   (b) Otherwise, the thread parks.

5.1 Assumptions

This protocol's correctness relies on a few guarantees provided by Java's *ReentrantLock* class on its default implementation.

1. The operation of swapping the state of a lock from *unlocked* to *locked* must be done atomically by the thread, so only one thread can be successful at a time.
2. A thread will only park when it's guaranteed some other thread can unpark it. Missing notifications will never happen and concurrent uses of park and unpark on the same thread will be resolved gracefully.
3. Inserts on each lock's waiting queue must be done atomically. If multiple threads concurrently attempt to insert themselves in the waiting queue on the same lock, they will both succeed eventually but the exact order of insertions is not important.
4. Once the last element in the waiting queue of a lock is read, it should be safe to read all threads in the waiting queue that arrived before the last element. Since the thread who reads the waiting queues is also the one who blocks every thread waiting on the queues, we can guarantee the only updates that could happen concurrently is new insertions at the end of each queue. However insertions in the end of the queue are not important once a last element pointer is obtained.

6 Formal Proof

On this subsection, we proof this protocol is sufficient to detect 2-deadlocks. First, we show a proof for the *liveness* property which states we can always detect 2-deadlocks when they happen. Lastly, we show a proof for the *safety* property which states we never throw exceptions when 2-deadlocks doesn't really happen.

Lemma 1. *Protocol can always detect deadlock when a 2-deadlock happens.*

Proof. Suppose not and a 2-deadlock occured without deadlock exception being raised. Let's assume that threads A and B have both acquired locks a and b respectively, as follows:

$$write_A(state_a = locked) \to write_A(owner_a = A) \tag{3}$$

$$write_B(state_b = locked) \to write_B(owner_b = B) \tag{4}$$

And now each thread will attempt to acquire the oppositing lock: thread A is trying to acquire lock b and thread B is trying to acquire lock a, as follows:

$$read_A(state_b == locked) \to write_A(waiting\_queue_b.insert(A)) \tag{5}$$

$$read_B(state_a == locked) \to write_B(waiting\_queue_a.insert(B)) \tag{6}$$

If a 2-deadlock happened, then both threads are now parked and all previous equations should be correct. But before parking, each thread must check for deadlock by inspecting each lock it owns if the oppositing thread is on its waiting queue. As we initially assumed no deadlock exception has been raised, then both threads are parked and also the following equations must be correct:

$$read_A(owner_b == B) \to read_A(waiting\_queue_a.contains(B) == false) \tag{7}$$

$$read_B(owner_a == A) \to read_B(waiting\_queue_b.contains(A) == false) \tag{8}$$

The problem with the previous equations is that they both cannot be true simultaneously. Before checking for deadlock, each thread must add itself on the waiting queue of its desired lock. If it holds that the oppositing thread is not in the waiting queue yet, then it must be because it did not start to check for deadlock yet, thus a contradiction.

Lemma 2. *Protocol never throw a deadlock exception for a non-existent 2-deadlock.*

Proof. Suppose the opposite: a deadlock exception was raised and there's no real 2-deadlock. At least one of the following equations must be true in order to raise a deadlock exception:

$$read_A(owner_b == B) \to read_A(waiting\_queue_a.contains(B) == true) \tag{9}$$

$$read_B(owner_a == A) \to read_B(waiting\_queue_b.contains(A) == true) \tag{10}$$

Suppose without loss of generality the first equation is correct. It means thread B is waiting for lock a and it is also the owner of lock b. If it is on the waiting queue, that thread is either parked already or about to park and in both cases it means thread B is going to depend on the release of lock a to proceed. However, as we have seem previously, thread A at this point is also about to park and is checking for a deadlock. If this condition holds, we have a circular dependency between threads A and B, a real 2-deadlock, thus we have a contradiction.

The only problem with this protocol is the lack of guarantee that both threads involved in a 2-deadlock will throw deadlock exception. If both threads are about to park and are both running the deadlock detection procedure, then the equations 7 and 8 will both be true and deadlock exception will be raised by both threads. However, it is possible that one of the threads did not finish inserting itself on the waiting queue for the lock it desires, then the conflicting thread will hit the case when one of the equations 7 or 8 will be false, thus not throwing a deadlock exception.

7 Protocol: Exception Raised On Both Threads

We have further extended the previous protocol to allow both threads involved in the deadlock to throw deadlock exceptions. This does not affect how deadlock is detected but what should be done after a deadlock is detected.

1. Each lock has a list of tainted threads. This list should only be read or updated by the owner of that lock, allowing immunity from interference without any extra synchronization cost.
2. Once a deadlock is detected and the current thread is about to raise a deadlock exception, it already knows: which thread is conflicting with itself; and which lock that thread is desiring. Then the current thread (the owner of the desired lock) will add this conflicting thread in tainted threads list for that lock. After that, deadlock exception is raised.
3. When the conflicting thread is unparked and finally acquires its desired lock (it becomes the owner of that lock), then it is allowed to read the list of tainted threads. If this thread identifies itself into this list, then it must be because it was part of a deadlock before, so it removes its reference from the list and also raise a deadlock exception.
4. Every operation on the list of tainted threads of any locks (either reading or inserting values) should be followed up by some cleanup on all references of threads that are no longer running.

This is sufficient to force both threads to throw exceptions when only one of them would raise an exception in the initial protocol. However when they both would raise an exception anyway, then this change introduces a different problem: dangling references.

Each thread would have added their conflicting thread on its owned locks's tainted threads list, but none of them would be able to acquire their respectives desired locks (as in *item 3*), thus leaving their references behind for others to cleanup (as in *item 4*).

8 Implementation

In this subsection we present pseudocode for the proposed protocols. We attach in the appendix pseudocode for the current implementation of *ReentrantLock*

which we will not focus here. Instead, we will look into what changes were done on top of that implementation to follow the protocols covered previously. The actual code can be found on our github repository.
Changes on ReentrantLock

```
// This is a thread-local inside a lock.
// Each thread keeps the list of locks they own.
DEFINE_PER_THREAD(vector<int>, ownedLocks);

// As soon as a lock is acquired or release, this function is called.
// Based on that, we call register or unregister owned lock.
void setExclusiveOwner(Thread thread) {
  owner = thread;
  if (owner == null) {
    unregisterOwnedLock();
  } else {
    registerOwnedLock();
  }
}

// These functions register or unregister the current lock
// in the thread-local list ownedLocks.
registerOwnedLock();
unregisterOwnedLock();

void park() {
  Thread conflictingThread = owner;
  if (isAnyOwnedLockDesiredBy(conflictingThread)) {
    clearOwnedLocksByCurrentThread();
    throw new DeadlockException();
  }
  LockSupport.park(this);
}

// Returns true if any of the locks owned by the current thread
// contain a given thread in the waiting queue.
isAnyOwnedLockDesiredBy(Thread);

// Clear all locks in the list of owned locks by the current thread.
clearOwnedLocksByCurrentThread();
```

5

Evaluation

In this paper we have also empirically evaluated how effective deadlock exception can be.

We had two implementations of reentrant locks where one of them was the one provided by Java's *ReentrantLock* and the other was that lock modified to throw exception when a deadlock happened. Later we may refer to our implementation as *LockA* and the default one as *LockB*.

In order to compare each implementation, we conducted a controlled experiment where students had to run two specific programs with deadlocks easy to reproduce while collecting the time taken to identify the problem. They also had to provide a clear explanation of the problem, describing what the problem is, which method calls were involved on it and a description of how it happens, so we could measure answer precision.

9 Experiment Definition

The goal of our experiment was to analyze the process of bug identification with the purpose of evaluating efficiency of deadlock exceptions, in respect to the time spent in order to identify the problem and the accuracy of the descriptions provided by the students. We can define two research questions we want to answer in this experiment:

RQ1. Is the time spent to identify the bug reduced for implementation with deadlock exception when compared to the default implementation?

The metric we watched to answer this question was the time, in seconds, to finish each question in the test.

RQ2. Is the accuracy of bug description improved for implementation with deadlock exception when compared to the default one?

Each question's answer was splitted in a few criterias and each criteria was rated between 0 and 1, where 0 means not present, 0.5 means partially present and 1 for fully present:

A. Correctly classified problem as deadlock.
B. Classified problem as different from deadlock.
C. Correctly identified method calls involved in the deadlock.
D. Correctly identified locks involved in the deadlock.
E. Pointed unrelated methods as part of the deadlock.

To answer this research question, we have classified students answers as either correct or incorrect. Correct answers should respect the following equation:

$$(A - B) + C \geq 1.5 \tag{11}$$

We decided to rule out criterias D and E because the problem statement was not clear they should describe which locks were involved in the deadlock; also, our deadlock implementation at that time could only guarantee at least one deadlock exception to be thrown thus affecting at least one method. In other words, this equation means that a correct answer is whenever the bug was described as deadlock and at least one of the methods involved were identified.

10 Experiment Planning

In order to evaluate each element described on the previous section, we describe the following statistical hypotheses.

10.1 Hypothesis

To answer *RQ1* regarding the time spent to identify a bug in the code:

$$H_0 : \mu_{TimeLockA} \geq \mu_{TimeLockB} \tag{12}$$

$$H_1 : \mu_{TimeLockA} < \mu_{TimeLockB} \tag{13}$$

And to answer *RQ2* regarding accuracy of answers:

$$H_0 : \mu_{CorrectAnswersLockA} \leq \mu_{CorrectAnswersLockB} \tag{14}$$

$$H_1 : \mu_{CorrectAnswersLockA} > \mu_{CorrectAnswersLockB} \tag{15}$$

10.2 Design, Instrumentation and Subjects

For this empirical experiment, we have chosen two metrics: time to answer a question and number of correct answers.

In order to prevent *bias*, we needed to control a few factors during the experiment execution. The first factor was the selection of subjects to participate on this experiment, as different background knowledge could potentially influence chosen metrics. The second factor we had to control was the complexity of programs that each subject. Complexity we define as the amount of files in the program, number of threads and number of locks to analyze; as we've assumed that easier programs could have little or no benefit from deadlock exceptions, we wanted to have one program that we considered easy to identify the problem and another that was more complex and composed by many files and classes, reflecting a more realistic case. We provided implementations of each program using either *LockA* or *LockB*: the two possible treatments that we want to compare.

We decided to use Latin Square Design to control these two factors mentioned earlier: subjects and program complexity factors. Since we had N subjects, 2 programs and 2 possible treatments, we disposed subjects in rows and programs in columns of latin squares, randomly assigning in each cell of the square a treatment that could be *LockA* or *LockB*, but also guaranteeing that for any given row or column in this square, each treatment appears only once (see Table 1). Consequently, we have replication, local control and randomization which are the three principles of experiment design [7].

Table 3. Latin Square design

	Program 1	Program 2
Subject 1	LockA	LockB
Subject 2	LockB	LockA

We wrote two programs with different complexity which were presented in the same order for all subjects. The first program, known as *Bank*, contained 4 classes spread in 4 files, 3 threads, 3 explicit locks, and 82 lines of code in average. The second program, known as *Eclipse* had 15 classes spread in 11 files, 4 threads, 5 explicit locks, and 40 lines of code in average. We expected the first program to be easier to identify the deadlock because it contained fewer classes and files. Each program could use either *LockA* or *LockB* but we randomly assigned a group to each student so that if they fall into group A, they would start with *LockA* in the first question, but change to *LockB* on the second question; or if they fall in group B, they they would start with *LockB* and switch to *LockA* in the second question. We randomly paired subjects in tuples composed of one subject in group A and another subject of group B, then we created latin squares for each one of these pairs, where any remainders were discarded.

We have repeated this experiment for two groups of students with different backgrounds. The first group consisted of undergraduate students attending Programming Language Paradigms course. They had classes about concurrent programming, including exercises in Java using ReentrantLock where deadlocks and other concurrent bugs should be avoided; however, these students were not experienced in this area. The second group consisted of graduate students enrolled in master's degree or PhD program attending Parallel Programming course where they had classes about advanced concepts of parallel programming and had a lot of practical exercises, including implementing their own lock; thus, they were expected to have a lot of experience. We did a survey with the second group to understand their background even further (see charts below) at the end of the experiment.

10.3 Metrics Collection

Each one should start the experiment with the first question containing *Program 1* and once they finish to provide an answer, they should request for the second question. At that point, we collect and place a timestamp in their answer.

Once they finish the second question containing *Program 2*, then they should again give us a notice so we can leave a new timestamp. We have used these timestamps to measure how long they took to finish each question. We have started this experiment with a time limit for each question of 60 minutes each. However, during the test we realized it could not be sufficient for all students so we expanded to 90 minutes each.

The timestamp was written by students conducting the experiment based on a counter we projected on the laboratory wall in real time. In a few circunstances the subject could write the timestamp when they finish, but we have double checked the value at the time we collected their answer, overwriting in case they did any mistake.

11 Experiment Operation

We executed this experiment in two different days. In the first day we did it with undergraduate students in replacement of their default exam, so their participation was obligatory but we disclaimed they could optionally leave a comment if they did not want to take part in this research, so we would not use their data. Fortunately no one chose to not participate. In the second day, we did it with graduate students after the last class of Parallel Programming course and it was optional. In total, 31 students participated on the first day and 16 students participated on the second day, but we had to discard 2 students data because they arrived late and they had to leave early.

On the first day we started with a time frame of 2 hours for the whole experiment, so we decided to set a deadline for each question and put a time limit of 1 hour each. Later we expanded the time limit to 1 hour 30 minutes for each question. On the second day we decided to stick with 1 hour each because there was no demand to extend it.

12 Experiment Results

We can split the experiment analysis in two parts: time and accuracy.

12.1 Time Analysis

Time analysis was conducted with R Statistical Software using the inputs[9] extracted from each experimentation day. We've used the linear model described in Figure 1 that considers the effect of different factors on the response variable similarly to Paola's work [3], but we've also added the effect between each replica and the treatment as explained by Sanchez in [2].

Initially, we've plotted the box-plot graphic shown in Figure 2. We can see that answers with *LockB* involved took more time to complete, but suddenly stop to grow not far from where *LockA* reaches its peak. If there was no time

[9] We provide the inputs we've used in the appendix.

$$Y_{lijk} = \mu + \tau_l + \tau\alpha_{li} + \beta_j + \gamma_k + \tau\gamma_{lk} + \epsilon_{lijk}$$

Y_{lijk} - response of l_{th} replica, i_{th} student, j_{th} program, k_{th} lock
τ_l - effect of l_{th} replica
$\tau\alpha_{li}$ - effect of interaction between l_{th} replica and i_{th} student
β_j - effect of j_{th} program
γ_k - effect of k_{th} lock
$\tau\gamma_{lk}$ - effect of interaction between l_{th} replica and k_{th} lock
ϵ_{lijk} - random error

Fig. 1. Regression model.

limit on each question, we believe that *LockB* times would show a much wider range.

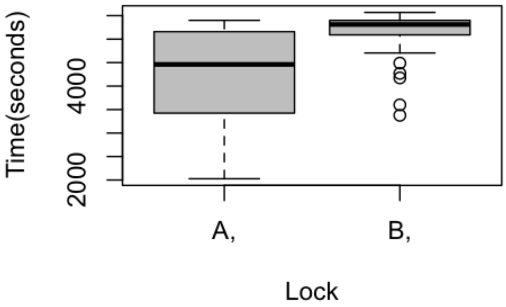

Fig. 2. First experiment box-plot graphic.

Then we've run the Box-Cox transformation - which is a power transformation - to reduce anomalies such as non-addivity and non-normality, obtaining the curve in the left of Figure 3. Since the value of λ at the maximum point in the curve is not approximately 1, we should apply the transformation; that is, Y_{lijk} should be powered to that λ on our regression model. Running the same analysis again with the transformed model, we obtain the curve shown in the right of Figure 3.

After applying Box-Cox transformation, we ran Tukey Test of Additivity that checks whether effect model is additive, so we can evaluate whether interaction between factors displayed on the rows and columns of each latin square won't affect significantly the response when the model is additive [7]; thus, considering the following hypothesis:

H_0 : The model is additive
H_1 : H_0 is $false$

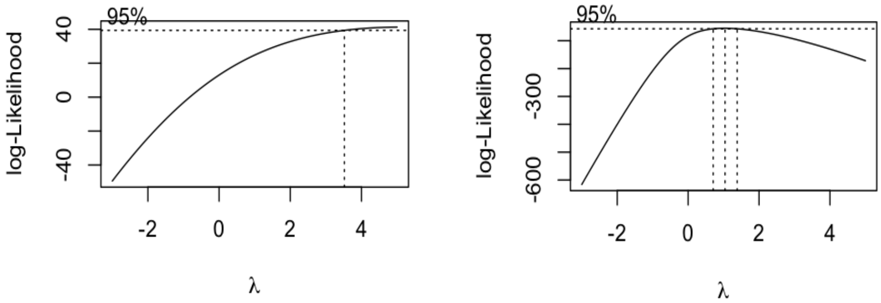

Fig. 3. First experiment: before and after box-cox transformation ($\lambda = 5$).

We have obtained a p-value of 0.514, which means we cannot reject H_0. Consequently the model was considered to be additive.

Finally, we ran the ANOVA (ANalysis Of VAriance) test which compares the effect of treatments on the response variable, providing an approximated p-value for every associated factor. When a variable has *p-value* < 0.05, it means that factor was significant to the response.

Table 4. Undergraduate students experiment ANOVA results.

	Df	Sum Sq	Mean Sq	F value	*p-value*
Replica	14	3.8633e+37	2.7595e+36	1.6553	0.1784197
Program	1	4.1460e+36	4.1460e+36	2.4869	0.1371197
Lock	1	3.9489e+37	3.9489e+37	23.6873	0.0002492 ***
Replica:Student	15	4.1013e+37	2.7342e+36	1.6401	0.1808595
Replica:Lock	14	2.4033e+37	1.7166e+36	1.0297	0.4785520
Residuals	14	2.3340e+37	1.6671e+36		

In Table 4, we can see that *Lock* factor was the most significant to the response, allowing us to reject our null hypothesis defined in Equation 10.

Now we will show the results collected by the second experiment with graduate students. They were exposed to the same set of problems in a different day, but as explained before, they only had a time limit of 1 hour per question.

When we analyze the box-plot for the second group displayed in Image, we can see there was a clear improvement on the time for students with *LockA*.

Moving foward with the analysis, we check if a box-cox transformation is needed. Since the value is not approximately 1, we apply the power transformation the same way we did with the first experiment, but with the corresponding lambda value.

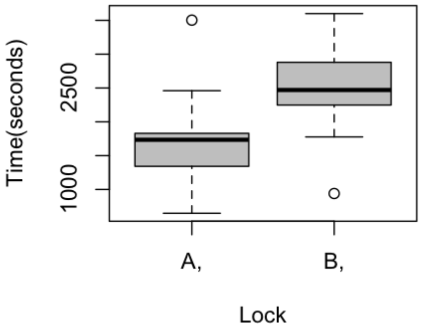

Fig. 4. Second experiment box-plot graphic.

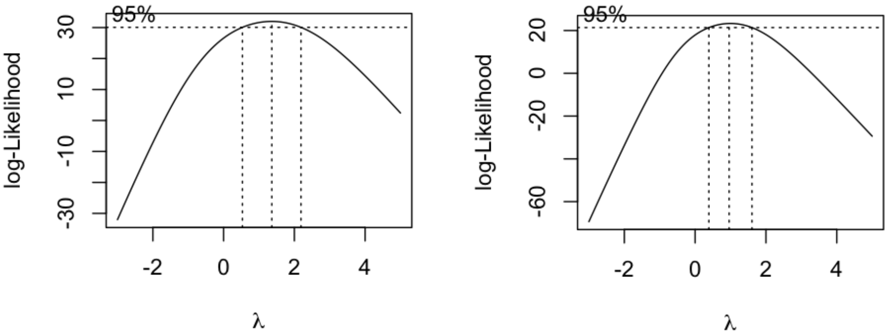

Fig. 5. Before and after box-cox transformation ($\lambda = 1.3636$).

Finally, running ANOVA, we can see that the type of lock was the most significant factor for the response time, as shown in Table 3. Again, we can reject the null hypothesis.

Table 5. Graduate students ANOVA results.

	Df	Sum Sq	Mean Sq	F value	Pr(¿F)
replica	6	2576883250	429480542	14.1891	0.0025793 **
program	1	6875586	6875586	0.2272	0.6505035
lock	1	1958179433	1958179433	64.6938	0.0001975 ***
replica:student	7	2328154077	332593440	10.9881	0.0047601 **
replica:lock	6	823830276	137305046	4.5362	0.0441188 *
Residuals	6	181610625	30268438		

In the end of this paper, we provide the list of inputs and instructions on how to reproduce these results as attachments.

12.2 Accuracy Analysis

Once we collected all answers, we manually evaluated each one of them according to the criterias established previously, where each criteria had an associated value between 0 and 1. Then we've ran a script that evaluates the equation we defined before to classify whether an answer was correct or not. Grouping the results in tables, we have Table 4 and Table 5.

Table 6. Undergraduate students answers accuracy

	Correct	Incorrect
LockA	29	2
LockB	16	15

Table 7. Graduate students answers accuracy

	Correct	Incorrect
LockA	13	1
LockB	10	4

Applying Fisher's exact test we can see that undergraduate students results presented a two-tailed P value equals 0.0004: the association between rows (groups) and columns (outcomes) is considered to be extremely statistically significant; consequently, there is clear evidence of improvement on accuracy (see Table 4).

Meanwhile graduate students results presented a two-tailed P value equals 0.3259, which does not represent a statistically significant evidence of improvement in accuracy (see Table 5).

The input and the script to generate the tables are available as attachments in the end of this paper.

13 Discussion

We can see in our results that both groups of students have improved their time to solve the problem when they had the lock with deadlock exception. Also, on the first group, we have found statistically significant evidence that it improved answers accuracy, but not for the second group.

We cannot draw conclusions regarding the improved accuracy on the second group, but we can bring up some relevant aspects we've observed and make a few hypothesis. Some students in the second group were greatly experienced on concurrent programming and they knew how to efficiently find a deadlock using the tools available in Eclipse. Thus, they have finished the exercise really quickly for both problems, knowing exactly which points in the code were involved in the deadlock. So we can see that deadlock exceptions are more helpful for unexperienced programmers in general, and it's possible that even if we had a bigger sample for the second group, we would still not see a significant difference that would indicate deadlock exceptions improved their accuracy.

However, we believe that the benefits of deadlock exceptions are beyond helping unexperienced programmers to find deadlocks more precisely. Experienced programmers would still benefit in many cases where the deadlock is not as obvious as in the exercise we've presented. For example, in a more realistic situation, a deadlock can happen in a background thread that doesn't really affect the program execution overall but make the execution lacking some expected behavior. Furthermore, in non-interactive systems where they are only running in background, is nearly impossible to know when there's a problem unless this software is monitored constantly which is very time consuming or the system produces output constantly that is affected by a potential deadlock. If we have a deadlock exception, we can either prepare and handle this exception on the code level, or just have this signal from output that would help developers to fix it later.

14 Threats To Internal Validity

In this experiment, we've collected evidence on how the presence of deadlock exception affects student ability to identify deadlocks accurately. However, we must raise a few considerations regarding the validity of our results.

14.1 Time Measurement

Since we wanted to run the experiment in a homogenous environment, we've decided to run it in a laboratory in Federal University of Pernambuco, and we've provided links to download the exercise and a few instructions explaining how to deploy it. We wanted to make it as easy as possible and before we've started the test, we gave a small presentation reproducing step by step the instructions that would be described on each exercise, so everyone could follow up and make the setup at the same time. Once everyone was done, we've started to count the time and allowed them to run the programs and start debugging. However, this procedure was not enough: there was a few students (approximately 3 in total) who did the setup differently and could not execute the program; therefore, they've lost a few minutes until we've fixed that for them. Since they've lost only a few minutes, we have still counted them as part of the experiment and did not discount the time.

Furthermore, some students arrived at the test more than 10 minutes late. We've allowed them to join, but some of the remaining computers in the laboratory had issues like they were not logging in or the mouse was not working. We've lost a few minutes to make them work or find a new computer and once each of them did the setup, we've started to count their time individually.

Whenever a student finished a given question, if the time was below the time limit they had available, we have marked the current timestamp on each student's name in the whiteboard. Each entry inserted was already sorted by time, so we easily tracked whether each student was close to the second question's time limit. It would have been better to do this automatically rather than doing manually, so we could potentially reduce overhead of these timestamp operations and increase their precision.

Also, we believe that our imposed time limit have limited more drastically the time ranges on the first group because they spent more time on each question. Also the fact it was an exam for them may have delayed the time to answer because they were more careful. We have observed during the experiment that many students wrote their answers but they were reluctant to ask for the next question because they still have plenty of time left and they wanted to make sure it was correct. We did not observe such behavior with the second group of students and we believe it is because they did not have the same pressure to deliver correct results as the first group had.

14.2 Exercises

We understand that the two questions we've used to evaluate the students are far easier than what most software engineers have to deal with in the real world. However we could not use any real world issue because it would easily take the time limit of the experiement for each bug.

On the other hand, we've created two questions based on real world bugs that we have found while searching for deadlock bugs in open source repositories. Each question had a particular level of granularity, where one should be easier to find a bug because of the less amount of code to examine and another that should be more difficult because of the reasonable amount of different files to look at.

Some researchers actually believe that empiric evaluations should not be limited to real projects. Buse claims there are benefits of using non-real artifacts [4] because it's easier for researchers to translate research questions into successful experiments as it allows a greater control over confouding factors. Otherwise, it would be necessary to turn all participants familiar with the codebase of a real and complex system before even starting the experiment.

15 Threats to External Validity

Let's consider a few conditions that might limit the generalization power of our findings in this experiment.

15.1 Students

Each student which participated in this experiment had a different background. What we did to minimize the differences was to select groups where students had at least basic experience in concurrent programming and they should be familiarized with the types of bugs such codes can have: the first group of students with undergraduate students attended the class Paradigms of Computaional Languages where deadlocks are covered in classes and exercises; the second group with graduate students attended the class Parallel Programming which covered concurrent programming in low level detail in classes and exercises, including deadlock detection.

Some studies have already addressed the problem of drawing conclusions made with students but some suggest that using students as subjects is as good as using industry professionals [6]. Runes ran an experiment which shows that there's not much significant differences between undergraduate, graduate and industry professionals, with the exception that undergraduate students often take more time to complete the tasks [5].

6

Sample Size Calculation In R

```
sample.size = function(c.lev, margin=.5,
                       c.interval=.05, population) {
  z.val = qnorm(.5+c.lev/200)
  ss = (z.val^2 * margin * (1-margin))/c.interval^2
  p.ss = round((ss/(1 + ((ss-1)/population))), digits=0)
  METHOD = paste("Recommended sample size for a population of ",
                 population, " at a ", c.lev,
                 "% confidence level", sep = "")
  structure(list(Population = population,
                 "Confidence level" = c.lev,
                 "Margin of error" = c.interval,
                 "Response distribution" = margin,
                 "Recommended sample size" = p.ss,
                 method = METHOD),
            class = "power.htest")
}

sample.size(95, 0.5, 0.05, 541)
```

7

Sample Analsysis in Python

```
import sys

f = open(sys.argv[1], "r")
headers = f.readline().split('\t')
lists = [ list() for i in xrange(4) ]

for line in f.readlines():
  data = line.split('\t')
  data = [ i.strip() for i in data ]
  unit = tuple((data[0], data[2], data[3], data[4], data[5], data[6]))
  group = ord(data[1])-ord('A')
  lists[group].append(unit)

total = 0
overall = len(lists[0]) + len(lists[3])
for u in lists[0]:
  if u[2] == u[3] and u[2] == '2':
    total += 1

print '== results =='
print 'found', str(total), 'deadlock bugs with 2 threads and 2 locks'
print 'found', str(overall), 'potential deadlock bugs'
print 'rate (worse case): ', str(float(total)/float(overall)*100), '%'
print 'rate (best case): ', str(float(total+len(lists[3]))/float(overall)*100), '%'
```

8

Fetch Bug Reports Data

```
import random
import sys
import datetime

import json
import urllib2
import os.path

f = open(sys.argv[1], "r")
headers = f.readline().split('\t')

path = './bugs/'
bugs = list()

for line in f.readlines():
  data = line.split('\t')
  data = [ i.strip() for i in data ]
  bugs.append(data[0])
  rep,bug_number = data[0].split('-')

  filepath = os.path.join(path, data[0] + '.xml')
  if os.path.exists(filepath):
    continue
  print 'fetching data for', data[0]
  url = ''
  if rep == 'ECLIPSE':
    url = ("https://bugs.eclipse.org/bugs/show_bug.cgi?" +
           "ctype=xml&id=" + bug_number)
  elif rep == 'JDK':
    url = ('https://bugs.openjdk.java.net/si/jira.issueviews:' +
           'issue-xml/' + data[0] + '/' + data[0] + '.xml')
  elif rep == 'LUCENE':
    url = ('https://issues.apache.org/jira/si/jira.issueviews:' +
           'issue-xml/' + data[0] + '/' + data[0] + '.xml')

  u = urllib2.urlopen(url)
  content = u.read()
  out = open(filepath, "wb")
  out.write(content)
  out.close()
  print 'done writing', filepath

def delta_hours(ts1,ts2):
  return round((ts2-ts1).total_seconds() / 60.0 / 60.0,2)

def import_timestamp_jira(str):
  s = str.split()
  s = ' '.join(s[1:5])
  return datetime.datetime.strptime(s, "%d %b %Y %H:%M:%S")

def import_timestamp_bugzilla(str):
  s = str.split()
  s = ' '.join(s[0:2])
  return datetime.datetime.strptime(s, "%Y-%m-%d %H:%M:%S")

def import_from_jira(bug):
  import xml.etree.ElementTree as ET
  tree = ET.parse('./bugs/'+bug+'.xml')
  root = tree.getroot()
  t = root.findall('channel')[0].findall('item')[0]
  created = t.findall('created')[0].text
  resolved = t.findall('resolved')[0].text
  ts1 = import_timestamp_jira(created)
  ts2 = import_timestamp_jira(resolved)
  c = t.findall('comments')
  comments = 0
  if len(c) > 0:
    comments = len(c[0].findall('comment'))
  return tuple((delta_hours(ts1,ts2),comments))

def import_from_bugzilla(bug):
  import xml.etree.ElementTree as ET
  tree = ET.parse('./bugs/'+bug+'.xml')
  root = tree.getroot()
  t = root.findall('bug')[0]
  created = t.findall('creation_ts')[0].text
  resolved = t.findall('delta_ts')[0].text
  ts1 = import_timestamp_bugzilla(created)
  ts2 = import_timestamp_bugzilla(resolved)
  comments = len(t.findall('long_desc'))
  return tuple((delta_hours(ts1,ts2), comments))

for bug in bugs:
```

```
rep = bug.split('-')[0]
r = tuple()
if rep == 'ECLIPSE':
  r = import_from_bugzilla(bug)
else:
  r = import_from_jira(bug)
print str(r[0]) + '\t' + str(r[1])
```

9

Java ReentrantLock pseudocode

```
int state;
Thread owner;
Node head;
Node tail;

void lock() {
  if (!tryFastAcquire()) {
    slowAcquire();
  }
}

boolean tryFastAcquire() {
  if (!hasQueuedPredecessors() && COMPARE_AND_SET(state, 0, 1)) {
    setExclusiveOwner(currentThread());
    return true;
  }
  return false;
}

// Returns true if current thread
// is the first in the queue or it’s empty
boolean hasQueuedPredecessors();

void setExclusiveOwner(Thread thread) {
  owner = thread;
}

void slowAcquire() {
  // Creates and atomically enqueue node with current thread
  Node waiterNode = new Node();
  enq(waiterNode);

  // Try a few times to acquire the waiterNode and then park
  // until its predecessor wakes up this thread
  boolean failed = true;
  try {
    while (true) {
      if (waiterNode.pred == head && tryFastAcquire())) {
        setHead(waiterNode);
        failed = false;
        return;
      }
      if (shouldParkAfterFailedAcquire(waiterNode.pred, waiterNode))
        park();
    }
  } finally {
    if (failed)
      cancelAcquire(waiterNode);
  }
}

void release() {
  if (tryRelease()) {
    unparkSuccessor(head);
  }
}

boolean tryRelease(int releases) {
  if (currentThread() != owner)
    return false;
  setExclusiveOwner(null);
  setState(0);
  return true;
}

void park() {
  LockSupport.park(this);
}

// Wakes up successor of a given node in the waiting queue
// if necessary by using LockSupport.unpark on its successor.
void unparkSuccessor(Node);

// Cancel the waiting node and remove from waiting queue.
// If there's a successor parked, unpark it.
void cancelAcquire(Node);

// Atomically checks if the node is really the head
// of the queue and try fastPath codepath.
// On success, the node is dequeued from queue
bool tryFastAcquireIfHead(Node);

// Atomically enqueues node in the waiting queue. It repeately
// tries to COMPARE_AND_SET to update tail until succeeds.
// If head and tail are not initialized yet, there will be
// an extra COMPARE_AND_SET on head to a new Node and tail
// will be set as head.
void enq(Node);

// Make sure to park only when is guaranteed an unpark signal
// can be received. It decides based on specific protocol
// between predecessor of a given node and that node.
shouldParkAfterFailedAcquire(Node, Node);

// Returns Thread corresponding to the current thread
Thread currentThread();

// Disables the current thread for thread scheduling
// purposes unless the permit is available.
LockSupport.park();

// Makes available the permit for the given thread,
// if it was not already available.  If the thread
// was blocked on park then it will unblock.
LockSupport.unpark(Thread);
```

10

Instructions in R to evaluate time

```
exp1.dat = read.table(file="/Users/rafaelbrandao/r_input.dat", header = T)
attach(exp1.dat)

replica = factor(replica.)
student = factor(student.)
program = factor(program.)
lock = factor(lock.)

# Plot the box plot graphic using the response variable (time)
# associated with the locks with the following command

plot(time˜lock,col="gray",xlab="Lock",ylab="Time(seconds)")

# We set the effect model that will serve as basis for posterior analysis.
# Notice that the factor student is associated with the factor replica since for
# each replica we used a different pair of students. We also included the factor
# lock associated with the replica.

anova.ql<-aov(time˜replica+student:replica+program+lock+lock:replica)

library(MASS)
bc <- boxcox(anova.ql,lambda = seq(-3, 5, 1/10))
# If transformation is needed, we calculate lambda and use it:
# anova.ql<-aov(time**<lambda>˜replica+student:replica+program+lock+lock:replica)
lambda <- bc$x[which.max(bc$y)]

TukeyNADD.QL.REP<-function(objeto1)
{
y1<-NULL
y2<-NULL
y1<- fitted(objeto1)
y2<- y1ˆ2
objeto2<- aov(y2 ˜ objeto1[13]$model[,2] +
objeto1[13]$model[,3]:objeto1[13]$model[,2]
+ objeto1[13]$model[,4]+ objeto1[13]$model[,5])
ynew <- resid(objeto1)
xnew <- resid(objeto2)
objeto3 <- lm(ynew ˜ xnew)
M <- anova(objeto3)
MSN <- M[1,3]
MSErr <- M[2,2]/(objeto1[8]$df.residual-1)

F0 <- MSN/MSErr
p.val <- 1 - pf(F0, 1,objeto1[8]$df.residual-1)
p.val
}
TukeyNADD.QL.REP(anova.ql)

plot(anova.ql)
anova(anova.ql)
```

11

Undergraduate students's time results. Input used in R for analysis

```
replica, student, program, lock, time
1, 1, p1, A, 4996
1, 1, p2, B, 5367
1, 2, p1, B, 5070
1, 2, p2, A, 5260
2, 3, p1, A, 2700
2, 3, p2, B, 5306
2, 4, p1, B, 4490
2, 4, p2, A, 4017
3, 5, p1, A, 2340
3, 5, p2, B, 5290
3, 6, p1, B, 3377
3, 6, p2, A, 4473
4, 7, p1, A, 5400
4, 7, p2, B, 5360
4, 8, p1, B, 5400
4, 8, p2, A, 3641
5, 9, p1, A, 5400
5, 9, p2, B, 5400
5, 10, p1, B, 3600
5, 10, p2, A, 2406
6, 11, p1, A, 3290
6, 11, p2, B, 5370
6, 12, p1, B, 5400
6, 12, p2, A, 5320
7, 13, p1, A, 3424
7, 13, p2, B, 5356
7, 14, p1, B, 5400
7, 14, p2, A, 5160
8, 15, p1, A, 2593
8, 15, p2, B, 5279
8, 16, p1, B, 4705
8, 16, p2, A, 4535
9, 17, p1, A, 5160
9, 17, p2, B, 5430
9, 18, p1, B, 5250
```

```
9, 18, p2, A, 4246
10, 19, p1, A, 4967
10, 19, p2, B, 5413
10, 20, p1, B, 5400
10, 20, p2, A, 3804
11, 21, p1, A, 5280
11, 21, p2, B, 5160
11, 22, p1, B, 4174
11, 22, p2, A, 4886
12, 23, p1, A, 4271
12, 23, p2, B, 5569
12, 24, p1, B, 5400
12, 24, p2, A, 4788
13, 25, p1, A, 5400
13, 25, p2, B, 5239
13, 26, p1, B, 5310
13, 26, p2, A, 5390
14, 27, p1, A, 2027
14, 27, p2, B, 4271
14, 28, p1, B, 5090
14, 28, p2, A, 4450
15, 29, p1, A, 3000
15, 29, p2, B, 5315
15, 30, p1, B, 5400
15, 30, p2, A, 4210
```

12

Graduate students's time results. Input used in R for analysis

```
replica, student, program, lock, time
1, 1, p1, A, 1757
1, 1, p2, B, 2404
1, 2, p1, B, 1777
1, 2, p2, A, 1716
2, 3, p1, A, 1342
2, 3, p2, B, 2552
2, 4, p1, B, 2597
2, 4, p2, A, 1238
3, 5, p1, A, 1572
3, 5, p2, B, 2248
3, 6, p1, B, 3168
3, 6, p2, A, 2460
4, 7, p1, A, 1822
4, 7, p2, B, 2455
4, 8, p1, B, 2486
4, 8, p2, A, 2434
5, 9, p1, A, 3503
5, 9, p2, B, 3600
5, 10, p1, B, 2454
5, 10, p2, A, 1753
6, 11, p1, A, 1830
6, 11, p2, B, 3300
6, 12, p1, B, 2880
6, 12, p2, A, 890
7, 13, p1, A, 648
7, 13, p2, B, 940
7, 14, p1, B, 2247
7, 14, p2, A, 1363
```

13

Bibliography

References

1. Lu, Shan, et al. "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics." ACM Sigplan Notices. Vol. 43. No. 3. ACM, 2008.
2. Iván Sanchez. Latin Squares and Its Applications on Software Engineering. Master's thesis, Federal University of Pernambuco, Recife, Brazil, 2011.
3. Paola Accioly. Comparing Different Testing Strategies for Software Product Lines. Master's thesis, Federal University of Pernambuco, Recife, Brazil, 2012.
4. Raymond P. L. Buse, et al. Benefits and barriers of user evaluation in software engineering research. ACM SIGPLAN Notices, 46(10):643-656, October 2011.
5. Per Runeson. Using students as experiement subjects - an analysis on graduate and freshmen student data. In Proceedings of the 7th International Conference on Empirical Assessment in Software Engineering. Keele University, UK, pages 95-102, 2003.
6. Miroslaw Staron. Using students as subjects in experiments - A quantitative analysis of the influence of experimentation on students' learning process. In CSEET, apges 221-228. IEEE Computer Society, 2007.
7. G. E. P. Box, J. S. Hunter, and W. G. Hunter, Statistics for experimenters: design, innovation, and discovery. Wiley-Interscience, 2005.