Navigation

Question 1. Word Bank Matching (1 point each, 14 points total)

For each statement below, input the letter of the term that is *best* described. Note that you can click each word (cell) to mark it off. Each word is used at most once.

A. — A/B Testing	B. — Agile Development	C. — Alpha Testing	D. — Beta Testing
E. — Competent Programmer Hypothesis	F. — Confounding Variable	G. — Dynamic Analysis	H. — Formal Code Inspection
I. — Fuzz Testing	J. — Integration Testing	K. — Milestone	L. — Mocking
M. — Oracle	N. — Pair Programming	O. — Passaround Code Review	P. — Perverse Incentives
Q. — Race Condition	R. — Regression Testing	S. — Risk	T. — Sampling Bias
U. — Software Metric	V. — Static Analysis	W. — Streetlight Effect	X. — Triage
Y. — Unit Testing	Z. — Waterfall Model		

Q1.1: P

The company GlazeBook wants to reward programmers for their bakery social media website with a pay raise for finding more bugs than other programmers. This leads to programmers intentionally creating and reporting new bugs, rather than finding and resolving existing ones.

Q1.2: S

When writing any sort of innovative software, developers often use several processes to account for this important source of uncertainty. Once considered, it is typically mitigated or reduced.

Q1.3: X

Startup winter social media company Sleddit just released their website and received over 1000 bug reports in the span of a week. They only have 10 programmers so they have to prioritize some bugs over others and decide which to address first.

Q1.4: G

Developers want to check how fast their program runs under a variety of conditions. They use execution time profiling to run their instrumented code on a variety of test inputs.

Q1.5: B

Train company StationWide wants to be reactive to changing requirements as they create software that shows users the trains they can ride from one place to another. They created a very early prototype of the software during a two-week sprint to get feedback and fix problems the original code had during the next sprint. Daily meetings are used to keep everyone on the same page.

Q1.6: Y

For each class Aidan writes, they include local test cases to ensure that class is working as intended.

Q1.7: I

Video conferencing company Nyoom wants to test their latest chat feature to include custom emojis. To do so, they randomly generate 1000 valid and invalid emojis and use them in the chat feature to see whether the program responds as it should.

Q1.8: T

To check how well-received their new yellow colored buttons are, video-sharing company BlueTube sends out a survey to a group of people that only like the color green. Based on the survey results, BlueTube mistakenly concludes that the new yellow colored buttons would be disliked by all users.

Q1.9: D

After implementing a more advanced internet structure to their old fighting game, the company Twister sends a version of the game to a small number of end users to make sure it works before the feature releases.

Q1.10: A

BlipPng is a program that generates special pngs that automatically delete themselves after a period of time. They want to add a notification feature to tell users that the generated png has deleted itself. They release this feature to half of users but not to the other half of users. They include a survey to see how satisfied users are with the new feature.

Q1.11: E

After Mollie fixed a small mistake where they had a less-than (<) sign instead of a less-than-or-equal-to (<=) sign, Mollie's grade went from 5% to 100% on the Autograder.

Q1.12: L

Bird adoption company Flapple uses inexpensive functions with pre-determined outputs while initially testing their code base.

Q1.13: R

Daniel finds a bug in their code and fixes it. To prevent this same bug from recurring later, Daniel writes a test case to detect the presence of that bug.

Q1.14: F

Conner is trying to find which methods take longer to run. A first analysis finds that methods with more lines of code often have longer running times. However, this analysis does not account for the algorithmic complexity (e.g., Big-Oh) of the code. Ignoring that aspect means the analysis is misleading: some methods with fewer lines of code may still take a long time to run because they contain complex algorithms.

Question 2. Code Coverage (20 points)

You are given the following C functions. Assume that statement coverage applies only to statements marked **STMT_#**. In this question, we consider the **entire** program. That is, even if program execution starts from one particular method, we consider coverage with respect to the contents of all methods shown.

```
1  void Euphoria(str a, str b, int c, int d) {
2      STMT_1;
3      if (c < d) {
4          medicine(b, a);
5      }
6      STMT_2;
7      apple_juice(d, c);
8  }
9
10 void medicine(str a, str b) {
11     STMT_3;
12     if (a == 'rue') {
13         STMT_4;
14     }
15     if (b == 'jules') {
16         STMT_5;
17     }
18 }
19
20 void apple_juice(int c, int d) {
21     if (c == d) {
22         STMT_6;
23         return;
24     }
25     STMT_7;
26     apple_juice(c, c);
```

```
27      STMT_8;
28  }
29
```

Navigation

(a) (6 points) Provide **1** input (i.e., all four arguments) to `Euphoria(str a, str b, int c, int d)` such that the *statement* coverage will be **50%**. (We only consider statements marked **STMT_#**.) Use a format such as `("hello", "goodbye", 123, 456)` if possible.

Your answer here.

ANSWER: not possible

Different students were presented with different coverage targest. Some example answers include:

- 37.5: ('hello', 'hello', 10, 10)
- 87.5: ('hello', 'rue', 9, 10) or ('jules', 'hello', 1, 2)
- 62.5: ('hello', 'hello', 10, 9)
- 75: ('hello', 'hello', 9, 10)
- 100: ('jules', 'rue', 9, 10)

A common mistake for students receiving 87.5% was missing the variable swap of `a` and `b` when calling `medicine`. Answers such as ('rue', 'hello', 1, 2) are not correct in that setting.

Some students were asked about 50% coverage, which is not obtainable in the code above. The instructions do note *if possible*, so 'not possible' is a full-credit answer. (Most students who asked about this were given relevant information on Piazza. A small number of students may have been given misleading information on Piazza when asking about this; such students should file regrade requests for this question.)

(b) (2 points) **True / False**: there exists a test suite of size > 0 such that the test suite obtains **100%** *statement* coverage. (We only consider statements marked **STMT_#**.)

- ◯ True
- ◯ False

ANSWER: True

(c) (2 points) **True / False**: your answer from **Q2a** provides the lowest possible *path* coverage for the given code snippet.

- ◯ True
- ◯ False

ANSWER: True

One input visits one path through a program. (This is a difference between path and branch coverage.) Since Q2a asked for a single input, it visits a single path, which is the lowest possible path coverage. (Students for whom the answer was 'not possible' for Q2a answered this question as if it were possible: the key concept here is the notion that one input visits one path. Alternatively, if you said Q2a was not possible and interpreted that as zero inputs, that also provides the lowest possible path coverage if empty answers are considered. The result should be 'True' in all cases.)

(d) (5 points) Give a *minimum* test suite to reach **100%** *branch* coverage. Provide the test cases with their input in the form `Euphoria(str a, str b, int c, int d)`. For `a` and `b`, choose from only the values `{'rue', 'jules'}`. For `c` and `d`, choose from only the values `{1, 2}`. Write each test input on a separate line, using a format such as `("hello", "goodbye", 123, 456)` for each input if possible.

Your answer here.

ANSWER: 3 test cases are required. Example set of possible test cases: {('jules', 'rue', 1, 2), ('rue', 'jules', 2, 1), ('rue', 'jules', 1, 2)}

(e) (5 points) Describe a scenario in which a test suite that achieves 100% *statement* coverage might miss a bug in a program. Then describe what other approach (testing, coverage, analysis, etc.) could find that bug. Use 4 sentences or fewer.

Your answer here.

ANSWER: Answers will vary. In the lecture, examples such as division by zero and SQL injection were given. In a division by zero bug, you can visit the line with a non-zero denominator value and not see the bug. One way to find such an issue would be to use a dataflow analysis that determines if values are zero. Another example might be a race condition: you might have 100% statement coverage but not observe the right scheduler interleavings. A tool such as Eraser or CHESS could help find the race condition in such a situation. Student responses should not exceed 4 sentences.

Navigation

- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Extra Credit

Question 3. Short Answer (5 points each, 25 points)

(a) (5 points) Consider the following two pairs of tools, techniques, or processes. For each pair, give a class of defects or a situation for which the *first* element performs better than the second (i.e., is more likely to succeed and reduce software engineering effort and/or improve software engineering outcomes) and explain why.

a. integration testing better than maximizing branch coverage
b. spiral development better than waterfall model

Your answer here.

ANSWER:

a. Maximizing branch coverage may be more useful in the early stages of software development, where test cases are primarily focusing on testing all possible routes the code may take and testing as much as possible to ensure the base of the code is strong. Branch coverage is also useful for error-handling code. Integration testing may be more useful in the later stages of development, when there is already a working model and a new feature is added that needs to be tested in how it interacts and performs with a previously-developed module, possibly using a testing framework to simulate different scenarios. Integration testing is also relevant when the output of one module flows into another in a meaningful way and the dependent interactions between those modules must be tested.
b. Full credit answer will discuss how spiral relies on continuous releases of prototypes to reduce risk. Waterfall is divided into discrete phases over the course of an entire project. Spiral contains the aspects of waterfall, but Spiral is iterated multiple times during a project. Spiral is typically better because it reiterates stages and tests multiple times throughout the development cycle — waterfall is a simplified model used mainly to explain software processes in a classroom.

(b) (5 points) Identify two risks associated with Netflix's adaptation and usage of the *Chaos Monkey* dynamic analysis. Identify a measurement that could be used to guide decisions to reduce each risk.

Your answer here.

ANSWER:

Answers may vary, but possible risks could include incomplete analysis, high performance overhead for reporting, analysis inaccuracy, transparency limitations of instrumentation, and high complexity. Measurements could include measuring execution time and the slowdown caused, measuring the time taken to complete dynamic analysis, measuring the number of modules that fail when a module they depend on is made unavailable, measuring the types of defects fixed this way versus the previous method, etc. Netflix's Chaos Monkey is a script that continuously runs in Netflix's development environments to cause the developers to face unexpected outages at any time and resolve them to create more resilient software, so any reasonable answer here relating to dynamic analysis works.

(c) (5 points) Here are two examples of bugs that need to be triaged:

- A conversion error causes integers to occasionally flip signs (e.g., 4 becomes -4 and -4 becomes 4).
- A graphical error causes images to display 1.5x as large as expected, resulting in cropping.

For each bug, give an example of a situation where it would have high severity and a situation where the bug would have low severity and explain why.

Your answer here.

ANSWER:

There are many answers to this question, but some examples for each:

Navigation

- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Extra Credit

A conversion error causes integers to occasionally display as negative (e.g., 4 becomes -4).
High: Any situation where the sign change could cause negative or dangerous effects. An example might be a trading bot evaluating monetary decisions or a banking app's balance UI.
Low: Any situation where the sign change wouldn't cause negative effects aside from mild confusion. Think a recipe website's ingredient measurements, or a fitness app's distance shown. In such a case the user knows that "-2 eggs" is not possible, so the impact is minimized.

A graphical error causes images to display 1.5x as large as expected causing cropping.
High: Any situation where the picture change would result in a significant loss of important displayed data. An example might be a graphing application causing part of the plot to be lost, or a blueprint application causing measurement data to be cut out.
Low: Any situation where the picture change wouldn't result in a significant loss of important displayed data. Think of a forum profile picture being cropped wrong. While not desireable, the partial picture (coupled with the name nearby) would reduce the impact of the cropping.

(d) (5 points) Give an example of a software situation where fuzzing would be a better testing method than unit testing in terms of finding many bugs. Then give a situation where unit testing would be a better testing method than fuzzing in terms of the time or cost required. What kinds of bugs are likely to be revealed by fuzzing?

Your answer here.

ANSWER:

Fuzzing is likely to reveal more defects than unit testing in situations where data values flow across modules and random values are likely to reveal defects. For example, consider a log-reading module that passes its output to a square-root module that has a bug involving negative numbers. Unit testing might overlook corner cases and miss the square-root bug, but fuzzing random numbers would likely find it quickly, and fuzzing random log strings would likely result in negative numbers that are passed to the square root function to reveal the bug in a moderate time.

Unit testing is likely to be better than fuzzing in terms of time taken or cost if only a small number of values are relevant. In HW2, many students saw that randomly-created tests for HTML or XML functions were not very effective, since random creation rarely produced well-formed strings with correct matching brackets and syntax. Similarly, a division function with a division-by-zero error might be an example: a fuzzer that just generates random numbers might take a very long time to randomly generate zero. (Some fuzzers are more likely to choose numbers such as 0, 1, -1, MAX_INT and MIN_INT as a heuristic for this reason.)

Example domains may vary, but generally any software that includes both user input and sensitive data might be relevant. Consider medical interfaces, bank websites, stockbrokers.

Fuzzing is good at catching bugs related to overruns, overflows, error handling, and out-of-bounds accesses. Any mention of code being broken by receiving too much data should also get points.

(e) (5 points) You are a new team lead at *Mozzarella* and are in charge of leading a group of several developers. Your manager asks you to begin collecting the following developer efficacy data:

- Lines of code written per day
- Pull requests accepted into the master branch per month
- Peer ratings from an annual survey completed by coworkers

For each measurement, describe why it might not accurately represent a worker's efficacy and explain one way a malicious worker might exploit it.

Your answer here.

ANSWER:

Lines of code per day
This metric can be an incorrect measure for a variety of reasons. Developers might be in a design-heavy period of development or may code in a language that requires fewer lines. The developers may be focusing on software maintenance and thus looking at or changing old code rather than creating new code. The LOC metric can be gamed with whitespace, comments, or verbose syntax.

Navigation

- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Extra Credit

Pull requests accepted into the master branch per month
This metric can be inaccurate for a variety of reasons. Developers could be working on an experimental branch, could be partner programming on a different machine, or could be working on intensive bug fixing which could result in fewer total pull requests (but still many bugs fixed). This can be gamed by doing multiple trivial, small separate changes to inflate the number of pull requests.

An annual survey done by coworkers
Depending on the work environment you're in, biases (e.g., race, gender, etc.) can heavily negatively affect how others might would view you in a survey. As another example, someone who works asynchronously (and thus is not seen very often) might receive a lower or more neutral score. If someone is seen as competent or attractive, but in reality does little, their scores would be inflated. This can be gamed by developers only helping others and not working on their own tasks. Alternatively, two developers might conspire to give each other perfect peer review scores on each survey regardless of their actual work.

Question 4. Mutation Testing & Invariants (15 points)

Consider the code snippet below defining a function `modest_liskov`.

```
def modest_liskov(x: int, y: int, z: int):
  baz = 7
  garply = 0

  if (z >= y) or (z > x):
    baz = baz + 5
  elif (z < y):
    baz = baz + 7

  if (x != y):
    baz = baz - 5

  if (z == y) and (z == x):
    garply = garply - 1

```

(a) (5 points) A *postcondition* is similar to an *invariant*, but is always true just as or just after a function returns. (Informally, you can think of it as an assertion right at the end of the function.)

Consider the *postcondition*: `baz >= 7`.

The postcondition, `baz >= 7`, may be falsified by a first-order mutant of the original `modest_liskov` function. Create that mutant by making at most one edit to the below definition of `modest_liskov`. (To phrase this another way, you should make a single change to the program so that on some input it does not satisfy the postcondition.) Create the mutant by clicking inside the code window below and directly changing the initial code.

Mutant 1 (click inside to edit this directly):

```
def modest_liskov(x: int, y: int, z: int):
  baz = 7
  garply = 0

  if (z >= y) or (z > x):
    baz = baz + 5
  elif (z < y):
    baz = baz + 7

  if (x != y):
    baz = baz - 5

  if (z == y) and (z == x):
    garply = garply - 1

```

(b) (10 points) Create two *additional* first-order mutants of `modest_liskov` by making *exactly* one edit to each of the following definitions of `modest_liskov`. These two should target the same postcondition as your first mutant. (There is only one postcondition: consider the same one each time.) Note that the mutants you create must be different from the original

Navigation

- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Extra Credit

`modest_liskov`, the first mutant above, and from each other.

Below, you will then be asked to provide a single test input to `modest_liskov` such that the mutation adequacy score of your suite of three mutants, when each is given that single input, is exactly 1/3. We consider a mutant that fails to satisfy a postcondition as failing that test (i.e., such a mutant is killed). You may use this requirement to guide how you create the mutants.

Mutant 2:

```
def modest_liskov(x: int, y: int, z: int):
  baz = 7
  garply = 0

  if (z >= y) or (z > x):
    baz = baz + 5
  elif (z < y):
    baz = baz + 7

  if (x != y):
    baz = baz - 5

  if (z == y) and (z == x):
    garply = garply - 1

```

Mutant 3:

```
def modest_liskov(x: int, y: int, z: int):
  baz = 7
  garply = 0

  if (z >= y) or (z > x):
    baz = baz + 5
  elif (z < y):
    baz = baz + 7

  if (x != y):
    baz = baz - 5

  if (z == y) and (z == x):
    garply = garply - 1

```

What is a single test input to `modest_liskov` such that the mutation adequacy score for the three mutants is 1/3? All test inputs must be integers. Express your answer as a list in the form `[x, y, z]`. For example, if your inputs are `x = 3, y = 4, z = 5`, then you would write `[3, 4, 5]`.

Your answer here.

ANSWER: Answers may vary depending on the provided mutants.

Critical to answering this problem was the definition of the mutation adequacy score, which is the fraction of mutants killed by the test suite (i.e., failing at least one test). Students were typically instructed to produce scores such as 1/3 or 2/3. Consider the 2/3 case. Since the first mutant was *required* to "fail the test" (i.e., falsify the invariant), this meant that the student would have to make one more mutant that would fail the invariant and one that would succeed it.

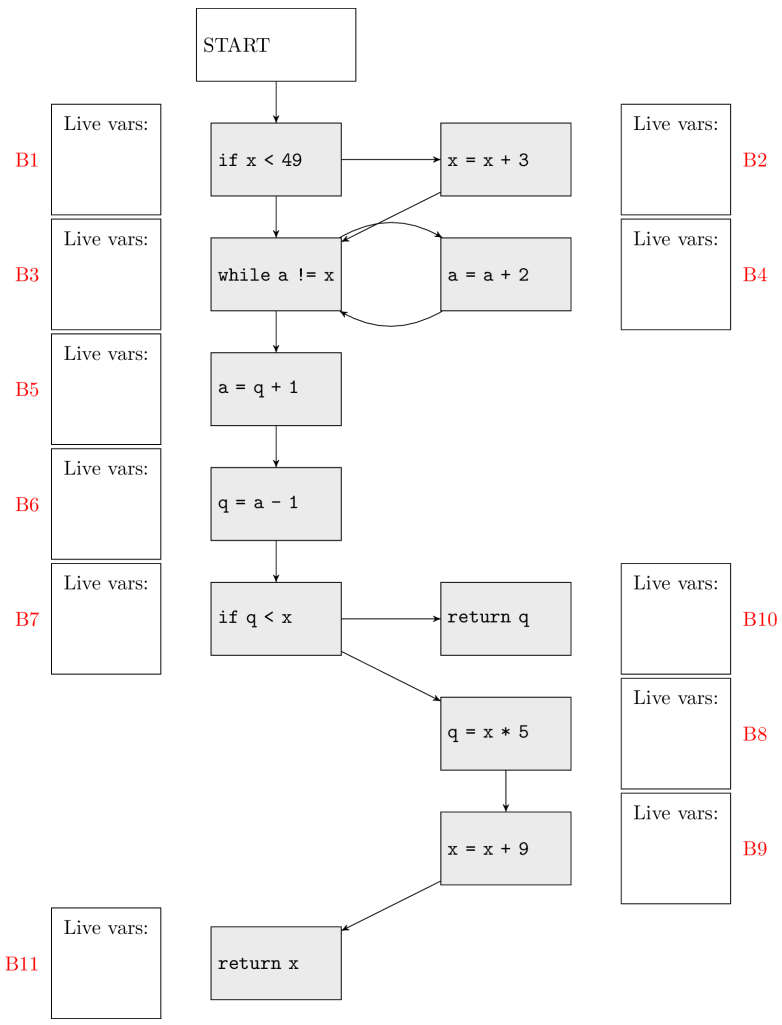
Creating a mutant that fails the invariant varies by invariant, but note that any first-order mutation operator (e.g., changing one variable reference, changing an expression, changing a statement, deleting a statement, etc.) mentioned anywhere in class (e.g., in the lecture, in the reading, in HW3, etc.) was fine to use.

Creating a mutant that is syntactically different but that does not falsify the invariant is often obtained by making "dead code". For example, inserting an `x = x` statement, or changing `p = q + r` to be `p = q + r + 0` work well.

Question 5: Dataflow Analysis (11 points total)

Consider a *live variable dataflow analysis* for three variables, `a`, `x`, and `q` used in the control-flow graph below. We associate with each variable a separate analysis fact: either the variable is possibly read on a later path before it is overwritten (live) or it is not (dead). We track the set of live variables at each point: for example, if `a` and `x` are alive but `q` is not, we write `{a, x}`. The

special statement `return` reads, but does not write, its argument. In addition, `if` and `while` read, but do not write, all of the variables in their predicates. (You must determine if this is a forward or backward analysis.)



(1 point each) For each basic block `B1` through `B11`, write down the list of variables that are live *right before* the start of the corresponding block in the control flow graph above. Please list only the variable names in lowercase without commas or other spacing (e.g., use either `ab` or `ba` to indicate that `a` and `b` are alive before that block).

B1 ANSWER: {'a', 'q', 'x'}

B2 ANSWER: {'a', 'q', 'x'}

B3 ANSWER: {'q', 'a', 'x'}

B4 ANSWER: {'q', 'a', 'x'}

B5 ANSWER: {'q', 'x'}

B6 ANSWER: {'x', 'a'}

B7 ANSWER: {'q', 'x'}

B8 ANSWER: {'x'}

B9 ANSWER: {'x'}

B10 ANSWER: {'q'}

B11 ANSWER: {'x'}

Question 6. Dynamic Analysis (15 points)

We decide to write our own dynamic analysis tool, Checkers, to help us deal with race conditions. Checkers works by following a standard *lockset* algorithm. For each shared variable, Checkers maintains a candidate set of locks that might protect that variable. The first time a shared variable is accessed by a thread, Checkers notes the set locks that thread currently holds. Every subsequent time that shared variable is accessed by a thread, the candidate set of locks guarding that variable is intersected with the currently-held locks of that thread. At the end, if a shared variable is not protected by any locks, a race condition is reported.

As part of its operation, Checkers instruments the program to log variable reads, variable writes, lock acquisition, and lock release. All such operations are instrumented to write the name and arguments of the operation, as well as a thread ID, to a log file.

(Note: This lockset algorithm works just like the one discussed in class. There are no hidden tricks or mistakes or changes in the description above, it is simply a summary for your convenience.)

(a) (2 points each, 4 points) We run Checkers on a series of programs and examine its output. For each of the programs below, consider if Checkers would report a race condition (i.e., if the computed lockset for a shared variable is empty) by examining the contents of the Checkers log file.

Variables with names that include `local` are thread-local variables that are not relevant for race conditions. Variables with names that include `shared` are shared variables that *can* be involved in race conditions. Variables with names that include `mu` are locks (short for *mutex* or *mutual exclusion*).

(ai) (2 points) Program:

```
1 int sharedA = 0;
2 mutex muA;
3 int sharedB = 0;
4 mutex muB;
```

```
5
6  void thread1() {
7      muA.lock();
8      muB.lock();
9      sharedA = 10;
10     sharedB = 20;
11     muB.unlock();
12     muA.unlock();
13 }
14
15 void thread2() {
16     muB.lock();
17     sharedB = 20;
18     muA.lock();
19     sharedA = 10;
20     muA.unlock();
21     muB.unlock();
22 }
```

Checkers log file:

```
1  thread 1: lock muA
2  thread 2: lock muB
3  thread 2: write sharedB
4  thread 1: lock muB
5  thread 2: lock muA
```

True/False: a race condition can be detected from the log file.

◯ True
◯ False
ANSWER: False

(aii) (2 points) Program:

```
1  int shared = 0;
2  mutex mu;
3
4  void thread1() {
5      mu.lock();
6      shared += shared;
7      mu.unlock();
8  }
9
10 void thread2() {
11     int local;
12     local = 12;
13     mu.lock();
14     shared -= 2;
15     mu.unlock();
16 }
```

Checkers log file:

```
1  thread 2: write local
2  thread 1: lock mu
3  thread 1: read shared
4  thread 1: read shared
5  thread 1: write shared
6  thread 1: unlock mu
7  thread 2: lock mu
8  thread 2: read shared
9  thread 2: write shared
10 thread 2: unlock mu
```

True/False: a race condition can be detected from the log file.

◯ True
◯ False
ANSWER: False

Navigation

- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Extra Credit

This question assesses student understanding of the relationship between a dynamic analysis and its test inputs and instrumentation. In the lecture, it was mentioned that not everything can be logged, and that the quality of a dynamic analysis (i.e., whether or not it produces a warning) can depend on whether or not it is run on high-quality inputs and is able to record relevant information.

With that in mind, we note that the question asks if a race condition can be detected **from the log file**. In some instances, students were presented with source code that clearly had a race condition, but where that could **not** be determined from the (low-quality) log provided. This highlights the difference between static and dynamic analyses: students who focused on the content of the code, effectively carrying out code inspection (a static analysis), were answering a different question (and potentially missing the issue that dynamic analyses can be hindered by poor inputs or incomplete log information).

(b) (4 points) We view Checkers as an analysis for helping us to conclude that a program has no race conditions. In this view, Checkers is sound if and only if it reports all such defects (i.e., has no false negatives). Is Checkers a sound analysis? Is it complete? Explain your reasoning in at most four sentences.

Your answer here.

ANSWER:

Checkers is not sound because it may erroneously categorize a program that contains a race condition as safe; that is, it may have false negatives. One way in this may happen is if it is not given enough inputs or if the log ends up not containing relevant information. For example, a program may immediately exit if its input is zero, rather than spawning any threads. If that is the only input/log available, Checkers will not see any accesses to shared variables, so it will not report any errors. However, the program may have race conditions on non-zero inputs.

Checkers is not complete because it may have false positives (false alarms). For example, a shared variable may be protected by different locks depending on different contexts. Lockset algorithms require that at least one lock protect the shared variable for the entire computation. See Slide 56 of the Dynamic Analysis lecture for an example.

While not required, students can find more information in the Eraser optional reading, since Checkers is Eraser renamed. That paper notes directly that Eraser can miss errors (page 398) and that Eraser also has false alarms (page 401).

1 point was given for Unsound, 1 point for Incomplete, and 1 point each for a valid justification. (Some students only responded to one half of the question.)

(c) (4 points) Support or refute the following statement: "A dynamic lockset algorithm such as Checkers is better suited than a static analysis tool would be for race condition detection."

Your answer here.

ANSWER:

Refute is possible. Dynamic lockset algorithms can be very inefficient (the Eraser algorithm reports 10x to 30x slowdowns). They may not halt on subject programs which run forever or deadlock. Some programs use concurrency control approaches other than locking. Dynamic analysis instrumentation may cause race conditions to disappear in practice ("Heisenbugs"). In addition, dynamic analyses require rich sets of test inputs. A student could describe a static analysis tool that keeps track of the set of locks held at each point, arguing that it fits in a static dataflow analysis framework (e.g., the set of locks only ever decreases, so the dataflow analysis terminates).

Support is possible. Dynamic algorithms are used quite a bit in practice for this task. The CHESS reading notes that there are many possible scheduler interleavings: enumerating and reasoning about them all with a purely static technique is not likely to be feasible (or will result in too much "I don't know" or "Top" sorts of approximations). Because the Checkers algorithm is simply the Eraser lockset algorithm from the lecture, students can bring in any evidence from the reading or lecture.

(d) (3 points) Suppose we want to test our dynamic analysis — that is, we want to gain confidence that it correctly reports a race condition if and only if the subject program has a race condition. To do so, we need a suite of subject programs for which we know whether each subject program has a race condition or not. Creating such a suite would be expensive. We decide to use just one part of mutation from mutation analysis: start with a single known-good program and randomly delete a call to lock or unlock to produce a new subject program that should now have a race condition. Support or refute the claim that using this simple part of mutation would be a good way to produce a test suite for Checkers. (Note that in this question a test input to the Checkers analysis is, itself, another program, which also has its own input. Note also that this question is about using a mutation operator, but is not about standard mutation analysis.)

Navigation

- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Extra Credit

Your answer here.

ANSWER:

Both are possible, but refute is more likely. The mutation approach does reduce the cost of developing new subject programs. However, there are a number of concerns. First, the resulting subject programs are not very diverse. For example, if the starter "known-good" program does not have any loops, none of the mutants will either, and so Checkers will never be tested on looping programs. In addition, the resulting test suite is unbalanced: only the original known-good program has "no races" as its expected answer, all of the others have "race condition" as the expected answer. Checkers could produce many false alarms (i.e., Checkers could basically always say "race condition") and that would not be noticed, because almost every expected answer is "race condition". Finally, a dynamic analysis relies on the quality of the input to the subject program. Nothing was discussed about how inputs would be made to the known-good subject program or the mutants.

A support answer would have to address some of the issues above for full credit; merely indicating that it would save development time would not be sufficient.

Much like HW3 or the in-class discussion of instrumentation, this question explicitly required students to think about notions of "time" or the "stages" of analysis. It also asked students to stretch and apply mutation in a setting other than pure mutation testing to assess test suite quality.

Extra Credit

Each question below is for 1 point of extra credit unless noted otherwise. We are strict about giving points for these answers. No partial credit.

(1) What is your favorite part of the class so far?

Your answer here.

(2) What is your least favorite part of the class so far?

Your answer here.

(3) If you read any optional reading, identify it and demonstrate to us that you have read it. (2 points)

Your answer here.

(4) If you read any *other* optional reading, identify it and demonstrate to us that you have read it. (2 points)

Your answer here.

(5) In your own words, identify and explain any of the bonus psychology effects. (2 points)

Your answer here.