Exercise 03

Structural Testing

Rigorous Software Engineering, ETH Zurich

Task 1 (Pairwise Testing)**.** Consider the following method signature.

```
int compute(boolean optimize, String name, int val, int limit)
```

The set of valid inputs to this method is given below. Unfortunately, creating unit tests for all possible combinations of these inputs results in 54 tests.

$$\begin{aligned}\texttt{optimize} &\in \{\texttt{true}, \texttt{false}\}\\ \texttt{name} &\in \{\text{“Alice”}, \text{“Bob”}, \text{“Charlie”}\}\\ \texttt{val} &\in \{0, 1, 2\}\\ \texttt{limit} &\in \{0, 10, 20\}\end{aligned}$$

a) Use pairwise (2-way) testing to derive a set of test cases that is as small as possible but includes all value combinations for each input pair. How many test cases do you need?

b) Let `f` be an arbitrary method with at least two parameters $p_1, p_2, \ldots, p_k$ and let n_i be the number of possible values for parameter p_i. Further, let $a = \max\{n_1, \ldots, n_k\}$ and $b = \max\{n_1, \ldots, n_k\} \setminus \{a\}$ be the largest respectively second-largest n_i.
Prove or disprove:

(i) Pairwise testing requires at least $a \cdot b$ test cases for `f`.

(ii) Pairwise testing requires at most $a \cdot b$ test cases for `f`.

Note: Subtask (ii) is challenging.

Task 2 (Coverage Metrics)**.** Let P be an arbitrary loop-free program and T a non-empty test input set for P. Prove or disprove the following statements.

a) If T has 100% branch coverage in P, then T has 100% statement coverage in P.

b) For arbitrary c, if T has path coverage c in P, then T has statement coverage at least c in P.

Task 3 (Structural Testing)**.** Consider the three methods below, which were documented by the developer. Each method implementation contains a bug.

```java
/**
 * Returns true if and only if the array a contains at least two
 * different entries with value n. The array a must not be null.
 */
int hasDuplicate(int[] a, int n) {
  if (a == null) {
    throw new IllegalArgumentException("Array␣is␣null");
  }
  int found = 0;
  for (int i = 0; i <= a.length; i++) {
    if (a[i] == n) {
      found = found + 1;
    }
    if (found == 2) {
      break;
    }
  }
  return found == 2;
}


/**
 * Returns the average of all integers in the array a.
 * The array a must neither be empty, nor null.
 */
double average(int[] a) {
  if (a == null) {
    throw new IllegalArgumentException("Array␣is␣null");
  }
  if (a.length == 0) {
    throw new IllegalArgumentException("Array␣is␣empty");
  }
  int sum = 0;
  for (int i = 1; i < a.length; i++) {
    sum += a[i];
  }
  return (double) sum / a.length;
}
```

```
/**
 * Returns the average of all integers in the array a at positions
 * ranging from l (inclusive) to r (exclusive).
 */
double averageSubarray(int[] a, int l, int r) {
  if (a != null && 0 <= l && r <= a.length && l <= r) {
    int sum = 0;
    int count = 0;
    while (l < r) {
      sum = sum + a[l];
      count = count + 1;
      l = l + 1;
    }
    return (double) sum / count;
  } else {
    return 0;
  }
}
```

a) Draw the control flow graph of the method `hasDuplicate`.

b) For each of the three methods `hasDuplicate`, `average` and `averageSubarray`:

  (i) Find a test input revealing the bug in the method implementation.

  (ii) Can you find a set of test inputs that achieves 100% *statement coverage* in the method but does not find the bug? If yes, provide the test inputs.

  (iii) Can you find a set of test inputs that achieves 100% *branch coverage* in the method but does not find the bug? If yes, provide the test inputs.

  (iv) Can you find a set of test inputs that achieves 100% *loop coverage* in the method but does not find the bug? If yes, provide the test inputs.

Exercise 09

Symbolic Execution

Rigorous Software Engineering, ETH Zurich

Task 1 (Symbolic Execution with Loops)**.** Consider the function `gcd` below, which computes the greatest common divisor of two natural numbers using the Euclidean algorithm. Assume that the function is always invoked with positive arguments. In this task, you will apply symbolic execution to `gcd`.

```
int gcd(int a, int b) {
    while (b != 0) {
        int tmp = b;
        b = a mod b;
        a = tmp;
    }
    return a;
}
```

a) As discussed in the lecture, symbolic execution cannot handle unbounded loops such as in `gcd`. In practice, one often employs a static bound on the number of loop iterations and explicitly unrolls the loop. Apply this idea to `gcd` to obtain a function `bounded_gcd` where the loop body from `gcd` is executed *at most twice*.

b) Perform symbolic execution on `bounded_gcd` to find all symbolic states with a satisfiable path constraint at the return statement. Remember that `a` and `b` are assumed to be positive.

c) The transformation performed in (a) changed the behavior of `gcd` for some inputs. Provide positive values for `a` and `b` such that `gcd(a, b)` and `bounded_gcd(a, b)` return different results.

   What does this mean for the output of symbolic execution? Is it still an under-approximation of the result computed by `gcd`?

d) How can you modify the transformation from (a) and the semantics of symbolic execution such that it computes an under-approximation of `gcd`?

   Hint: Try to rule out executions not possible in `gcd` using path constraints.

Task 2 (Concolic Execution)**.** Consider the two functions below. The function `pow`, written by Bob, is supposed to compute the power `b`$^{\texttt{e}}$ for non-negative `e`. Bob's coworker Alice uses `pow` in her `main` method. Note that mathematically, for any $b \in \mathbb{Z}$ and *even* $e \in \mathbb{N}_{\text{even}}$ it is $b^e \geq 0$. Alice added an according assertion to `main`.

```
void main(int b, int e) {
  int r = pow(b, e);
  if (e mod 2 == 0) {
    if (r < 0) {      // (*)
      assert(false);  // should not happen
    }
  }
}
```

```
int pow(int b, int e) {
  int r = b;
  for (int i = 1; i < e; i++) {
    r = r * b;
  }
  return r;
}
```

Unfortunately, Bob's implementation of `pow` contains a bug, which may lead to a violation of Alice's assertion. In this task, you will use concolic execution to derive a test input for `main` which spots the bug.

a) Perform a first run of `main` using concolic execution with concrete inputs `b` = `e` = 0. What is the path constraint gathered during the execution?

b) The previous run entered the (empty) **`else`** branch of the if-statement `(*)`. Modify the path constraint from before by negating the sub-constraint collected for `(*)`. Then, find a satisfying assignment for the new constraint.

c) Perform a second run using your new inputs from (b). What is the path constraint gathered during the execution? Can you reach the assertion?

d) Now, assume that the function `pow` is part of a library whose source code is not accessible for Alice. This is, she can only execute `pow` for concrete inputs. Repeat subtask (a) for this setting.

e) Can you proceed analogously as in subtasks (b–c) to find inputs violating the assertion? Will concolic execution ever reach the assertion in this example?

f) The tool "PathCrawler Online"[1] can be used to perform concolic execution of C code. Compare your concrete inputs from (a–c) to the test cases generated by this tool for an equivalent C implementation.

   Hints:

   - Use `pathcrawler_assert(0);` to model the assertion.
   - Navigate to "Test your code" and follow the guidelines to upload your C implementation. You have to create a ZIP archive containing the code and select `main` as the test function.

[1] `http://pathcrawler-online.com:8080/`

- Select “Customize test parameters” and configure the variable domains by $0 \leq e \leq 5$ and $-10 \leq b \leq 10$ (to avoid integer overflows). Note that the result is not deterministic.
- To view the generated test cases, navigate to “Test Cases” and then “Input Values”. In the “Verdict” tab, an assertion failure will be reported for test cases reaching the assertion.

Exercise 11

Static Alloy Models

Rigorous Software Engineering, ETH Zurich

Task 1 (Modeling a University). Consider the university system described below. You plan to implement this system in an object-oriented programming language (e.g., Java).

- There are undergraduate and graduate students. No student is both an undergraduate and a graduate student.
- A student should register at a university. Registered students are considered legal, while unregistered students are not.
- Every student has a unique student ID, and he or she has exactly one major subject. There must not be any unassigned student IDs.
- Students registered at the same university with the same major are classmates.
- Graduates and undergraduates are never classmates. Also, students from different universities or with different majors are never classmates.
- A student cannot be his/her own classmate.

a) Think about how to structure this system in terms of classes and relations (i.e., pointers between classes). Draw an informal class diagram. The fact whether a student is legal should be encoded using a simple boolean flag.

b) Create an Alloy model of the university system, reflecting your structure from subtask (a). Visualize the model for 2 universities, 3 majors, 3 students, and 3 IDs by running Alloy. [1]

Task 2 (ETH Bus—*from a previous exam*). Consider the following Alloy model of the ETH bus that connects ETH Zentrum with Hönggerberg. The model is incorrect.

```
sig ETHBusStation {
  next: set ETHBusStation
}

```

[1]Instructions how to download and run Alloy: `https://alloytools.org/download.html`

```
one sig Polyterrasse, HaldeneggRight, HaldeneggLeft,
 ETHHonggerberg in ETHBusStation {}

sig ETHBus {
  station: lone ETHBusStation
}

fact {
  no (ETHHonggerberg - HaldeneggRight)
  all s: ETHBusStation | ETHBusStation in s.^next and some s.next
  all b1, b2: ETHBus | b1.station != b2.station
}

pred show {}
```

a) Without running the Alloy analyzer, find and visualize an instance produced by the command `run show for 2 but 1 ETHBusStation`.

b) Consider the following list of specifications (i–v), not all of which are currently fulfilled by the given Alloy model. Fix the Alloy model such that all specifications are fulfilled.

   Hint: Only few local changes to the model are required.

   (i) There are exactly 4 ETHBusStations: Polyterrasse, HaldeneggRight, Haldenegg-Left and ETHHonggerberg.

   (ii) The ETHBusStations form one (directed) circle.

   (iii) There is no ETHBusStation between ETHHonggerberg and HaldeneggRight.

   (iv) There can be at most one ETHBus at each ETHBusStation.

   (v) There is at least one ETHBus.

Task 3 (Exercise A.3.1 of [1]). A song by Doris Day goes as follows: "Everybody loves my baby, but my baby don't love nobody but me". David Gries has pointed out that, from a strictly logical point of view, this implies "I am my baby".

a) Check this implication by formalizing the song as an Alloy model.

b) Doris Day probably meant something different. Modify the model accordingly and show that the implication is no longer true.

References

[1] Daniel Jackson. *Software abstractions: logic, language, and analysis*. Rev. ed. Cambridge, Mass: MIT Press, 2012.

Exercise 05

Mathematical Concepts

Rigorous Software Engineering, ETH Zurich

Task 1 (Proofs)**.**

a) Let $(L, \sqsubseteq)$ be a poset and assume there exists a least element $\bot \in L$. Prove that this least element must be unique.

b) Consider the relation $\preceq \subseteq \mathbb{N} \times \mathbb{N}$ on natural numbers, defined as

$$a \preceq b \quad :\Longleftrightarrow \quad a \leq b \ \ \wedge \ \ (b - a) \bmod 3 = 0.$$

For example, it is $2 \preceq 5$, but $3 \not\preceq 5$. Prove or disprove the following statements:

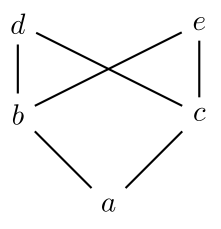
(i) $(\mathbb{N}, \preceq)$ is a poset.

(ii) $(\mathbb{N}, \preceq)$ is a complete lattice.

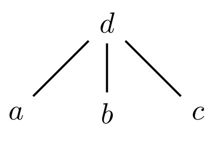
(iii) the function $f : \mathbb{N} \to \mathbb{N}$ defined as $f(x) := x + 2$ is monotone (w.r.t. $\preceq$).

(iv) the function $f : \mathbb{N} \to \mathbb{N}$ defined as $f(x) := 2x$ is monotone (w.r.t. $\preceq$).

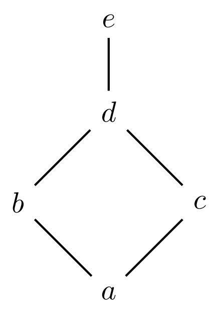
Task 2 (Lattices)**.** Which of the following Hasse diagrams represent complete lattices?



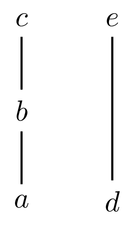
(a)



(b)

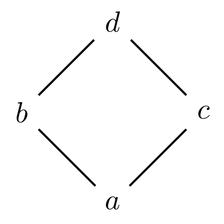


(c)



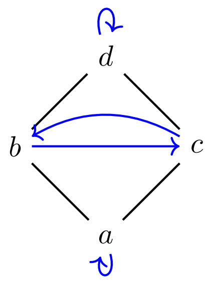
(d)

Task 3 (Monotonicity and Fixpoints)**.** Consider the lattice $(A, \sqsubseteq)$, where $A = \{a, b, c, d\}$. The partial order $\sqsubseteq \subseteq A \times A$ is depicted in the Hasse diagram below.

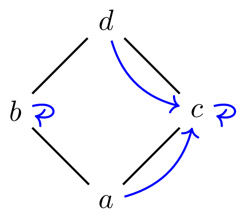


a) List the elements of $\sqsubseteq$.

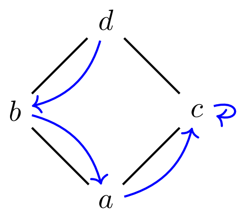
b) For each of the functions $f, g, h \colon A \to A$ indicated below: (i) decide whether the function is monotone, (ii) provide the set Fix$(\cdot)$ of fixpoints, (iii) provide the set Red$(\cdot)$ of post-fixpoints, and (iv) provide the least fixpoint, if it exists.



Function f

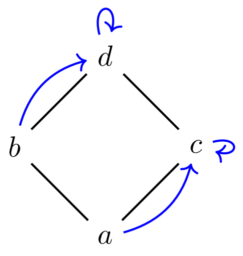


Function g

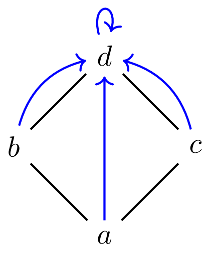


Function h

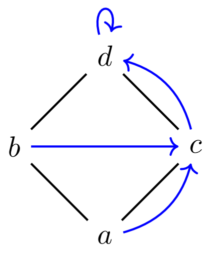
Task 4 (Approximation)**.** Consider again the lattice $(A, \sqsubseteq)$ from task 3 and the monotone functions $f', g', h' \colon A \to A$ indicated below.



Function f'



Function g'



Function h'

a) Which of the functions f', g' and h' approximate each other?

b) Next, we introduce a lattice $(B, \preccurlyeq)$ with $B = \{u, v, w\}$ and $\preccurlyeq$ as depicted below. We connect this lattice with the lattice $(A, \sqsubseteq)$ from before using the concretization $\gamma\colon B \to A$ given below.

$$\begin{tikzcd}
w & \gamma(w) = d \\
v \arrow[u, dash] & \gamma(v) = b \\
u \arrow[u, dash] & \gamma(u) = a
\end{tikzcd}$$

Provide *monotone* functions $k, k'\colon B \to B$ such that, in the concretization-based view, (i) k approximates *all* functions f', g', h', and (ii) k' does *not* approximate any of f', g', h'.

Exercise 10

Dynamic Race Detection

Rigorous Software Engineering, ETH Zurich

Task 1 (Dynamic Race Detection for Fork-Join)**.** In this task, we are going to apply dynamic race detection to the **`fork-join`** program below. In particular, the function `voting` encodes a parallel voting system where `n_people` people decide how to vote (`vote` input array). Multiple threads then update the `yes` and `no` counts accordingly. Our goal is to detect that `voting` behaves non-deterministically.

```
1  void voting(boolean[] vote, int n_people) {
2      // BLOCK: A
3      int yes = 0;
4      int no = 0;
5
6      for (int i = 0; i < n_people; i++){
7          fork {
8              if (!vote[i]) {         // BLOCK: B-i
9                  int tmp = no + 1;   // BLOCK: C-i
10                 no = tmp;           // BLOCK: D-i
11             } else {
12                 int tmp = yes + 1;  // BLOCK: E-i
13                 yes = tmp;          // BLOCK: F-i
14             }
15         }
16     }
17     join;
18     // BLOCK: G
19     print(yes + "␣voted␣yes");
20     print(no + "␣voted␣no");
21 }
```

a) What memory locations (i.e., variables) are potentially the target of a data race? In the following, we will call these variables $V_{critical}$.

b) Define the Happens-Before (HB) model in graph representation for concrete inputs `vote = [true, false]`, `n_people = 2` (the first person votes "yes" while the second person votes "no"). To this end, treat lines 3–4 as a single atomic action (Block `A`) and line 8 as 2 blocks (one per thread: Block `B-0` for person 0 and Block `B-1` for person 1). Similarly, treat each line in 9, 10, 12, 13 as two blocks (one per thread). Finally, treat lines 19–20 as a single block. Do not include any other blocks.

c) Extend your HB model by vector clocks, as discussed in the lecture.

d) For each node in your HB model, find the set of variables from $V_{critical}$ accessed in the atomic action, separating between reads and writes. Does the dynamic race detection algorithm from the lecture find a race condition in your HB model?

e) Repeat subtasks (b–d) for an input where both people vote "yes" (`vote = [true, true]`).

f) In the HB model of subtask (e), find a concrete example of non-determinism. That is, find two different total orderings of the atomic actions respecting the happens-before relation but leading to a different message being printed in line 19.

g) Now, assume that each thread is executed atomically (i.e., once a tread starts, it cannot be interrupted by another thread). Can the dynamic race detection algorithm find a data race for the input of subtask (e)?

Note: In case you are interested, you can find more details about the approach from the lecture in the original research paper [1].

References

[1] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *ACM SIGPLAN Notices*, volume 48, pages 151–166. ACM, 2013. Available at: `https://files.sri.inf.ethz.ch/website/papers/oopsla13-web.pdf`.

Exercise 02

Modularity and Functional Testing

Rigorous Software Engineering, ETH Zurich

Task 1 (Identifying Design Patterns)**.** Design patterns are commonly used in real-world software. Carefully examine the documentation of the Java APIs listed below. [1] For each of the classes/methods, decide which of the following design patterns is present: (i) Factory, (ii) Flyweight, (iii) Observer, (iv) Strategy, or (v) Visitor.

a) `javax.swing.plaf.basic.BasicToolBarUI.FrameListener`

b) `java.text.NumberFormat`, method `getInstance()`

c) `javax.lang.model.element.AnnotationValueVisitor`

d) `javax.net.ssl.SSLSessionBindingEvent`

e) `java.util.Comparator`, method `compare(T o1, T o2)`

f) `javax.xml.parsers.DocumentBuilderFactory`

g) `java.lang.Boolean`, method `valueOf(boolean b)`

h) `javax.servlet.Filter`, method `doFilter()` [2]

i) `javax.lang.model.type.TypeMirror`

j) `java.lang.Character`, method `valueOf(character c)`

Task 2 (Coupling)**.** Consider the following GUI framework with support for multiple operating systems. Using the framework, clients can create and display buttons.

```
public interface Button {
    void display();
}
```

[1]See `https://docs.oracle.com/javase/9/docs/api/overview-summary.html`
[2]See `https://docs.oracle.com/javaee/7/api/toc.htm`

```java
public class WinButton implements Button {
    @Override
    public void display() {
        // render the button in Windows
    }
}

public class OSXButton implements Button {
    @Override
    public void display() {
        // render the button in OSX
    }
}
```

Assuming clients can determine the OS currently running the application, they can create an according button using `new WinButton()` or `new OSXButton()`.

a) What is the problem of this implementation in terms of maintainability? Identify coupling issues faced by clients of the class hierarchy.

b) Which design pattern discussed in the lecture can be used to remedy the problem? Apply this pattern to the code above.

Task 3 (More Coupling)**.** Consider the following Java classes, which were implemented by an inexperienced developer. The class `StudentsManager` is used to track the students registered for a course, where students are identified by an integer id. At the end of the course, the class `GradesManager` is used to register student grades and compute the overall grade average.

```java
public class StudentsManager {
    public int nStudents;
    public Set<Integer> studentIds;

    public StudentsManager(Set<Integer> studentIds) {
        this.studentIds = studentIds;
        this.nStudents = studentIds.size();
    }

    public void addStudent(int id) {
        if (!this.studentIds.contains(id)) {
            this.nStudents++;
            this.studentIds.add(id);
        }
    }
    public int getNumberOfStudents() {
        return this.nStudents;
    }
}


public class GradesManager extends StudentsManager {
    private double rollingSum = 0.0;
    private Set<Integer> studentsWithoutGrades;

    public GradesManager(Set<Integer> studentIds) {
        super(studentIds);
        this.studentsWithoutGrades = new HashSet<Integer>(studentIds); // cloning
    }

    public void setGrade(int id, double grade) {
        if (this.studentsWithoutGrades.contains(id)) {
            this.rollingSum += grade;
            this.studentsWithoutGrades.remove(id);
        }
    }

    public double computeAverageGrade() {
        if (!this.studentsWithoutGrades.isEmpty())
            throw new RuntimeException("Not␣every␣student␣is␣assigned␣a␣grade!");
        return this.rollingSum / this.nStudents;
    }
}
```

The developer intends to use the classes in the following client code:

```
StudentsManager sMgr = new StudentsManager(new HashSet<Integer>());
sMgr.addStudent(10); sMgr.addStudent(20);

GradesManager gMgr = new GradesManager(sMgr.studentIds);
gMgr.setGrade(10, 4.5); gMgr.setGrade(20, 6.0);
gMgr.computeAverageGrade();
```

a) The classes `StudentsManager` and `GradesManager` exhibit severe coupling issues. Find all problems related to coupling in the classes and briefly explain why they should be fixed.

b) Refactor the classes and the client code in order to minimize coupling and address the issues identified previously.

Task 4 (Functional Testing)**.** Consider the function `computeMedian` below, which aims to implement the following specification: *Given a non-null array of numbers in* $[-100, 100]$*, return the median of the numbers. If the array is empty, the median is* 0*.*

```
class Median {
    // requires: numbers != null and for all i: -100 <= numbers[i] <= 100
    public static double computeMedian(double[] numbers) { ... }
}
```

a) Think about the functionality of the median. Following the functional testing strategy, partition the input space of `computeMedian` into three equivalence classes. Consider only inputs that fulfill the method's precondition. For each equivalence class, find a representative input.

In the accompanying ZIP file, we provide a candidate implementation of the class `Median` and various unit test skeletons. The code comes with a `pom.xml` file defining a Maven[3] project configuration where JUnit is already set up. You can either import this project in your favorite Java IDE (e.g., Eclipse[4] or VS Code[5]) or use Maven directly from the command line as exemplified below.

```
# installs Maven and OpenJDK on Ubuntu 18
sudo apt update && sudo apt install maven

mvn clean   # clean build
mvn test    # compiles the code and runs unit tests
```

b) The current implementation of `computeMedian` contains a bug, which we aim to find using unit tests.

   In the file `MedianPartitionsTest.java`, implement the three test cases designed in subtask (a) and run the tests. Based on the failed test, find and fix the bug in `Median.java`. Run the tests again to see whether your fix addresses the issue.

c) In order to easily extend our test suite, we want to introduce parametrization. To this end, we can use the `@RunWith(Parameterized.class)` and `@Parameterized.Parameters` annotations in JUnit. [6]

   In the file `MedianParameterizedTest.java`, implement a parameterized unit test covering all equivalence classes from subtask (a). The test should have two parameters: The input number list, and the expected median. Instantiate the parameters such that you obtain at least 5 concrete test cases. Run your tests again.

d) In the previous subtasks, you have manually provided the expected results. We will now get rid of this "ground truth" using a test oracle. In particular, the oracle

[3]See `https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html` for a quick introduction to Maven.
[4]`https://www.eclipse.org/`
[5]`https://code.visualstudio.com/`
[6]See `https://junit.org/junit4/javadoc/4.12/org/junit/runners/Parameterized.html` for details and an example.

checks whether a computed candidate median is correct, but without actually computing the true median. [7] To this end, we can leverage the following formal definition of the median: [8] *The median is a value such that at most half of the numbers in the array are less, and at most half of the numbers in the array are greater than this value.*

In the file `MedianOracleTest.java`, implement a generic test oracle checking the correctness of a proposed median according to the definition above. Based on this oracle, implement a parameterized unit test where the expected median is not required as a parameter. Run your tests again.

e) Given the test oracle from the previous subtask, we can now automatically generate an arbitrary number of test cases. In the file `MedianRandomTest.java`, leverage your solution of the previous subtask to implement a random unit test which generates 100 random test inputs with random lengths. Run your tests again.

[7] Note that if the oracle would make use of `computeMedian` itself, testing would be pointless.

[8] This definition is less rigid than the usual definition, which uses the average of the two middle values if the number of values is even. Still, we can use it to find bugs in our implementation.

Exercise 04

First Steps with Abstract Interpretation

Rigorous Software Engineering, ETH Zurich

Consider the following program P. Based on manual inspection, we believe that the value of x at the end of P is always even and negative, and line 3 is unreachable (see the three assertions below). In this exercise, we are trying to prove these properties.

```
   function(int x, int y) {
0:    x := y * 2
1:    while x >= 0
2:       if x = 1
3:          x := x + 1   // assert not reachable
         else
4:          y := x - 1
5:          x := y - 1
6:    skip               // assert x even
                         // assert x < 0
   }
```

Task 1 (Concrete States and Traces)**.** A *concrete state* is a pair $\langle l, \sigma \rangle$ of a *label* l and *store* σ, where σ is a mapping assigning concrete values to variables. Usually, the label is the program counter (line number), pointing at the state *before* that line. For example, when executing `function(3, 7)`, the initial state is $\langle 0, \{\mathsf{x} \mapsto 3, \mathsf{y} \mapsto 7\} \rangle$. In order to point at the final state, we introduced a **`skip`** (no-op) statement at the end of P.

We write $\langle l_1, \sigma_1 \rangle \to \langle l_2, \sigma_2 \rangle$ if the program action at label l_1 changes the state to $\langle l_2, \sigma_2 \rangle$. A *concrete trace* of a program P is a sequence of states $\langle l_1, \sigma_1 \rangle \to \langle l_2, \sigma_2 \rangle \to \cdots$ encountered during an execution of P (for some concrete parameters). Finally, $[\![P]\!]$ is the set of all states in any trace of a program P (for any parameters).

a) Determine the concrete traces for `function(5, 0)` and `function(3, 1)`.

b) Which concrete states occur at labels 3 and 6 in these traces? Do they satisfy the assertions above?

c) Assume you were given the set $[\![P]\!]$. Could you use this to prove the assertions?

d) Is it possible to compute $[\![P]\!]$ for a general program P? More precisely, can we construct an algorithm that decides for any P whether $s \in [\![P]\!]$ for a given s?

Task 2 (Parity Domain)**.** In the lecture, you have already seen the *Sign* and *Interval* abstract domains. In this task, we are going to use another domain called *Parity*:

$$\begin{tikzcd}
 & \top & \\
even \arrow[ur, dash] & & odd \arrow[ul, dash] \\
 & \bot \arrow[ul, dash] \arrow[ur, dash] &
\end{tikzcd}$$

In this domain, *even* represents all even integers (including zero), and *odd* represents all other integers. The elements $\top$ and $\bot$ are defined as usual. We consider integers only.

An *abstract (parity) state* is a mapping Labels $\to$ (Vars $\to \{\bot, \top, odd, even\}$) from labels to variables to elements from the *Parity* domain. Note that in contrast to a concrete state, an abstract state incorporates all labels.

a) Think about transformers for *Parity.* Specifically, how would you define *even* `+ 1` and $\top$ `* 2` in order to be sound?

b) Starting from the initial abstract state given below, perform abstract interpretation in the *Parity* domain by hand as introduced in the lecture. That is, iterate over P's state until you reach a fixed-point.

pc	`x`	`y`
0	$\top$	$\top$
1	$\bot$	$\bot$
2	$\bot$	$\bot$
3	$\bot$	$\bot$
4	$\bot$	$\bot$
5	$\bot$	$\bot$
6	$\bot$	$\bot$

c) Does the resulting abstract state incorporate the concrete states from task 1b?

d) Can you use the abstract fixed-point state to prove the three assertions?

Task 3 (Comparing Domains)**.** Are the domains *Sign*, *Interval*, and *Parity* pair-wise comparable in terms of precision? For those that are comparable, which one is more precise? For those that are incomparable, provide an example program with an assertion that can be verified using one but not the other domain, and vice-versa.

Exercise 07

Widening and Pointer Analysis

Rigorous Software Engineering, ETH Zurich

Task 1 (Widening)**.** Consider the program below.

```
    g(int x) {
1:     y := 0
2:     while (x > 0 && x <= 100)
3:        y := y + x
4:        x := x - 1
5:  }
```

a) Compute a fixpoint of the program in the Interval domain, applying widening.

b) Widening is used to ensure (fast) termination of fixpoint computation, however it may introduce unnecessary imprecision. Show this by providing an example program where interval analysis without widening terminates with a more precise result than analysis with widening.

Task 2 (Transformers for Pointer Analysis)**.** In the lecture, you have discussed transformers for flow-sensitive pointer analysis by examples. Provide formal definitions of sound abstract transformers for the following actions (be as precise as possible). Here, `x` and `y` are pointer variables pointing to objects with a field `f`.

a) Object creation: `x := alloc`

b) Pointer comparison (equality constraint): `x == y`

c) Pointer heap store: `x.f := y`

d) Pointer heap load: `x := y.f`

Task 3 (Alias Analysis)**.** Consider the following program, where objects created by **`alloc`** have a single field `f`. We would like to check whether `t` may alias with `c` at two different points in the program (see the two assertions).

Note: The assertions make sure that `t == c == null` is not counted as aliasing.

```
     h(int count) {
1:      c := alloc
2:      t := c
3:      i := 0
4:      while (i < count)
5:         n := alloc
6:         c.f := n
7:         c := n
           // assert  t != c || t == null
8:         i := i + 1
9:      c.f := t
10:     // assert t != c || t == null
     }
```

a) Perform flow-sensitive pointer analysis on the function `h` as introduced in the lecture using the transformers from task 2. Assume that uninitialized pointer variables may point to anything, including null. You don't have to perform numerical analysis (you can ignore the variables `count` and `i`).

b) Can you use the result to prove the two non-aliasing assertions? For assertions that cannot be proven, can you instead prove the negated claim `t == c && t != null`, meaning that `t` and `c` are aliases?

Exercise 12

Dynamic Alloy Models

Rigorous Software Engineering, ETH Zurich

Task 1 (Abstract Machine Idiom)**.** Consider the following Alloy model of a counter, where the predicate `inc` models the increment operation of the counter.

```
open util/integer

sig Counter {
    n: Int
}

pred inc[c, c’: Counter] {
    c’.n = c.n.add[Int[1]]
}
```

a) Use the abstract machine idiom discussed in the lecture to define the traces of the counter. For this, use the Alloy library `util/ordering`. Initially, the value of `n` must be 0. Define the initial state of the counter using a predicate `init[c: Counter]`.

   Hint: Alloy's integers are signed, have limited size, and can overflow. You can specify the size of the integers explicitly; e.g., `run show{} for 10 but 5 Int` generates instances of the predicate `show` with at most ten instances of every signature and 5-bit signed integers.

b) Express the following two invariants using Alloy assertions and check them.

   (i) The counter's value is always greater than or equal to zero.

   (ii) The counter's value never decreases.

   Hint: Invariant (ii) is a temporal invariant and you must hence check it for all pairs of successor states. You can use the predicate `lt[c,c’]` (provided by the Alloy `ordering` library) to check whether `c’` is a successor of `c`.

Task 2 (ETH Bus Driving—*from a previous exam*)**.** Recall the ETH bus Alloy model from exercise 11, task 2. In this task, we will consider the dynamic behavior of a single ETH bus, starting from the fixed Alloy model provided in the accompanying ZIP file.

Extend the Alloy model using appropriate predicates and facts to model the route of an ETHBus which:

- is initially (in state s_0) at the ETHBusStation Polyterrasse,
- if in s_i the bus was at an ETHBusStation, then in s_{i+1} the bus is no longer at any ETHBusStation (that is, it is driving between 2 stations), and
- if in s_i the bus was driving between 2 stations, then in s_{i+1} the bus arrives at the next ETHBusStation (according to the ETHBusStation at which it was in s_{i-1}).

Hint: Use `open util/ordering[ETHBus]` to get access to the following functions.

```
fun first: one ETHBus      /* returns the first element */
fun last: one ETHBus       /* returns the last element */
fun prev: ETHBus->ETHBus   /* returns a mapping from each element to its predecessor */
fun next: ETHBus->ETHBus   /* returns a mapping from each element to its successor */
```

Task 3 (Constraint Encoding)**.** Consider the Alloy model below.

```
sig Node {
    next: Node
}

assert demo {
    all n: Node | some m: Node | m.next = n
}
```

For the given model we have one constraint: each node must have exactly one next node.

The assertion `demo` holds iff it is satisfied in all instances that satisfy the model's constraints. To check whether the assertion holds, Alloy searches for a counter-example (i.e., an instance where the model's constraint is satisfied but the assertion `demo` is violated).

a) Encode the constraint and the assertion corresponding to `check demo for 1` into a boolean formula. Check if the formula has a satisfying assignment. If it is satisfiable, give a counter-example where the assertion is violated.

   Hint: Given two nodes n and m, introduce a boolean variable $x_{n,m}$ to denote $(n, m) \in$ `next`. Treat the universal and existential quantifiers as conjunction and disjunction, respectively.

b) Repeat subtask (a) for `check demo for 2`.

c) Extend the Alloy model as follows: Add a field `prev: Node` to the signature of `Node`, and add the fact `all n: Node | n.next.prev = n`. Now, repeat subtasks (a) and (b).

   Hint: The formula for the repeated subtask (b) is somewhat large. We don't expect you to manually decide satisfiability.

Exercise 08

Verifying Determinism

Rigorous Software Engineering, ETH Zurich

Task 1 (Non-deterministic Loops)**.** Consider the program below, where all variables are pointer variables pointing to objects with fields `p` and `next`. The condition $*$ of the **`while`** loop is non-deterministic, meaning that the loop may exit after an arbitrary (unknown) number of iterations. We are interested in proving the three assertions A1–A3.

```
1:    k1 := alloc
2:    k2 := alloc
3:    n0 := alloc
4:
5:    n := n0
6:    n.p := k1
7:    while ( * ) {
8:        c := alloc
9:        c.p := k2
10:       n.next := c
11:       n := c
12:       c := alloc
13:       c.p := k1
14:       n.next := c
15:       n := c
16:   }
17:   x := n0.next
18:
19:   // assert  x.next.next != x      (A1)
20:   // assert  x != n0               (A2)
21:   // assert  x.p != x.next.p       (A3)
```

You can assume that the fields `p` and `next` of freshly allocated objects are initialized to null. For this task, treat null as a special abstract object and explicitly track it in the analysis. You can ignore potential null pointer dereferences during your analysis.

a) Run flow-insensitive pointer analysis on the program above. Which of the three assertions A1–A3 (ignoring null pointer deferences) can you prove using the result of the analysis?

   Recall: Flow-insensitive pointer analysis maintains one points-to state S for the whole program (not per label). First, collect all assignment statements in the program (i.e., drop the control flow). Then, iteratively apply the abstract transformers for these statements to S, until you reach a fixed point. Start with the all-$\bot$ state and always join the result with the previous state.

b) Run flow-sensitive pointer analysis on the program above. Assume that uninitialized variables (e.g., `c` in line 4) may point to anything, including null. Which of the three assertions A1–A3 (ignoring null pointer dereferences) can you prove using the result of the analysis?

c) Which of the three assertions A1–A3 actually hold (in the concrete)?

Task 2 (Verifying Data-race Freedom)**.** Consider the program below, which sums up the first `n` $= 10$ elements of the `input` array using two threads (see **`fork`** and **`join`**). In this task, we would like to prove the absence of data-races (write-write and read-write conflicts) in order to ensure determinism.

```
        // allocation site of input: A
 1:     int sum(int[] input, boolean may_overwrite) {
 2:         int n = 10;
 3:
 4:         // prepare output
 5:         int[] output;
 6:         if (may_overwrite) {
 7:             output = input;
 8:         } else {
 9:             output = new int[n]; // allocation site: B
10:         }
11:
12:         // fill output
13:         output[0] = input[0];
14:         int half = n/2;
15:         fork {
16:             output[0] = input[0];
17:             for (int i = 1; i < half; i++) {
18:                 int val = output[i-1] + input[i];
19:                 output[i] = val;
20:             }
21:         }
22:         fork {
23:             output[half] = input[half];
24:             for (int j = half+1; j < n; j++) {
25:                 int val = output[j-1] + input[j];
26:                 output[j] = val;
27:             }
28:         }
29:         join;
30:
31:         return output[half-1] + output[n-1];
32:     }
```

a) Run flow-insensitive pointer analysis to determine points-to sets of the variables `input` and `output`. Assume that `input` was allocated at an allocation site A, and line 9 corresponds to allocation site B.

b) Perform abstract interpretation using the interval domain to determine the possible values of `n`, `half`, `i` and `j` inside the **`fork`** contexts.

c) Similarly as discussed in the lecture, verify that the two threads cannot introduce non-determinism due to data-races. Read-read conflicts can be treated as safe.

   Note: In case you are interested, you can find more details about the approach from the lecture in the original research paper [1].

References

[1] Martin Vechev, Eran Yahav, Raghavan Raman, and Vivek Sarkar. Automatic verification of determinism for structured parallel programs. In *International Static Analysis Symposium*, pages 455–471. Springer, 2010. Available at: `https://files.sri.inf.ethz.ch/website/papers/SAS%202010%20-%20Determinism.pdf`.

Exercise 06

Interval Analysis

Rigorous Software Engineering, ETH Zurich

Task 1 (Abstract Transformers)**.**

a) Consider the Interval domain $L^i = \{\bot_i\} \cup \{[a, b] \mid a, b \in \mathbb{Z} \cup \{-\infty, \infty\}, a \leq b\}$. Derive sound abstract interval transformers for the following actions, assuming `x` and `y` are variables. Your transformers should be as precise as possible.

   (i) Multiplication: `x * y`

   (ii) Absolute value: `|x|`

   (iii) Equality constraint: `x == y`

   Note: An abstract transformer is a function $(\text{Vars} \to L^i) \to (\text{Vars} \to L^i)$, where Vars is the set of variables. For arithmetic expressions such as (i) and (ii), it is sufficient to define the corresponding operation on L^i (i.e., a function $L^i \times L^i \to L^i$, resp. $L^i \to L^i$), which is naturally lifted to $(\text{Vars} \to L^i)$ in combination with assignments.

b) Recall the Sign domain $\{\bot, \top, +, -, 0\}$ from an earlier lecture. Derive a sound abstract transformer for strict inequality constraints (`x < y`) in the Sign domain.

Task 2 (Applying Interval Analysis)**.** Consider the program below. Using abstract interpretation in the Interval domain, we would like to prove that for all *positive and odd* inputs `x`, the value of `z` at the end of `f(x)` is at most 12.

```
    f(int x) {
1:     y := 2
2:     if x <= y
3:        z := 3 * x
       else
4:        z := y
5:     z := y * z
6:  }
```

a) Over-approximate the set of positive odd inputs by a suitable interval.

b) Compute the least fixpoint lfp of `f` in the Interval domain using your interval for `x`. Can you prove the desired property?

c) Give a concrete program trace which is in $\gamma^i(\text{lfp})$, but is not a valid trace of `f`.

Task 3 (Soundness and Precision)**.** You have already talked about precision in the lecture. Formally, an abstract transformer $F^\sharp$ for F in domain L is *precise* if and only if: $\forall z \in L.\ \gamma(F^\sharp(z)) \subseteq F(\gamma(z))$. Note the difference to soundness (flipped subset relation).

In this task, we consider alternative abstract transformers for addition and multiplication in the Interval domain over $\mathbb{Z}$. Prove or disprove the following statements.

a) The transformer $[a,b] +' [c,d] = [a+c,\ b+|d|]$ for addition is sound.

b) The transformer $+'$ from (a) is precise.

c) The transformer $[a,b] +'' [c,d] = [-\infty,\ a+b+d]$ for addition is sound.

d) The transformer $[a,b] +''' 5 = [a+5,\ b+5]$ for adding the constant 5 is precise.

e) The transformer $[a,b] *' 2 = [2a, 2b]$ for multiplication by the constant 2 is precise.

Exercise 01

Documentation and Contracts

Rigorous Software Engineering, ETH Zurich

Task 1 (Documentation). Your colleague implemented a graph library, where nodes and edges are represented as objects of type `Node` and `Edge` (see below). Being a considerate developer, she provided javadoc documentation for `Node`.

```
/**
 * Anything which can be treated as a node in a graph.
 */
public interface Node {
    /**
     * @return unique non-null String identifying the node
     */
    public String getId();
}

public class Edge {
    Node from;
    Node to;
    int distance;

    public Edge(Node from, Node to, int distance) { ... }
}
```

As part of a navigation system, you have implemented the following class for repeatedly querying shortest paths in a fixed graph.

```
public class ShortestPathsManager {
    private List<Node> nodes;
    private List<Edge> edges;
    private Map<String, List<Node>> cache;

    public ShortestPathsManager(List<Node> nodes, List<Edge> edges) { ... }

    public List<Node> getShortestPath(Node source, Node target) {
        String key = source.getId() + "-" + target.getId();
        if !cache.contains(key) {
            List<Node> path = computeShortestPath(source, target);
            cache.put(key, path);
        }
        return cache.get(key);
    }

    private List<Node> computeShortestPath(Node source, Node target) { ... }
}
```

a) The current implementation of `getShortestPath` implicitly relies on a hidden assumption on the nodes' implementations. Create an implementation of `Node` which respects the documentation, but leads to undesired behavior of `getShortestPath`.

b) How can you fix the implementation of `getShortestPath` such that it behaves correctly for all implementations of `Node` respecting the documentation?

Even though `ShortestPathsManager` does not use code comments for documentation, the identifier names and type declarations already allow for some understanding of the code.

c) Consider the method `getShortestPath`. Create a list of properties which are currently insufficiently documented but should be part of the method's interface documentation.

d) Based on your list, write suitable documentation (in javadoc comment format[1]) for `getShortestPath`, assuming a reasonable implementation of `computeShortestPath`.

Task 2 (Contracts)**.** In the lecture, you have seen contracts for code documentation. In this exercise, we consider three kinds of contracts, exemplified in the code below. The class `RootMachine` computes square roots and counts how many times this happens.

```
public class RootMachine {
    // invariant: cnt >= 0
    public int cnt = 0;

    // requires: x >= 0
    // ensures: result >= 0 && result*result == x && cnt == old(cnt)+1
    public float getRoot(float x) { ... }
}
```

- *Preconditions* (`// requires`). The conditions the caller of a method must fulfill. For example, to call `getRoot(x)`, the caller has to provide a value `x >= 0`.
- *Postconditions* (`// ensures`). The guarantees provided to the caller after executing the method, given that the caller satisfied the precondition. Within postconditions, one can use `old(...)` to refer to the state *before* the method is executed, and `result` to indicate the return value. For example, the method `getRoot(x)` returns a nonnegative `result` such that `result*result == x`, and increments the counter `cnt`.

[1] `https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html`

- *Invariants* (`// invariant`). The conditions that all instances of a class must satisfy while being observable by any client. For example, the counter `cnt` is guaranteed to be non-negative.

Contracts must be pure expressions (i.e., no side-effects). Note that contracts are part of the *specification*, not the implementation, so they cannot modify the program state.

Consider a simple game, in which a player can move left, right, up, and down on a (non-empty) `n`-by-`n` board. The game's Java implementation is given below.

```java
public class Player {
    private int x, y;
    public final int n;

    public Player(int x, int y, int n) {
        this.x = x;
        this.y = y;
        this.n = n;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void moveLeft() {
        x = x - 1;
        if (x < 0) {
            x = n - 1;
        }
    }

    ...
}
```

a) Write a suitable invariant for the class `Player`.

b) Write a suitable precondition for the constructor of `Player`.

c) Write a suitable postcondition for the method `moveLeft`.

Task 3 (C# Code Contracts)**.** C# Code Contracts allow for both compile-time static analysis and runtime dynamic checks of contracts. Consider the following C# class.

```
public class Bag {
    private int[] elems;
    private int count;

    public Bag(int[] initialElements) {
        this.count = initialElements.Length;
        int[] e = new int[initialElements.Length];
        initialElements.CopyTo(e, 0);
        this.elems = e;
    }

    public Bag(int[] initialElements, int start, int howMany) {
        this.count = howMany;
        int[] e = new int[howMany];
        Array.Copy(initialElements, start, e, 0, howMany);
        this.elems = e;
    }

    public int Count() {
        return count;
    }

    public int[] GetElements() {
        return elems;
    }

    public int RemoveMin() {
        int m = System.Int32.MaxValue;
        int mindex = 0;
        for (int i = 0; i < count; i++) {
            if (elems[i] < m) {
                mindex = i;
                m = elems[i];
            }
        }
        count--;
        elems[mindex] = elems[count];
        return m;
    }

    public void Add(int x) {
        if (count == elems.Length) {
            int[] b = new int[2*elems.Length];
            Array.Copy(elems, 0, b, 0, elems.Length);
            elems = b;
        }
        elems[count] = x;
        count++;
    }
}
```

Find class invariants and preconditions for all methods of the class `Bag`, and postconditions for the method `Add`. Express them using the syntax of C#'s Code Contracts: [2]

- At the beginning of a method, `Contract.Requires(expr)` can be used to denote a precondition. Here, `expr` must be a pure boolean C# expression referring only to fields and pure methods with greater or equal visibility than the method. For example, in the precondition of a **`public`** method, you can not refer to **`private`** fields. Similarly, `Contract.Ensures(expr)` can be used to denote a postcondition.
- Class invariants are declared in a special contract invariant method (see below).

```
[ContractInvariantMethod]
private void ObjectInvariant() {
    Contract.Invariant(expr);
    ...
}
```

- `Contract.ForAll(lower, upper, i => expr)` can be used to express that `expr` holds for all integers `i` (`i` can occur in `expr`) from `lower` (inclusive) to `upper` (exclusive).
- In a postcondition, `Contract.OldValue(expr)` can be used to refer to the value of `expr` before the execution of the method.
- Methods without side effects (e.g., getters) can be marked as `[Pure]` in the line before the declaration. Only pure methods can be used in contract expressions.

[2] If you are interested, you can find more details here: `https://docs.microsoft.com/en-us/dotnet/framework/debug-trace-profile/code-contracts`