CS130: Software Engineering

Homework 2

Winter 2025

Jan Kasen

GitHub Link:

https://github.com/jankasen/CS130-HW2

Due Mar 9 2025 11:55pm

Problem 1

Part 1

```
def process_age(age):
    if age < 0:
        return "Invalid"
    if age < 18:
        return "Minor"
    if age > 65:
        return "Senior"
    if age < 0: # Unreachable
        return "Error"
    return "Adult"
```

(a) Write SMT formulas for each path.

SMT Formulas

```
(declare-const age Int)
(assert (< age 0)) ; Invalid
(check-sat)
(get-model)
```

```
(declare-const age Int)
(assert (>= age 0)) ; Not < 0
(assert (< age 18)) ; Minor
(check-sat)
(get-model)
```

```
(declare-const age Int)
(assert (>= age 0)) ; Not < 0
(assert (>= age 18)) ; Not < 18
(assert (> age 65)) ; Senior
(check-sat)
(get-model)
```

```
(declare-const age Int)
(assert (>= age 0)) ; Not < 0
(assert (>= age 18)) ; Not < 18
(assert (<= age 65)) ; Not > 65
(assert (< age 0)) ; Error
(check-sat)
```

```
(declare-const age Int)
(assert (>= age 0)) ; Not < 0
(assert (>= age 18)) ; Not < 18
(assert (<= age 65)) ; Not > 65
(assert (>= age 0)) ; Not < 0, Adult
(check-sat)
(get-model)
```

(b) Write one test input for each path.

```
"Invalid": -1
"Minor": 8
"Senior": 73
"Error": Unreachable
"Adult": 22
```

(c) What is the minimum set of test cases needed for 100% branch coverage?

Four test cases at minimum. One for each branch with "Error" branch being unreachable

Part 2

```
# Assume input array has array size 3
def sum_until_negative(numbers):
    total = 0
```

```
    i = 0
    while i < len(numbers) and numbers[i] > 0:
        total += numbers[i]
        i += 1
    return total
```

(a) Write SMT formulas for each path.

```
# Loop unrolling for array size 3
def sum_until_negative(n_0, n_1, n_2):
    total = 0
    # Iteration 1
    if n_0 > 0:
        total += n_0
        # Iteration 2
        if n_1 > 0:
            total += n_1
            # Iteration 3
            if n_2 > 0:
                total += n_2
    return total
```

SMT Formulas

```
(declare-const n_0 Int)
(declare-const n_1 Int)
(declare-const n_2 Int)
(assert (< n_0 0))
(check-sat)
(get-model)
```

```
(declare-const n_0 Int)
(declare-const n_1 Int)
(declare-const n_2 Int)
(assert (>= n_0 0))
(assert (< n_1 0))
(check-sat)
(get-model)
```

```
(declare-const n_0 Int)
(declare-const n_1 Int)
(declare-const n_2 Int)
(assert (>= n_0 0))
(assert (>= n_1 0))
(assert (< n_2 0))
(check-sat)
(get-model)
```

```
(declare-const n_0 Int)
(declare-const n_1 Int)
(declare-const n_2 Int)
(assert (>= n_0 0))
(assert (>= n_1 0))
(assert (>= n_2 0))
(check-sat)
(get-model)
```

(b) Write one test input for each path.

```
numbers = [-1, 2, 3]
numbers = [1, -2, 3]
numbers = [1, 2, -3]
numbers = [1, 2, 3]
```

(c) What is the minimum set of test cases needed for 100% branch coverage?

Four test cases, one for each path.

Part 3

```
def classify_sequence(numbers) :
    if len(numbers) == 0:
        return "Empty"

    count = 0
    i = 0
    while i < len(numbers) and i < 5: # Process at most 5 numbers
        if numbers [i] > 0:
            count += 1
```

```
        i += 1
    if count == 0:
        return "AllNonPositive"
    elif count == i:
        return "AllPositive"
    else:
        return "Mixed"
```

(a) For an input array of size 3, identify major paths through the code.

- Array of size 3 will bypass the empty check
- The while loop iterates at most 5 times, 3 < 5 so we are fine
- Major paths
    1. All numbers are non-positive
    2. All numbers are positive
    3. The numbers are a mix of positive and non-positive

(b) Write SMT formulas for achieving each return value. Use predicate logic + propositional logic if needed:

- "Empty"
- "AllNonPositive"
- "AllPositive"
- "Mixed"

```
# Loop unrolling for array size 3
def classify_sequence(n_0, n_1, n_2) :
    count = 0
    if n_0 > 0:
        count += 1
    if n_1 > 0:
        count += 1
    if n_2 > 0:
        count += 1

    if count == 0:
        return "AllNonPositive"
    elif count == 3:
        return "AllPositive"
```

```
    else:
        return "Mixed"
```

SMT Formulas

```
; For an array of size 3, "Empty" is unsatisfiable
(assert false)
(check-sat)
```

```
(declare-const n_0 Int)
(declare-const n_1 Int)
(declare-const n_2 Int)

(assert (and (<= n_0 0) (<= n_1 0) (<= n_2 0))) ; AllNonPositive
(check-sat)
(get-model)
```

```
(declare-const n_0 Int)
(declare-const n_1 Int)
(declare-const n_2 Int)

(assert (and (> n_0 0) (> n_1 0) (> n_2 0))) ; AllPositive
(check-sat)
(get-model)
```

```
(declare-const n_0 Int)
(declare-const n_1 Int)
(declare-const n_2 Int)

(assert (and                                ; Mixed
  (or (> n_0 0) (> n_1 0) (> n_2 0))        ; At least one positive
  (or (<= n_0 0) (<= n_1 0) (<= n_2 0))     ; At least one non-positive
))
(check-sat)
(get-model)
```

(c) Write a python code to implement your "All Positive" SMT constraint using Z3 and solve for the answer(ref 1,2).

```
from z3 import *

n0 = Int('n0')
n1 = Int('n1')
n2 = Int('n2')

solver = Solver()

solver.add(n0 > 0, n1 > 0, n2 > 0)

if solver.check() == sat:
    model = solver.model()
    print("Satisfiable model:")
    print(model)
else:
    print("Unsatisfiable")
```

(d) Write test inputs for each case.

```
AllNonPositive: [-1, -1, -1]
AllPositive: [1, 1, 1]
Mixed: [1, -1, 1]
```

(e) Now assume the input array has no size limit, What special cases should be tested? (Think about boundary conditions)

- Empty array
- Array with fewer than 5 elements
- Array with exactly 5 elements
- Array with more than 5 elements
- Array values at exactly zero

Problem 2

Your company runs a cloud service that processes user requests. The system monitors latency (response time in ms) and failure rate (percentage of failed requests), which follow a Poisson distribution. If these metrics exceed defined thresholds, an alert is issued.

Alerts have different severities:

- **P0 (Critical):** Latency > 2000ms or Failure Rate > 10%
- **P1 (Major):** Latency > 1000ms or Failure Rate > 5%
- **P2 (Minor):** Latency > 500ms or Failure Rate > 2%

Alerts must repeat notifications at:

- **P0:** Every 2 hours
- **P1:** Every 12 hours
- **P2:** Every 48 hours

If an alert condition resolves, it should be removed from active alerts.

Requirements

Simulate Incoming Metrics:

- Generate latency and failure rate data via simulation.
- Use Poisson distribution with manual tuning (to show persistent error behavior).
- The flaky metric can be transient (disappear in a few minutes) or persistent (last for a few days).
- Generate new metrics every 5 minutes.

Trigger Alerts Based on Thresholds:

- Classify alerts as P0, P1, or P2 based on the metric values.
- If things are getting worse, priority shall go up.
- Priority will not downlevel.

Notification Handling:

- Send an initial alert notification when an issue is detected.
- Resend unresolved alerts based on their priority timing (P0—2h, P1—12h, P2—48h).
- An email shall be sent out to a team address (mock this behavior by printing).
- An email shall be sent out to the team's skip-level boss if not resolved in 5x of the alert active time (mock this behavior by printing).

Alert Resolution:

- If this is a persistent issue, mock the resolving operations by printing some PRs are merged.
- If a metric returns to normal, mark the alert as resolved.

Logging & Reporting:

- Maintain a log of active alerts with timestamps.
- Maintain a log of the current system status every 5 minutes.
- Delete stale logs after 90 days.

Sample Output on a Running Console

```
[2025-02-23 12:00:05] Latency: 1200ms, Failure Rate: 6.2% -> P1 Alert
Triggered!
[2025-02-23 14:00:05] ALERT: Resending P1 alert (Still unresolved)
[2025-02-23 16:00:05] ALERT: Resending P1 alert (Still unresolved)
......
[2025-02-25 08:00:00] INFO: Commit 3a4b6c submitted
[2025-02-25 08:05:00] INFO: Commit 3b4b6f submitted
......
[2025-02-25 10:00:05] ALERT: Resending P1 alert (Still unresolved)
......
[2025-02-26 16:05:00] INFO: Commit 3c4b6a submitted
......
[2025-02-27 12:00:05] INFO: Latency Normalized. Resolving P1 alert.
```