Software Testing and Quality Assurance

CSC510: Software Engineering

Tim Menzies

prof, cs, **ncstate**, usa
acm,ieee,ase fellow; eic ASEj
timm@ieee.org
http://timm.fyi

October 6, 2025

Introduction: Why Test?

- Testing checks code correctness and prevents regressions
- Good manners for team: don't commit breaking changes
- Multi-stakeholder systems have competing requirements
- Functional vs non-functional requirements
- "Program testing shows presence of bugs, hopelessly inadequate for showing absence" - Dijkstra

Discussion: Toronto CS department information system - good if parents can track children, but good if students maintain privacy. How do you test for both?

The V-Diagram: Software Lifecycle

- Planning down to coding
  - Planning = requirements → design → implementation
- Coding across to testing
- Then upwards to more and more complex tesing
  - Testing = unit test→ integration → system test → acceptance testing

- Verification: “building the system right”
- Validation: “building the right system”

- Brooks, *Mythical Man Month*: 1/3 planning, 1/6 coding, 1/4 unit test, 1/4 system test
- More time in testing than coding

Fault vs Failure (Fenton & Neil)

- **Fault**: incorrect step/process/definition in code
- **Failure**: when something actually goes wrong
- Key insight: pre-release faults != post-release failures

Discussion Question: How can a system with *few* pre-release faults have *many* post-release failures? And vice versa?

- Answer: Untested systems show few faults (none found!) but fail heavily in production
- Well-tested systems expose many faults pre-release, resulting in fewer failures post-release
- Usage patterns, defect detection capability, design effort all matter

Testing Types: Concrete Examples

- **Unit testing**: Test `calculateDiscount(price, percentage)` function in isolation
- **Integration testing**: Test checkout service calling payment gateway API
- **System testing**: End-to-end purchase flow from cart to confirmation email
- **Acceptance testing**: Customer validates system meets contract requirements
- **Alpha testing**: Internal QA team uses prototype before release
- **Beta testing**: 1000 external users try new mobile app version
- **A/B testgs**: split some population in half
  - Grpup A gets "it"
  - Group B does not
  - Collect data, apply stats to check if "it" worked.
- **Regression testing**: Re-run all tests after adding shopping cart feature
  - warning: slow, Voluminous

```
US_PHONE_GRAMMAR = {
    "<start>": ["<phone-number>"],
    "<phone-number>": ["(<area>)<exchange>-<line>"],
    "<area>": ["<lead-digit><digit><digit>"],
    "<exchange>": ["<lead-digit><digit><digit>"],
    "<line>": ["<digit><digit><digit><digit>"],
    "<lead-digit>": ["2","3","4","5","6","7","8","9"],
    "<digit>": ["0","1","2","3","4","5","6","7","8","9"]
}
```

Generates: `(692)449-5179`, `(519)230-7422`, etc.

Black-Box: Random Generation from Grammars

```python
import random

def generate(grammar, rule, depth=3):
    if depth == 0 or rule not in grammar:
        return random.choice(grammar.get(rule, [rule]))
    expansion = random.choice(grammar[rule])
    return "".join(generate(grammar, r, depth-1)
                   for r in expansion)

# Generate 5 random phone numbers
for _ in range(5):
    print(generate(US_PHONE_GRAMMAR, "<start>"))
```

Uniform random sampling from grammar rules.

Black-Box: Manual Weighting

```python
# Weight grammar to prefer certain area codes
WEIGHTED_GRAMMAR = {
    "<start>": ["<phone-number>"],
    "<phone-number>": ["(<area>)<exchange>-<line>"],
    "<area>": [
        ("919", 0.5),           # 50% weight - local area
        ("212", 0.3),           # 30% weight - NYC
        ("<random-area>", 0.2)  # 20% other
    ],
    "<random-area>": ["<lead-digit><digit><digit>"],
    # ... rest of grammar
}

def weighted_choice(options):
    choices, weights = zip(*options)
    return random.choices(choices, weights)[0]
```

Black-Box: Automatic Reweighting (Coverage-Guided)

```python
def coverage_guided_fuzzing(grammar, test_fn, iterations=100):
    covered_branches = set()

    for _ in range(iterations):
        # Generate input
        test_input = generate(grammar, "<start>")

        # Run test, track coverage
        new_branches = test_fn(test_input)

        # Increase weight for rules that hit new branches
        if new_branches - covered_branches:
            increase_weight_for(rules_used_in(test_input))
            covered_branches.update(new_branches)
```

Automatically biases generation toward unexplored code paths.

Reading documentation to infer state transitions:

```
Elevator Door States (inferred from docs):
- CLOSED -> OPENING (on button press)
- OPENING -> OPEN (after 2 seconds)
- OPEN -> CLOSING (after 5 second timeout OR button)
- CLOSING -> OPEN (if obstruction detected)
- CLOSING -> CLOSED (after 2 seconds)
```

Discussion: Draw a state diagram. What test cases cover all transitions? What if door stays OPEN between floors 3 and 7 - is this a violation?

Coverage Metrics: The Differences

```python
def example(x, y):
    a = x + y           # Line
    if x > 0:           # Line 2
        b = a * 2       # Line 3
    else:
        b = a * 3       # Line 4

    if y > 0:           # Line 5
        c = b + a       # Line 6
    else:
        c = b - a       # Line 7

    return c            # Line 8
```

Coverage Examples

Test 1: `example(1, 1)` - Lines covered: 1,2,3,5,6,8 (6/8 = 75% line coverage) - Branches: T,T (2/4 = 50% branch coverage)

Test 2: `example(-1, -1)` - Lines: 1,2,4,5,7,8 (6/8 = 75% line coverage) - Branches: F,F (2/4 = 50% branch coverage)

Both tests: 100% line, 100% branch

But: Never tested `x>0` AND `y<0` interaction (du-path from line 3 to line 7 uncovered)

Problems with Coverage Metrics

- Gopinath et al. (2014): Statement coverage best predicts mutation kills, not branch/path
- Inozemtseva & Holmes (2014): Coverage not strongly correlated with test suite effectiveness
- Kochhar et al. (2015): Real bugs in large systems - coverage helps but insufficient

Key findings:

- 100% coverage still misses logic errors
- Concurrency bugs evade coverage metrics
- Complex component interactions not captured

Mutation Testing: Operators

Original Code:

```
def calculate_discount(price, rate):
    if price > 100:
        discount = price * rate
        return price - discount
    return price
```

AOR (Arithmetic): `discount = price / rate` (changed * to /)

ROR (Relational): `if price >= 100:` (changed > to >=)

CR (Constant): `if price > 50:` (changed 100 to 50)

Mutation Testing: Genetic Programming

Beyond point mutations - swap entire code blocks:

```python
# Original
def process(data):
    if validate(data):
        result = transform(data)
        return result
    return None

# Mutant (swapped if/else)
def process(data):
    if validate(data):
        return None
    result = transform(data)
    return result
```

Crossover: Take branches from two passing variants, combine them, see if combined version still passes.

Killed mutant = the test suite detects the fault - i.e. at least one test fails when run against the mutated program.

Mutation Score Interpretation

Mutation Score = (Killed Mutants / Total Mutants) * 100

Example: 100 mutants generated, 70 killed -> 70% score

What survives?

- Equivalent mutants (semantically identical)
- Untested edge cases
- Weak test assertions

Better indicator than coverage: test *quality* not just *quantity*.

Regression Testing & Test Prioritization

- Elbaum, Rothermel, Penix (FSE 2014): "Techniques for improving regression testing in continuous integration"
- Large test suites: 3-5 hours (cloud) to 30 hours (local)
- Slow feedback kills CI/CD agility

APFD (Average Percentage of Faults Detected):

- Measures how quickly tests find faults
- Higher APFD = faults found earlier
- Cost-aware variant: APFDc weights by test execution time

Elbaum's Prioritization Heuristic (2014)

Study on continuous integration environments:

Prioritize tests that:

1. Failed recently
2. Haven't been tested for a while (long time since last run)
3. Are new functionality

Result: For very large suites, catches 50% of failures within first hour (vs 3-5 hours for full run)

Works well when test history available and failures cluster.

Ling, Agrawal, Menzies (TSE 2022): "How Different is Test Case Prioritization for Open and Closed Source Projects?"

Strategies compared:

- **A2 (Optimal/Omniscient):** Actually knows where bugs are
  - all results baselined against A2
- **D1 (Diversity):** Maximize coverage of different code regions
- **B1 (History - fail rate):** Run tests with highest historical failure rate
- **B3 (History - recency):** Run tests that failed most recently

Open vs Closed Results

Closed-source projects:

- D1 performs as well as A2
- Diversity-based selection optimal

Open-source projects:

- B1, B3 perform as well as A2
- History-based selection optimal

Key insight: Test case prioritization strategies that work best for industrial closed-source can work *worse* for open-source (and vice versa)

Context matters: release cadence, developer distribution, test characteristics differ.

All-Pairs Testing Example

Five inputs: (2, 2, 2, 7, 10) values each

Full combinatorial: 2×2×2×7×10 = 560 tests

All-pairs generates only 68 tests:

```
(2,2,1,1,1) (2,1,2,2,2) (1,2,2,3,3) (1,1,1,4,4)
(2,2,2,7,5) (2,2,2,6,6) (2,2,2,5,7) (2,2,2,4,8)
...
```

Every pair of values appears together in at least one test.

Dramatic reduction with high coverage of 2-way interactions.

Zeller's Implementation (Fig 1 from TSE 2002): Reduce 'inp' to a 1-minimal failing subset, using the outcome of 'test(inp, *test_args)', which should be 'PASS', 'FAIL', or 'UNRESOLVED'.

Systematically shrink input until nothing smaller still fails.

```
def ddmin(test: Callable, inp: Sequence[Any], *test_args) -> Sequence:
    assert test(inp, *test_args) != PASS
    n = 2  # Initial granularity
    while len(inp) >= 2:
        subset_length: int = int(len(inp) / n)
        some_complement_is_failing: bool = False
        start = 0
        while start < len(inp):
            # Cut out inp[start:start + subset_length]
            complement: Sequence[Any] = inp[:start] + inp[start + subset_length:]
            if test(complement, *test_args) == FAIL:
                inp = complement
                n = max(n - 1, 2)
                some_complement_is_failing = True
                break
            start += subset_length
        if not some_complement_is_failing:
            if n == len(inp): break
            n = min(n * 2, len(inp))
    return inp
```

Zeller's Test Function (Fig 2 from TSE 2002): Run collected function with 'args'. Return PASS if no exception occurred, FAIL if the collected exception occurred, UNRESOLVED if some other exception occurred.

```python
def test(self, args: Dict[str, Any]) -> str:
    try:
        result = self.call(args)
    except Exception as exc:
        self.last_exception = exc
        if (type(exc) == type(self.exception()) and
            str(exc) == str(self.exception())):
            return FAIL
        else:
            return UNRESOLVED  # Some other failure
    self.last_result = result
    return PASS
```

Key: Compare exception **type AND message** to distinguish target failure from syntax errors.

Delta Debugging: Complete SQL Example

Initial Scenario: Grammar-generated SQL query crashes server

```sql
-- Original 847-character query (simplified for slides)
SELECT users.id, users.name, users.email, products.title
FROM users LEFT JOIN products ON users.id = products.user_id
WHERE users.active = 1 OR 1=1 AND users.created > '2020-01-01'
  AND products.price < 100 OR products.stock > 0
ORDER BY users.name, products.title LIMIT 50 OFFSET 10;
```

Crash signature: `SIGSEGV at address 0x0000, stack trace: query_parser.c:1247`

Delta Debugging: Step-by-Step

Initial setup:

- Baseline: empty string (passes - no crash)
- Full input: 847 chars (fails with SIGSEGV)
- Goal: find minimal subset that crashes

Iteration 1: Try removing first half (424 chars)

```sql
-- Remove chars 0-423, keep 424-846
WHERE users.active = 1 OR 1=1 AND users.created > '2020-01-01'
  AND products.price < 100 OR products.stock > 0
ORDER BY users.name, products.title LIMIT 50 OFFSET 10;
```

Result: UNRESOLVED (syntax error: “WHERE without FROM”) - ddmin tries other half

Delta Debugging: Iteration 2

Try removing second half (chars 424-846)

```
SELECT users.id, users.name, users.email, products.title
FROM users LEFT JOIN products ON users.id = products.user_id
WHERE users.active = 1 OR 1=1 AND users.created > '2020-01-01'
```

Result: FAIL (SIGSEGV at query_parser.c:1247) - Same crash! Keep this smaller input (423 chars) - Reduced by 50%

Delta Debugging: Iteration 3

Now work with 423-char input, try removing first half

```
WHERE users.active = 1 OR 1=1 AND users.created > '2020-01-01'
```

Result: UNRESOLVED (syntax error)

Try removing second half:

```
SELECT users.id, users.name, users.email, products.title
FROM users LEFT JOIN products ON users.id = products.user_id
```

Result: PASS (no crash) - Neither half crashes alone –> increase granularity

Try quarters instead. Remove first quarter:

```
name, users.email, products.title
FROM users LEFT JOIN products ON users.id = products.user_id
WHERE users.active = 1 OR 1=1 AND users.created > '2020-01-01'
```

Result: UNRESOLVED

Try removing second quarter:

```
SELECT users.id, users.
FROM users LEFT JOIN products ON users.id = products.user_id
WHERE users.active = 1 OR 1=1 AND users.created > '2020-01-01'
```

Result: UNRESOLVED

Delta Debugging: Iterations 5-8

After trying various quarters, narrow down to the WHERE clause:

```
SELECT * FROM users WHERE 1=1 OR users.active = 1
```

Result: FAIL (crashes!) - Now 54 characters

Continue reducing...

```
SELECT * FROM users WHERE 1=1 OR 1
```

Result: FAIL - 35 characters

```
SELECT * FROM users WHERE 1=1
```

Result: FAIL - 31 characters

Delta Debugging: Final Result

After ~15 iterations (each taking ~5 seconds to test):

```
SELECT * FROM users WHERE 1=1
```

Final minimal input:

- 31 characters (from 847)
- 96% reduction
- Still produces identical crash signature
- 1-minimal: removing any single character causes PASS or UNRESOLVED

Key insight revealed: Server crashes on trivial tautology `WHERE 1=1`

- Development team can no longer dismiss as "unrealistic"
- Bug is obviously critical

Reference: Zeller & Hildebrandt (TSE 2002): "Simplifying and isolating failure-inducing input"

Fault Localization: Tarantula

```python
def suspiciousness(passed, failed, total_passed, total_failed):
    """Tarantula heuristic"""
    if passed + failed == 0:
        return 0

    passed_ratio = passed / total_passed if total_passed > 0 else 0
    failed_ratio = failed / total_failed if total_failed > 0 else 0

    return failed_ratio / (passed_ratio + failed_ratio)
```

For each statement: track how many passing/failing tests execute it.

Higher suspiciousness → more likely to contain fault.

Fault Localization: Ochiai

```python
def ochiai(passed, failed, total_failed):
    """Ochiai heuristic - often more effective"""
    import math
    if failed == 0:
        return 0

    return failed / math.sqrt(total_failed * (failed + passed))
```

Alternative heuristics: - **Jaccard:** `failed / (total_failed + passed)` - **Dstar:** `failed^2 / (passed + (total_failed - failed))`

Reference: Jones, Harrold, Stasko (ICSE 2002): "Visualization of test information to assist fault localization"

Automated Program Repair: GenProg

1. Run tests, identify failing tests
2. Use fault localization (e.g., Tarantula) to find suspicious code regions
3. Apply genetic programming *only* in suspicious regions:
   - **Mutation:** Change operator, swap statement, delete line
   - **Crossover:** Swap code blocks between variants
4. Evaluate: does mutated code pass all tests?
5. Repeat until repair found or budget exhausted

Key insight: Don't search entire program space - localize first, then repair.

Reference: Le Goues et al. (TSE 2012): "GenProg: A generic method for automatic software repair"

Metamorphic Testing Example

Testing a hotel booking site without complete specification:

Query 1: “Hotels in Sydney” $\rightarrow$ 1,671 results

Metamorphic Relation: Filtering should not increase results

Query 2: “Hotels in Sydney, 4-star or higher” $\rightarrow$ 423 results

Check: results(Q2) within results(Q1)

If Q2 returned 1,800 results ==> **BUG DETECTED**

Reference: Zhou, Tse, Witheridge (TSE 2019): “Metamorphic Robustness Testing: Exposing Hidden Defects in Citation Statistics”

Symbolic Execution: BigTest

Problem: Testing big data analytics on gigabytes of data

Naive approach: Need tests for all data combinations

BigTest insight: Only need to cover code branches, not data combinations

```python
def analyze_sales(records):
    total = 0
    for record in records:
        if record.amount > 1000:      # Branch 1
            total += record.amount * 0.9
        else:                          # Branch 2
            total += record.amount
    return total
```

Need only 2 test inputs: one with amount>1000, one with amount <= 1000.

Reference: Marinescu & Cadar (ICSE 2013)

Problem: Which Linux kernel configurations are valid?

Feature model with 4000 variables, 300,000 constraints.

Solution: Express as CNF (Conjunctive Normal Form), use SAT solver

```
import pycosat

# Example: 5 variables, 3 constraints
cnf = [[1, -5, 4],          # x1 OR NOT x5 OR x4
       [-1, 5, 3, 4],       # NOT x1 OR x5 OR x3 OR x4
       [-3, -4]]            # NOT x3 OR NOT x4

solution = pycosat.solve(cnf)
print(solution)  # [1, -2, -3, -4, 5]
```

Each solution = valid configuration. Can enumerate all solutions via `itersolve`.

Minimal install problem:

```
# Add costs to each package
costs = {1: 100, 2: 50, 3: 200, 4: 150, 5: 75}

solutions = []
for sol in pycosat.itersolve(cnf):
    cost = sum(costs[abs(x)] for x in sol if x > 0)
    solutions.append((cost, sol))

minimal = min(solutions, key=lambda x: x[0])
print(f"Cheapest install: {minimal}")
```

User preferences: If user dislikes solution, negate it and add as constraint. Future solutions avoid that configuration.

Reference: Lerner et al. (FSE 2008): "Opium: Optimal Package Install/Uninstall Manager"

AWS AuthV2: Formal Verification at Scale

Challenge: 1 billion requests/second authorization engine. Changes risk security/availability.

Approach (4-year effort):

1. Reverse-engineer formal spec from AuthV1 (Java) in Dafny
2. Write verified implementation in Dafny, prove correct vs spec
3. Custom compiler to idiomatic Java (DafnyLite)
4. Shadow testing: 10ˆ15 production requests

Result: 3x faster, proved correct, increased development agility

Key lesson: Rewrite in verification-aware language beats verifying legacy code.

AuthV2: Why Shadow Testing?

Question: If code is formally verified, why test with 10ˆ15 samples?

Answer:

- Formal proof: implementation matches **specification**
- Testing: specification matches **intended behavior**
- Found 7 specification errors missed by proof
- Specification != Requirements

Proof connects impl ==> spec. Testing connects spec ==> reality.

Reference: Amazon Science (2024): “Formally verified cloud-scale authorization”

ARIMA Forecasting: Issues ==> Bugs

Study (832 projects): Can raw issue counts predict bug/enhancement workload?

ARIMA model: AutoRegressive Integrated Moving Average

- AR: current value depends on past values
- I: differencing to make stationary
- MA: current value depends on past errors

Methodology:

- Rolling window: train on 20 weeks, forecast 4 weeks
- Slide forward by 1 week, repeat
- Metric: MAE (Mean Absolute Error)

ARIMA Results: Surprising Finding

RQ1: Do issues/bugs/enhancements show temporal trends? **YES** (low MAE)

RQ2: Are trends correlated? **YES** (moderate positive correlation)

RQ3: Can issues forecast bugs/enhancements? **YES** (low MAE)

RQ4: As accurate as using bug history? **YES** (statistically similar)

Practical implication: Skip labeling effort. Use easily collected issue counts to forecast workload.

Reference: Krishna et al. (arXiv 2017): "What is the Connection Between Issues, Bugs, and Enhancements? (Lessons Learned from 800+ Software Projects)"

TERMINATOR: Active Learning for TCP

(TCP= test case priorization.)

Problem: UI test suites take 3-30 hours. Black-box only (no coverage).

Approach: Frame as Total Recall problem

- Find all failures (positives) with minimum cost (running tests)
- Use Active Learning with SVM

Features:

- Text: TF from test descriptions
- History: past pass/fail/skip rates
- Hybrid: text + history (best)

TERMINATOR Algorithm

1. Start with empty executed set L, empty failed set LR
2. While tests remain:
   - If |LR| < 30: Uncertainty sampling (near SVM boundary)
   - If |LR| >= 30: Certainty sampling (confident failures)
   - Select batch of 10 tests
   - Execute batch, update L and LR
   - If failures found: train/update SVM
   - Use aggressive undersampling for balance
3. Continue until all tests run or budget exhausted

TERMINATOR Results

Performance:

- Hybrid features achieved ~75% of optimal (A2)
- Found 60% of failures in 20% of test time
- Next best method: ~30% failures in 20% time
- Computational overhead: 0.33% of total test time

Key insight: Dynamic adaptation via active learning beats static prioritization.

Comparison: Simple history methods beat complex text-only methods, but active learning hybrid beats both.

Reference: Bertolino et al. (FSE 2020): “Learning-to-rank vs ranking-to-learn”

Non-Functional Requirements: Testing Challenges

How do you test for:

- **Maintainability?** –> Years of observation needed
- **Usability?** –> Subjective, user studies required
- **Security?** –> Adversarial thinking, penetration testing
- **Performance?** –> Load testing, profiling
- **Scalability?** –> Simulate production-scale traffic
- **Availability?** –> Chaos engineering, fault injection

Trade-offs common: security vs usability, performance vs maintainability.

Context Switching Costs

Weinberg (1992): Context switching between projects is expensive

Modern reality: Agile teams, multiple simultaneous projects

Cost factors:

- Mental context rebuild: 15-30 minutes per switch
- Tool/environment switching
- Re-familiarization with codebase

Forecasting helps: Predict upcoming bug/enhancement spikes, staff proactively, minimize thrashing.

TDD in Practice: The Reality

TDD= test driven development

- write (a few) tests before (a little) coding
- intially tests fail (red)
- repeat: fix tests till all green
  - then write some more tests
- sometimes, pause and reogranize
- mantra: red, green, refactor

Karac & Turhan (2018): "What Do We Really Know about TDD?"

Findings:

- Only 12% of projects claiming TDD actually write tests first
- GitHub study: 0.8% truly TDD
- No clear evidence for higher velocity or quality
- TDD hard to define rigorously
- Success confounded with better tools (IDEs, languages)

Discussion: Is TDD itself the benefit, or is it proxy for other good practices (small functions, clear interfaces, refactoring discipline)?

Take-Home Messages

- Coverage necessary but insufficient - use mutation testing
- Test prioritization strategies are context-dependent
- Active learning effective for large black-box test suites
- Delta debugging automates input minimization
- Formal verification increasingly practical at scale (but needs testing too)
- Issue trends forecast workload without expensive labeling
- Symbolic execution enables white-box testing of data-intensive systems
- TDD effectiveness debatable; good testing habits matter more

Question 1: Fault vs Failure

(a) [1 mark] Define "fault" and "failure" and explain the key difference between them.

(b) [2 marks] Using the Fenton & Neil causal model, explain how a system with extensive pre-release testing could show many pre-release faults but few post-release failures. Sketch the causal factors involved.

(c) [3 marks] A startup releases software with minimal testing, observes few reported bugs in the first month, and concludes their code quality is excellent. Critique this reasoning using concepts from reliability engineering. What alternative explanations might account for the low bug reports?

Question 2: Coverage Metrics

(a) [1 mark] For the following code, explain why one test achieving 100% line coverage might still have 50% branch coverage:

```
def func(x):
    if x > 0:
        return x * 2
    return x
```

(b) [2 marks] Write a function with 2 if-statements where achieving 100% branch coverage requires 4 tests, but 100% line coverage requires only 2 tests. Show your test cases.

(c) [3 marks] Research shows statement coverage best predicts mutation kills (Gopinath 2014), yet coverage alone poorly correlates with test effectiveness (Inozemtseva & Holmes 2014). Reconcile these findings. When is coverage useful and when is it misleading?

Question 3: Mutation Testing

(a) [1 mark] What is a "mutation operator" and list three types (AOR, ROR, CR, etc.) with examples.

(b) [2 marks] Given this code:

```
def discount(price, rate):
    if price > 100:
        return price - (price * rate)
    return price
```

Generate 3 mutants using different operators and explain whether your test suite `[discount(50, 0.1), discount(150, 0.2)]` kills each mutant.

(c) [3 marks] A test suite has 90% line coverage but only 60% mutation score. Another has 70% coverage but 85% mutation score. Which indicates higher quality testing? Justify your reasoning considering what each metric measures and their practical implications for fault detection.

(a) [1 mark] Define APFD (Average Percentage of Faults Detected) and explain why it's more informative than simply measuring "time to find first fault."

(b) [2 marks] You have 4 tests: A (passed 10 runs ago, execution time 5s), B (failed yesterday, 10s), C (new test, 2s), D (passed yesterday, 15s). Apply the Elbaum heuristic to prioritize these tests. Show your reasoning.

(c) [3 marks] Ling et al. (2022) found optimal TCP strategies differ for open-source vs closed-source projects (diversity-based optimal for closed-source, history-based for open-source). Propose three hypotheses explaining why this difference exists. Design an experiment to test one hypothesis.

Question 5: Delta Debugging

(a) [1 mark] Explain what "1-minimal" means in the context of delta debugging's ddmin algorithm.

(b) [2 marks] You have a 16-character input "ABCDEFGHIJKLMNOP" that crashes a program. Walk through the first 3 steps of ddmin assuming: removing "ABCDEFGH" still crashes, but removing "IJKLMNOP" passes. What does ddmin try next?

(c) [3 marks] The "oracle problem" requires distinguishing the target failure from other failures. Design a test oracle for a compiler that should detect "segmentation fault" as the target failure while ignoring "syntax error" and "type error". Show code and explain how it handles ambiguous cases.

Question 6: Black-Box Testing

(a) [1 mark] Write a simple grammar (in any notation) that generates arithmetic expressions with numbers 1-9 and operators +, -, *.

(b) [2 marks] Extend your grammar with probability weights such that + is chosen 50% of the time, while - and * are each chosen 25%. Show Python code implementing weighted random selection.

(c) [3 marks] Coverage-guided fuzzing automatically reweights grammar rules to explore uncovered code paths. Compare this to manual weighting (where domain experts specify weights). Under what circumstances would each approach be more effective? Consider factors like: domain knowledge availability, code complexity, and testing budget.

(a) [1 mark] What is the difference between formal verification and traditional testing, and why might formal verification provide stronger correctness guarantees?

(b) [2 marks] AWS rewrote their authorization engine in Dafny rather than verifying the existing Java code. Explain two technical reasons why rewriting might be more practical. Consider proof brittleness, legacy code complexity, and verification tool limitations.

(c) [3 marks] The AuthV2 project performed 10ˆ15 shadow tests despite having formally verified code. Explain the relationship between proof and testing: what errors does each approach catch, why are both necessary, and what does this tell us about the role of specifications in formal methods?

Question 8: ARIMA Forecasting

(a) [1 mark] Describe the three components of ARIMA (AutoRegressive, Integrated, Moving Average) in the context of time series forecasting.

(b) [2 marks] A project has these weekly issue counts for 4 weeks: [10, 15, 12, 18]. Using a simple moving average (MA(2)), forecast the next week. Then explain how ARIMA's "integrated" component would handle a non-stationary trend.

(c) [3 marks] Research found issue counts predict bugs as accurately as bug history itself. Evaluate the practical implications for organizations: what does this enable, what are the risks/limitations, and under what conditions might this approach fail? Consider factors like project maturity, issue quality, and labeling accuracy.

Question 9: Active Learning (TERMINATOR)

(a) [1 mark] Explain the difference between "uncertainty sampling" and "certainty sampling" in active learning.

(b) [2 marks] TERMINATOR uses uncertainty sampling early ($|LR| < 30$ failures) then switches to certainty sampling. Explain the rationale for this adaptive strategy. What would go wrong if it used only uncertainty sampling throughout?

(c) [3 marks] TERMINATOR frames test prioritization as a "Total Recall" problem. Evaluate this framing: what assumptions does it make about the testing goal, how does this differ from traditional coverage-based approaches, and when might this framing be inappropriate (i.e., when is finding *all* failures quickly not the right objective)?