CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Lecture 16: Course Summary and Review

Gregory Gay
DIT636/DAT560 - March 11, 2026

The Impending Exam

- Wednesday, March 18, 8:30 - 12:30
- Practice exam on Canvas.
  - **Longer than the real exam!**
  - Try solving first without using the sample solutions. Compare your answers.
- Ask questions about any course content!

Topics

- **Quality Attributes**
- **Scenarios**
- **Test Design**
- **Unit Testing**
- System Testing
- **Exploratory Testing**
- **Structural Testing**
  - **Control-Flow**
  - **Data-Flow**
- **Mutation Testing**
- Model-Based Testing
- **Finite State Verification**
- Automated Test Generation

Primarily on the re-exams!

Practice Exam

Warm Up

1. For the expression (a || !c) || (a && b), the test suite (a, b, c) = {(T, F, T), (F, T, T), (T, T, T), (F, F, F)} provides:
   a. Path Coverage
   b. Decision Coverage
   c. Basic Condition Coverage
   d. Compound Condition Coverage
2. During exploratory testing, the couch potato tour recommends running the system for a long period of time (e.g., overnight) to see if there are failures that emerge over time.
   a. True
   b. False

Warm Up

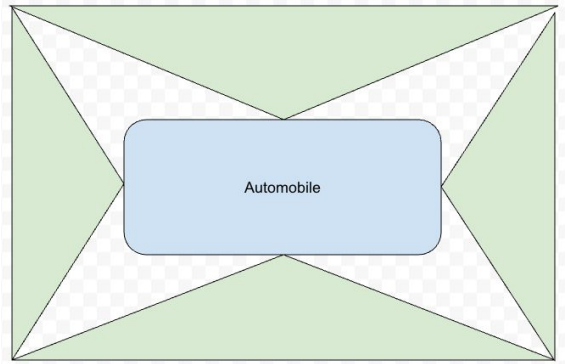
1. A liveness property is a property that should hold over a path of unknown length.
   a. **True**
   b. False
2. Mutation is considered useful because a sequence of small code changes can approximately model a larger fault in the program.
   a. **True**
   b. False
5. You have designed the software for an ATM to meet the following requirement: “If the amount of remaining money in the machine cannot be calculated, then display an error screen and prevent users from performing any additional interactions until the functionality is restored.” Which type of property is this?
   a. Correctness
   b. Reliability
   c. **Robustness**

Warm Up

5. An invalid mutant is one that is detected by many test cases.
   a. True
   b. False
6. You are designing a pedestrian detection system for a vehicle. Which one of the following quality attributes would be of most importance to you?
   a. Reliability
   b. Performance
   c. Scalability

Scenarios

Consider a camera-based object detection system in an automobile:



Create one **reliability** and one **availability** scenario.

Scenarios - Reliability

- **Description:** The system shall be able to reliably identify objects in the roadway, even when there is heavy snowfall.
- **System State:** The system is operating under normal conditions (no recent failures).
- **Environment State:** All cameras are functioning as expected, returning readings every 1ms. There is heavy snowfall (15cm accumulation in the past 12 hours, with more actively falling).
- **External Stimulus:** An animal enters the road in front of the vehicle.
- **System Response:** The object detection system detects the animal in one or more of the camera vision areas and issues a warning to the driver.
- **Response Measure:** Even under the current weather conditions, we expect fewer than 2 out of every 10000 requests to fail (POFOD).

Scenarios - Reliability

- **Description:** If the snowfall is sufficiently bad to prevent the cameras from working, the system shall warn the driver of degraded capabilities.
- **System State:** The system is operating under normal conditions (no recent failures).
- **Environment State:** All cameras are functioning as expected, returning readings every 1ms. There is extremely heavy snowfall (25cm accumulation in the past 12 hours, with the quantity falling steadily increasing).
- **External Stimulus:** The camera on the right-hand side of the vehicle has been covered in snow.
- **System Response:** The system shall detect that the sensor is failing to send valid data (i.e., no image or an image with no usable data). After detecting that the sensor is no longer working as expected, it shall issue a warning on the driver's heads-up display that the system cannot reliably detect obstacles under the current conditions. The system shall continue to monitor the camera data. Once the camera resumes sending usable image data, then the warning shall be disabled.
- **Response Measure:** The system shall detect the failure within 2ms in 97% of cases, and within 4ms in 99.99% of cases. The system shall then issue the warning within 1 additional ms in 95% of cases and 2ms in 99% of cases. Once the camera is working again, the system shall remove the warning within 1ms in 95% of cases and 2ms in 99% of cases.

Testing Concepts

- Define **integration** testing and **exploratory** testing.
- Explain **how systems are tested** at each stage.
- Explain how the two **stages differ**.
- Explain **types of faults** that may be more likely to be exposed by each of the two stages.

Testing Concepts

Integration

- Through an interface (API, CLI, GUI)
- Code-based
- Pre-planned
- Exposes interaction faults, code mistakes.

Exploratory

- Through an interface (generally GUI)
- Manual
- Ad-hoc
- Can expose interaction faults, code mistakes, but also usability, graphical issues, user perception

Test Design

```
public boolean applyForVacation (String userID,
String startingDate, String endingDate)
```

- A user ID is a string in the format “firstname.lastname”, e.g., “gregory.gay”.
- The two dates are strings in the format “YYYY-DD-MM”.
- The function returns TRUE if the user was able to successfully apply for the vacation time. It returns FALSE if not. An exception can also be thrown if there is an error.

Test Design

User database with following items for each user:

- User ID
- Quantity of remaining vacation days for the user
- An array containing already-scheduled vacation dates (as starting and ending date pairs)
- An array containing dates where vacation cannot be applied for (e.g., important meetings).

Test Design

Perform functional test design for this function.

1. Identify choices (controllable aspects that can be varied when testing)
2. For each choice, identify representative values.
3. For each value, apply constraints (IF, ERROR, SINGLE) if they make sense.

- **Choice: Value of userID**
  - Existing user
  - Non-existing user **[error]**
  - Null **[error]**
  - Malformed user ID (not in format “firstname.lastname”) **[error]**
- **Choice: Value of starting date**
  - Valid date
  - Date before the current date **[error]**
  - Current date **[single]**
  - Null **[error]**
  - Malformed date (not in format “YYYY-MM-DD”) **[error]**
- **Choice: Value of ending date**
  - Valid date
  - Date before the current date **[error]**
  - Current date **[single]**
  - Date before the starting date **[error]**
  - Date same as the starting date **[single]**
  - Null **[error]**
  - Malformed date (not in format “YYYY-MM-DD”) **[error]**

- **Choice: Remaining vacation time for the userID**
  (Note: We are assuming the database schema prevents storing malformed/invalid values)
  - 0 days remaining
  - 1 day remaining, 1 day applied for **[single]**
  - Number of days remaining < number applied for
  - Number of days remaining = number applied for **[single]**
  - Number of days remaining > number applied for
  - User does not exist **[if user ID does not exist]**
- **Choice: Conflicts with vacation time**
  (Note: We are assuming the database schema prevents storing malformed/invalid date ranges)
  - No conflicts with scheduled vacation or banned dates
  - Banned date(s) fall within the starting and ending dates
  - Starting date falls within already-scheduled vacation time
  - Ending date falls within already-scheduled vacation time
  - Already-scheduled vacation time falls within starting and ending dates applied for
  - The starting and ending dates fall within already-scheduled vacation time
  - User does not exist **[if user ID does not exist]**

Test Design

```
public String[] analyzeObstacles (List<Float> cameraData)
```

`cameraData` - list of distances of 0+ potential obstacles.

Returns a String array, where each item is a warning:

- Any detected obstacle that is within 2 meters of the camera.
- Obstruction may be blocking the camera (a detected object 0.00 meters from the camera).

Test Design

Choice: Number of items in cameraData

- 0
- 1
- 10 (i.e., “typical”)
- 10000 (i.e., “large set to test performance”) **[single]**

Choice: Number of obstacles that could cause a collision (< 2 meters)

- 0
- 1 **[if number of items > 0]**
- 2+ **[if number of items > 1]**

Choice: Potential obstruction? (0.00 meters)

- Yes **[if number of items > 0]**
- No

Choice: Are there items in cameraData that could cause an error?

- No
- At least one null value **[error] [if number of items > 0]**
- At least one negative value **[error] [if number of items > 0]**

Exploratory Testing

1) Describe one of the tours that we discussed in class.
2) Consider a web-based discussion forum. This forum offers the following functionality:
   - Users can register for an account.
   - Administrators can create, edit the name and description of, and delete “boards”.
   - Users can also post, reply, edit posts in topics.
   - Administrators can also edit or delete posts in topics.

Describe three interaction sequences.

Exploratory Testing

- **FedEx Tour**
  - Models data creation and manipulation across features.
  - Identify data that is stored, and “track” it as it gets changed or used.
  - Particular focus on interactions with same data across multiple “functions”.

Exploratory Testing

Describe three interaction sequences

1. Register, log in, log out, try to register with same username or email, verify rejection, log in again.
2. Log in as admin, post topic, edit post, reply, delete reply.
3. Log in as admin, create board, edit name/description, delete board.

Unit Testing

```
public String[] analyzeObstacles (List<Float> cameraData)
```

Write three JUnit-format test cases:

1. Create one test case that checks a normal usage of the method, with at least one detected obstacle.
2. Create one test case that checks a normal usage of the method, with a potential obstruction.
3. Create one test case reflecting an error-handling scenario (an exception is thrown).

Unit Testing

```
@Test
public void testDetectedObstacle() {
    List<Float> input = new ArraytList<Float>();
    input.add(0.50);
    String[] expected = new String[1];
    expected[0] = “Obstacle detected at 0.50 meters”;
    String[] output = analyzeObjects(input);
    assertArrayEquals(output, expected);
    assertTrue(output.length == 1);
}

@Test
public void testDetectedObstruction() {
    List<Float> input = new ArraytList<Float>();
    input.add(0.00);
    String[] expected = new String[1];
    expected[0] = “There is a potential obstruction blocking the
camera”;
    String[] output = analyzeObjects(input);
    assertArrayEquals(output, expected);
    assertTrue(output.length == 1);
}
```

```
@Test
public void testNullContents() {
    List<Float> input = new ArraytList<Float>();
    input.add(null);

    assertThrows(NullPointerException.class, () -> {
        analyzeObjects(input);
    });
}
```

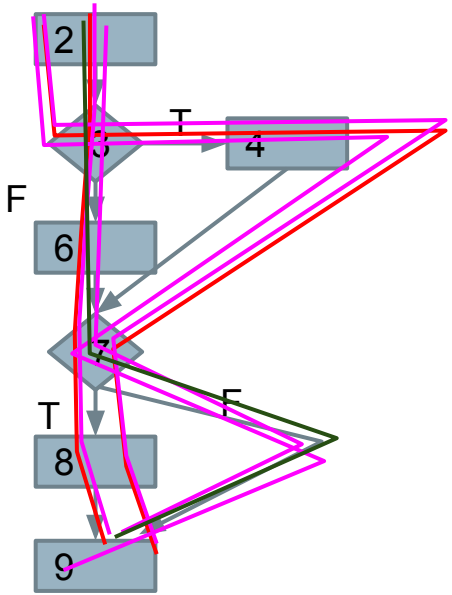
Let’s Take a Break

Structural Testing

- Draw the control-flow graph for this method.
- Develop test input that will provide statement coverage.
- Develop test input that will provide branch coverage.
- Develop test input that will provide path coverage.

```
int findMax(int a, int b, int c) {
     int temp;
     if (a > b)
          temp=a;
     else
          temp=b;
     if (c > temp)
          temp = c;
     return temp;
}
```

Structural Testing




```
1. int findMax(int a, int b, int c) {
2.   int temp;
3.   if (a>b)
4.        temp=a;
5.   else
6.        temp=b;
7.   if (c>temp)
8.        temp = c;
9.   return temp;
10. }
```

Statement:
(3,2,4), (2,3,4)
Branch:
(3,2,4), (3,4,1)

Path:
(4,2,5), (4,2,1), (2,3,4), (2,3,1)

Structural Testing

- Modify the program to introduce a fault such that even path coverage *could* miss the fault.

Use (a > b+1) instead of (a>b) and the test input from the last slide:
(4,2,5), (4,2,1), (2,3,4), (2,3,1)
will not reveal the fault.

```
int findMax(int a, int b, int c)
{
    int temp;
    if (a>b)
        temp=a;
    else
        temp=b;
    if (c>temp)
        temp = c;
    return temp;
}
```

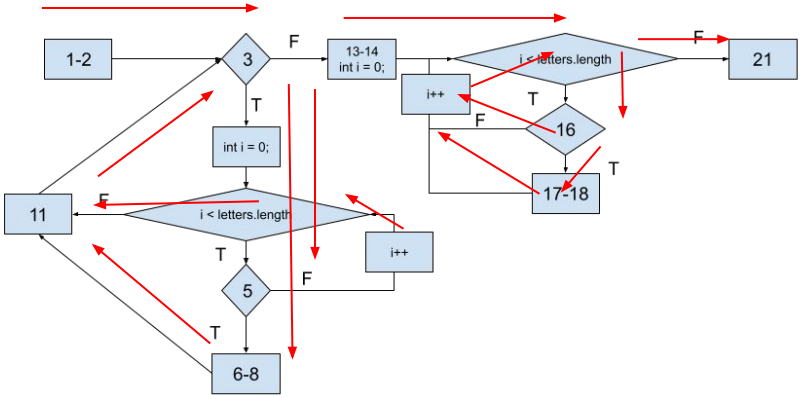
Structural Testing

- removeLetters(["s", "t", "r", "i", "n", "g", "w"], "string") → ["w"]
- removeLetters(["b", "b", "l", "l", "g", "n", "o", "a", "w"], "balloon") → ["b", "g", "w"]
- removeLetters(["d", "b", "t", "e", "a", "i"], "edabit") → [ ]

1. Draw the control-flow graph for this function.
2. Identify test input that will provide statement and branch coverage.

```
1.  public String[] removeLetters(String[] letters, String word) {
2.      int lettersRemoved = 0;
3.      while(word.length() != 0){
4.          for(int i = 0; i < letters.length; i++){
5.              if(letters[i].equals(word.substring(0, 1))){
6.                  letters[i] = "";
7.                  lettersRemoved++;
8.                  break;
9.              }
10.         }
11.         word = word.substring(1);
12.     }
13.     int idx = 0;
14.     String[] output = new String[letters.length - lettersRemoved];
15.     for(int i = 0; i < letters.length; i++){
16.         if(!letters[i].equals("")){
17.             output[idx] = letters[i];
18.             idx++;
19.         }
20.     }
21.     return output;
22. }
```

Structural Testing



removeLetters(["d", "i", "t"], "dir") → ["t"]

Data Flow Testing

- Identify all DU pairs and write test cases to achieve All DU Pair Coverage.

```
1.    public String[] removeLetters(String[] letters, String word) {
2.        int lettersRemoved = 0;
3.        while(word.length() != 0){
4.            for(int i = 0; i < letters.length; i++){
5.                if(letters[i].equals(word.substring(0, 1))){
6.                    letters[i] = "";
7.                    lettersRemoved++;
8.                    break;
9.                }
10.           }
11.           word = word.substring(1);
12.       }
13.       int idx = 0;
14.       String[] output = new String[letters.length - lettersRemoved];
15.       for(int i = 0; i < letters.length; i++){
16.            if(!letters[i].equals("")){
17.                output[idx] = letters[i];
18.                idx++;
19.            }
20.       }
21.       return output;
22.   }
```

Data Flow Testing

Variable	D-U Pairs
letters	(1, 4), (1, 5), (6, 4), (6, 5), (6, 6), (1, 14), (6, 14), (1, 15), (6, 15), (1, 16), (6, 16), (1, 17), (6, 17)
word	(1, 3), (1, 5), (1, 11), (11, 5), (11, 11), (11,3)
lettersRemoved	(2, 7), (7, 7), (2, 14), (7, 14)
i	(4, 4), (4, 5), (4, 6), (15, 15), (15, 16), (15, 17)
idx	(13, 17), (13, 18), (18, 18), (18, 17)
output	(14, 21), (17,21)

```
1.   public String[] removeLetters(String[] letters, String word) {
2.       int lettersRemoved = 0;
3.       while(word.length() != 0){
4.           for(int i = 0; i < letters.length; i++){
5.               if(letters[i].equals(word.substring(0, 1))){
6.                   letters[i] = "";
7.                   lettersRemoved++;
8.                   break;
9.               }
10.          }
11.          word = word.substring(1);
12.      }
13.      int idx = 0;
14.      String[] output = new String[letters.length - lettersRemoved];
15.      for(int i = 0; i < letters.length; i++){
16.          if(!letters[i].equals("")){
17.              output[idx] = letters[i];
18.              idx++;
19.          }
20.      }
21.      return output;
22.  }
```

Data Flow Testing

removeLetters(["d", "i", "t", “y”], "dir") → ["t", “y”]

Variable	New D-U Pairs Covered
letters	(1, 4), (1, 5), (6, 4), (6, 5), (6, 6), (6, 14), (6, 15), (6, 16), (6, 17)
word	(1, 3), (1, 5), (1, 11), (11, 3), (11, 5), (11, 11)
lettersRemoved	(2, 7), (7, 7), (7, 14)
i	(4, 5), (4, 6), (4, 4), (15, 16), (15, 15), (15, 17)
idx	(13, 17), (13, 18), (18, 18), (18, 17)
output	(17, 21)

```
1.  public String[] removeLetters(String[] letters, String word) {
2.      int lettersRemoved = 0;
3.      while(word.length() != 0){
4.          for(int i = 0; i < letters.length; i++){
5.              if(letters[i].equals(word.substring(0, 1))){
6.                  letters[i] = "";
7.                  lettersRemoved++;
8.                  break;
9.              }
10.         }
11.         word = word.substring(1);
12.     }
13.     int idx = 0;
14.     String[] output = new String[letters.length - lettersRemoved];
15.     for(int i = 0; i < letters.length; i++){
16.         if(!letters[i].equals("")){
17.             output[idx] = letters[i];
18.             idx++;
19.         }
20.     }
21.     return output;
22. }
```

Data Flow Testing

removeLetters(["d"], "x") → ["d"]

Variable	New D-U Pairs Covered
letters	(1, 14), (1, 15), (1, 16), (1, 17)
lettersRemoved	(2, 14)

```
1.    public String[] removeLetters(String[] letters, String word) {
2.        int lettersRemoved = 0;
3.        while(word.length() != 0){
4.            for(int i = 0; i < letters.length; i++){
5.                if(letters[i].equals(word.substring(0, 1))){
6.                    letters[i] = "";
7.                    lettersRemoved++;
8.                    break;
9.                }
10.           }
11.           word = word.substring(1);
12.       }
13.       int idx = 0;
14.       String[] output = new String[letters.length - lettersRemoved];
15.       for(int i = 0; i < letters.length; i++){
16.            if(!letters[i].equals("")){
17.                output[idx] = letters[i];
18.                idx++;
19.            }
20.       }
21.       return output;
22.   }
```

Data Flow Testing

removeLetters(["d"], "d") → [ ]

Variable	New D-U Pairs Covered
output	(14, 21)

```
1.     public String[] removeLetters(String[] letters, String word) {
2.         int lettersRemoved = 0;
3.         while(word.length() != 0){
4.             for(int i = 0; i < letters.length; i++){
5.                 if(letters[i].equals(word.substring(0, 1))){
6.                     letters[i] = "";
7.                     lettersRemoved++;
8.                     break;
9.                 }
10.            }
11.            word = word.substring(1);
12.        }
13.        int idx = 0;
14.        String[] output = new String[letters.length - lettersRemoved];
15.        for(int i = 0; i < letters.length; i++){
16.             if(!letters[i].equals("")){
17.                 output[idx] = letters[i];
18.                 idx++;
19.             }
20.        }
21.        return output;
22.    }
```

Data Flow Testing

- Identify all DU pairs and write test cases to achieve All DU Pair Coverage.

```
1.  public static boolean canPartition(int[] arr) {
2.      Arrays.sort(arr);
3.      int product = 1;
4.      if ((Math.abs(arr[0]) >= arr[arr.length-1])
    || arr[0] == 0) {
5.          for (int i = 1; i < arr.length; i++){
6.              product *= arr[i];
7.          }
8.          return arr[0] == product;
9.      } else{
10.         for (int i = 0; i < arr.length-1; i++){
11.             product *= arr[i];
12.         }
13.         return arr[arr.length-1] == product;
14.     }
15. }
```

Data Flow Testing

```
1.   public static boolean canPartition(int[] arr) {
2.       Arrays.sort(arr);
3.       int product = 1;
4.       if ((Math.abs(arr[0]) >= arr[arr.length-1])
     || arr[0] == 0) {
5.           for (int i = 1; i < arr.length; i++){
6.               product *= arr[i];
7.           }
8.           return arr[0] == product;
9.       } else{
10.          for (int i = 0; i < arr.length-1; i++){
11.              product *= arr[i];
12.          }
13.          return arr[arr.length-1] == product;
14.      }
15.  }
```

arr	(1, 2), (2, 4), (2, 5), (2, 6), (2, 8), (2, 10), (2, 11), (2, 13)
product	(3, 6), (6, 6), (3, 8), (6, 8), (3, 11), (11, 11), (11, 13)
i	(5, 5), (5, 6), (10, 10), (10, 11)

Data Flow Testing

```
1.  public static boolean canPartition(int[] arr) {
2.      Arrays.sort(arr);
3.      int product = 1;
4.      if ((Math.abs(arr[0]) >= arr[arr.length-1])
    || arr[0] == 0) {
5.          for (int i = 1; i < arr.length; i++){
6.              product *= arr[i];
7.          }
8.          return arr[0] == product;
9.      } else{
10.         for (int i = 0; i < arr.length-1; i++){
11.             product *= arr[i];
12.         }
13.         return arr[arr.length-1] == product;
14.     }
15. }
```

arr	(1, 2), (2, 4), (2, 5), (2, 6), (2, 8), (2, 10), (2, 11), (2, 13)
product	(3, 6), (6, 6), (3, 8), (6, 8), (3, 11), (11, 11), (11, 13)
i	(5, 5), (5, 6), (10, 10), (10, 11)

Input	**Additional DU Pairs Covered**
`[2, 8, 4, 1]`	arr: (1, 2), (2, 4), (2, 10), (2, 11), (2, 13) product: (3, 11), (11, 11), (11, 13) i: (10, 10), (10, 11)
`[-1, -10, 0, 10]`	arr: (2, 5), (2, 6), (2, 8) product: (3, 6), (6, 6), (6, 8) i: (5, 5), (5, 6)
`[0]`	product: (3, 8)

Mutation Testing

Consider the following function:

```
 1.   public static long lonelyNumbers(int n) {
 2.       String s = " " + n + " ";
 3.       long a = 0;
 4.       StringBuilder sb = new StringBuilder();
 5.       for(int i = 1; i < s.length() - 1; i++){
 6.           if(s.charAt(i) == s.charAt(i - 1) ||
      s.charAt(i) ==
                s.charAt(i + 1)){
 7.               sb.append(s.charAt(i));
 8.           } else
 9.               sb.append("" + s.charAt(i) +
      s.charAt(i) + s.charAt(i));
10.     }
11.     a = Long.parseLong(sb.toString().trim());
12.     return a;
13.   }
```

- `lonelyNumbers(4666)` → `444666`
- `lonelyNumbers(544)` → `55544`
- `lonelyNumbers(123)` → `111222333`
- `lonelyNumbers(33)` → `33`

Mutation Testing

Consider the following function:

```
1.  public static long lonelyNumbers(int n) {
2.      String s = " " + n + " ";
3.      long a = 0;
4.      StringBuilder sb = new StringBuilder();
5.      for(int i = 1; i < s.length() - 1; i++){
6.          if(s.charAt(i) == s.charAt(i - 1) ||
    s.charAt(i) ==
                s.charAt(i + 1)){
7.              sb.append(s.charAt(i));
8.          } else
9.              sb.append("" + s.charAt(i) +
    s.charAt(i) + s.charAt(i));
10.     }
11.     a = Long.parseLong(sb.toString().trim());
12.     return a;
13.  }
```

Relational Operator Replacement (ROR)

Can mutate lines 5, 6

Line 6:

```
if(s.charAt(i) != s.charAt(i - 1) ...
```

```
lonelyNumbers(4666)
```

Original: returns 444666

Mutant: returns 466666

Mutation does not add duplicate “4”s, adds additional “6”s at the end.

Mutation Testing

Consider the following function:

```
1.  public static long lonelyNumbers(int n) {
2.      String s = " " + n + " ";
3.      long a = 0;
4.      StringBuilder sb = new StringBuilder();
5.      for(int i = 1; i < s.length() - 1; i++){
6.          if(s.charAt(i) == s.charAt(i - 1) ||
    s.charAt(i) ==
                s.charAt(i + 1)){
7.              sb.append(s.charAt(i));
8.          } else
9.              sb.append("" + s.charAt(i) +
    s.charAt(i) + s.charAt(i));
10.     }
11.     a = Long.parseLong(sb.toString().trim());
12.     return a;
13. }
```

Arithmetic Operator Replacement (AOR)

Can mutate lines 2, 5, 6, 9

Line 6:

```
if(s.charAt(i) == s.charAt(i + 1) ...
```

`lonelyNumbers(4666)`

Original: returns 444666

Mutant: returns 44466666

Adds additional “6”s at the end.

Mutation Testing

Consider the following function:

```
1.  public static long lonelyNumbers(int n) {
2.      String s = " " + n + " ";
3.      long a = 0;
4.      StringBuilder sb = new StringBuilder();
5.      for(int i = 1; i < s.length() - 1; i++){
6.          if(s.charAt(i) == s.charAt(i - 1) ||
    s.charAt(i) ==
                s.charAt(i + 1)){
7.              sb.append(s.charAt(i));
8.          } else
9.              sb.append("" + s.charAt(i) +
    s.charAt(i) + s.charAt(i));
10.     }
11.     a = Long.parseLong(sb.toString().trim());
12.     return a;
13. }
```

Constant-for-Constant Replacement (CRP)

Can mutate lines 2, 3, 5, 6, 9

Line 6:

```
if(s.charAt(i) == s.charAt(i - 0) ...
```

```
lonelyNumbers(4666)
```

Original: returns 444666

Mutant: returns 4666

Does not add the duplicate “4”s that should be present.

Finite State Verification

Temporal Operators:

- **G p:** p holds globally at every state on the path from now until the end
- **F p:** p holds at some future state on the path (but not all future states)
- **X p:** p holds at the next state on the path
- **p U q:** q holds at some state on the path and p holds at every state before the first state at which q holds.
- **A:** for all paths reaching out from a state, used in CTL as a modifier for the above properties (i.e., **AG p**)
- **E:** for one or more paths reaching out from a state (but not all), used in CTL as a modifier for the above properties (i.e., **EG p**)

AG (pedestrian_light = walk -> traffic_light != green)

State variables:

- **traffic_light: {RED, YELLOW, GREEN}**
- **pedestrian_light: {WAIT, WALK, FLASH}**
- **button: {RESET, SET}**

Initially: **traffic_light = RED, pedestrian_light = WAIT, button = RESET**

Transitions:

pedestrian_light:

- **WAIT → WALK if traffic_light = RED**
- **WAIT → WAIT otherwise**
- **WALK → {WALK, FLASH}**
- **FLASH → {FLASH, WAIT}**

traffic_light:

- **RED → GREEN if button = RESET**
- **RED → RED otherwise**
- **GREEN → {GREEN, YELLOW} if button = SET**
- **GREEN → GREEN otherwise**
- **YELLOW→ {YELLOW, RED}**

button:

- **SET → RESET if pedestrian_light = WALK**
- **SET → SET otherwise**
- **RESET → {RESET, SET} if traffic_light = GREEN**
- **RESET → RESET otherwise**

G (traffic_light = RED & button = RESET -> F (traffic_light = green))

State variables:

- **traffic_light: {RED, YELLOW, GREEN}**
- **pedestrian_light: {WAIT, WALK, FLASH}**
- **button: {RESET, SET}**

Initially: **traffic_light = RED, pedestrian_light = WAIT, button = RESET**

Transitions:

pedestrian_light:

- **WAIT → WALK if traffic_light = RED**
- **WAIT → WAIT otherwise**
- **WALK → {WALK, FLASH}**
- **FLASH → {FLASH, WAIT}**

traffic_light:

- **RED → GREEN if button = RESET**
- **RED → RED otherwise**
- **GREEN → {GREEN, YELLOW} if button = SET**
- **GREEN → GREEN otherwise**
- **YELLOW→ {YELLOW, RED}**

button:

- **SET → RESET if pedestrian_light = WALK**
- **SET → SET otherwise**
- **RESET → {RESET, SET} if traffic_light = GREEN**
- **RESET → RESET otherwise**

Negate to get trap property:
G !(button = SET -> F (pedestrian_light = WALK))

State variables:

- **traffic_light: {RED, YELLOW, GREEN}**
- **pedestrian_light: {WAIT, WALK, FLASH}**
- **button: {RESET, SET}**

Initially: **traffic_light = RED, pedestrian_light = WAIT, button = RESET**

Transitions:

pedestrian_light:

- **WAIT → WALK if traffic_light = RED**
- **WAIT → WAIT otherwise**
- **WALK → {WALK, FLASH}**
- **FLASH → {FLASH, WAIT}**

traffic_light:

- **RED → GREEN if button = RESET**
- **RED → RED otherwise**
- **GREEN → {GREEN, YELLOW} if button = SET**
- **GREEN → GREEN otherwise**
- **YELLOW→ {YELLOW, RED}**

button:

- **SET → RESET if pedestrian_light = WALK**
- **SET → SET otherwise**
- **RESET → {RESET, SET} if traffic_light = GREEN**
- **RESET → RESET otherwise**

Finite State Verification

Microwave controller

- Door: {Open, Closed} -- sensor input indicating state of the door
- Button: {None, Start, Stop} -- button press
- Timer: 0...999 -- (remaining) seconds to cook
- Cooking: Boolean -- state of the heating element

In CTL:

- The microwave shall never cook when the door is open.
- **AG (Door = Open -> !Cooking)**

Finite State Verification

Microwave controller

- Door: {Open, Closed} -- sensor input indicating state of the door
- Button: {None, Start, Stop} -- button press
- Timer: 0...999 -- (remaining) seconds to cook
- Cooking: Boolean -- state of the heating element

In CTL:

- The microwave shall cook only as long as there is remaining cook time.
- **AG (Cooking -> Timer > 0)**

Verification

Microwave controller

- Door: {Open, Closed} -- sensor input indicating state of the door
- Button: {None, Start, Stop} -- button press
- Timer: 0...999 -- (remaining) seconds to cook
- Cooking: Boolean -- state of the heating element

In CTL:

- If the stop button is pressed when the microwave is not cooking, the remaining cook time shall be cleared.
- **AG (Button = Stop & !Cooking -> AX (Timer = 0))**

Finite State Verification

Microwave controller

- Door: {Open, Closed} -- sensor input indicating state of the door
- Button: {None, Start, Stop} -- button press
- Timer: 0...999 -- (remaining) seconds to cook
- Cooking: Boolean -- state of the heating element

In LTL:

- It shall never be the case that the microwave can continue cooking indefinitely.
- **G (Cooking -> F (!Cooking))**

Finite State Veri.

Microwave controller

- Door: {Open, Closed} -- sensor input indicating state of the door
- Button: {None, Start, Stop} -- button press
- Timer: 0...999 -- (remaining) seconds to cook
- Cooking: Boolean -- state of the heating element

In LTL:

- The only way to initiate cooking shall be pressing the start button when the door is closed and the remaining cook time is not zero.
- **G (!Cooking U ((Button = Start & Door = Closed) & (Timer > 0)))**

Finite State Verification

Microwave controller

- Door: {Open, Closed} -- sensor input indicating state of the door
- Button: {None, Start, Stop} -- button press
- Timer: 0...999 -- (remaining) seconds to cook
- Cooking: Boolean -- state of the heating element

In LTL:

- The microwave shall continue cooking when there is remaining cook time unless the stop button is pressed or the door is opened.
- **G ((Cooking & Timer > 0) -> X (((Cooking | (!Cooking & Button = Stop)) | (!Cooking & Door = Open)))**

Any other questions?

Thank you for being a great class!

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY