LTAT.05.006: Software Testing

Lecture 05:
State-Transition, Random, and Metamorphic Testing

Dietmar Pfahl
email: dietmar.pfahl@ut.ee

Spring 2021
Lectures

• Lecture 1 (11.02) – Introduction to Software Testing
• Lecture 2 (18.02) – Basic Black-Box Testing Techniques: Boundary Value Analysis & Equivalence Class Partitioning
• Lecture 3 (25.02) – BBT advanced: Combinatorial Testing
• Lecture 4 (04.03) – Basic White-Box Testing Techniques: Control-Flow Coverage
• Lecture 5 (11.03) – BBT adv.: State-Transition, Metamorphic, Random Testing
• Lecture 6 (18.03) – Test Levels, Test Tools, Test Automation
• Lecture 7 (25.03) – BBT adv.: Exploratory Testing, Behaviour Testing
• Lecture 8 (01.04) – BBT adv.: GUI / Visual Testing, Usability Testing, A/B Testing
• Lecture 9 (08.04) – Data-Flow Testing / Test-Suite Effectiveness: Mutation Testing
• Lecture 10 (15.04) – WBT adv.: Symbolic Execution, Static Code Analysis, Review
• Lecture 11 (22.04) – Defect Estimation / Test Documentation, Organisation and Process Improvement (Test Maturity Model)
• Lectures 12+13 (29.04 + 06.05) – Industry Guest Lectures + Advanced Topics
• Lecture 14 (13.05) – Exam Preparation
Structure of Lecture 5

• State-Transition Testing
• Random Testing
• Certification Testing
• Metamorphic Testing
• Lab 5
State-Transition Testing - Example

Use Case Diagram

Check Account
Withdraw Money
...

ATM
State-Transition Testing - Example

Use Case Description: Check Account

Role: Customer
Goal: Customer wants to check the amount of money in his/her accounts

Scenario (actions):
1. ATM asks for customer card
2. Customer enters card
3. ATM asks for PIN code
4. Customer enters PIN code
5. …
State-Transition Diagram

Scenario (actions):
1. ATM asks for customer card -> **State: Wait for card**
2. Customer enters card
3. ATM asks for PIN code -> **Wait for PIN**
4. Customer enters PIN code
5. …
Now create a set of test cases that trigger each state-transition at least once.
### State Table

<table>
<thead>
<tr>
<th>Input (Event)</th>
<th>State</th>
<th>Wait for Card (S1)</th>
<th>Wait for PIN (S2)</th>
<th>Next (S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Card inserted</td>
<td>Ask for PIN -&gt; S2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Invalid PIN</td>
<td>-</td>
<td>Beep -&gt; S2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Valid PIN</td>
<td>-</td>
<td>Ask amount -&gt; S3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cancel</td>
<td>-</td>
<td>Return card -&gt; S1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
State Table

4 Test Cases:
S1 -> 'Card inserted' / 'Ask for PIN' -> S2
S2 -> 'Invalid PIN' / 'Beep' -> S2
S2 -> 'Valid PIN' / 'Ask amount' -> S3
S2 -> 'Cancel' / 'Return card' -> S1

<table>
<thead>
<tr>
<th>Input (Event)</th>
<th>State</th>
<th>Wait for Card (S1)</th>
<th>Wait for PIN (S2)</th>
<th>Next (S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Card inserted</td>
<td>Ask for PIN -&gt; S2</td>
<td>-</td>
<td>Beep -&gt; S2</td>
<td>-</td>
</tr>
<tr>
<td>Invalid PIN</td>
<td>-</td>
<td>Beep -&gt; S2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Valid PIN</td>
<td>-</td>
<td>Ask amount -&gt; S3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cancel</td>
<td>-</td>
<td>Return card -&gt; S1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Test scenario (=sequence of test cases):
S1 -> 'Card inserted' / 'Ask for PIN' -> S2 -> 'Cancel' / 'Return card' -> S1
State-Transition Testing: Example 2

Extract of a Specification Doc:

Parameters

- PORT_A: calling phone
- PORT_B: called phone

PORT_A identifies the connection from which a call is to be set up. The actual state of the call setup is globally available. Depending on this a new state arises after the evaluation of the transferred event. The delivered state is DISCONNECTED, if the call setup was terminated, it is DIALING, if the call setup is in progress but not completed yet. It is CONNECTED, if the call setup was successfully completed. In this case PORT_B delivers the connection of the selected subscriber, otherwise the data content of PORT_B is undefined. A call setup requires the sequence UNHOOK (DIGIT_N)* and the digit sequence must represent a valid number. HANG UP always leads to the complete termination of the call. If TIMEOUT occurs, HANG UP brings the software back into the initial state (DISCONNECTED).
State-Transition Testing: Example 2

State Chart
State-Transition Testing: Example 2

The minimal test strategy is to cover each state at least once.

A better strategy is to cover each transition at least once, which leads, e.g., to the following test scenarios …
State-Transition Testing: Example 2

State Chart

- DISCONNECTED, unhook -> DIALING, hang up -> DISCONNECTED
State-Transition Testing: Example 2

State Chart

- DISCONNECTED, unhook → DIALING, timeout → TIMEOUT OCCURRED, hang up → DISCONNECTED
State-Transition Testing: Example 2

State Chart

DISCONNECTED, unhook ->
DIALING, Digit 0..9 ->
DIALING, Digit 0..9 ->
DIALING, dialed number valid -> CONNECTED, hang up -> DISCONNECTED
State-Transition Testing: Example 2

State Chart

- DISCONNECTED, unhook ->
- DIALING, Digit 0..9 ->
- DIALING, Digit 0..9 ->
- DIALING, dialed number invalid -> INVALID NUMBER, timeout -> TIMEOUT OCCURRED, hang up -> DISCONNECTED
State-Transition Testing: Example 2

The minimal test strategy is to cover each state at least once.

A better strategy is to cover each transition at least once, which leads, e.g., to the following test scenarios (or, short: tests):

- DISCONNECTED, unhook -> DIALING, hang up -> DISCONNECTED

- DISCONNECTED, unhook -> DIALING, timeout -> TIMEOUT OCCURRED, hang up -> DISCONNECTED

- DISCONNECTED, unhook -> DIALING, Digit 0..9 -> DIALING, Digit 0..9 -> DIALING, dialed number valid -> CONNECTED, hang up -> DISCONNECTED

- DISCONNECTED, unhook -> DIALING, Digit 0..9 -> DIALING, Digit 0..9 -> DIALING, dialed number invalid -> INVALID NUMBER, timeout -> TIMEOUT OCCURRED, hang up -> DISCONNECTED

... and so on ...
State-Transition Testing: Example 2

The minimal test strategy is to cover each state at least once.

A better strategy is to cover each transition at least once, which leads, e.g., to the following tests:

- DISCONNECTED, unhook -> DIALING, hang up -> DISCONNECTED
- DISCONNECTED, unhook -> DIALING, timeout -> TIMEOUT OCCURRED, hang up -> DISCONNECTED
- DISCONNECTED, unhook -> DIALING, Digit 0..9 -> DIALING, Digit 0..9 -> DIALING, dialed number valid -> CONNECTED, hang up -> DISCONNECTED
- DISCONNECTED, unhook -> DIALING, Digit 0..9 -> DIALING, Digit 0..9 -> DIALING, dialed number invalid -> INVALID NUMBER, timeout -> TIMEOUT OCCURRED, hang up -> DISCONNECTED

Furthermore, it is useful to test all events, if transitions can be initiated by more than one event. The result is a strength-hierarchy of test techniques:

all states ≤ all transitions ≤ all events

Important: Do not forget to test the failure treatment!
State-Transition Testing: Example 2

<table>
<thead>
<tr>
<th>Event</th>
<th>DISCONNECTED</th>
<th>DIALING</th>
<th>CONNECTED</th>
<th>INVALID NUMBER</th>
<th>TIMED OUT OCCURRED</th>
</tr>
</thead>
<tbody>
<tr>
<td>unhook</td>
<td>DIALING</td>
<td>FAILURE</td>
<td>FAILURE</td>
<td>FAILURE</td>
<td>FAILURE</td>
</tr>
<tr>
<td>hang up</td>
<td>FAILURE</td>
<td>DISCONNECT ED</td>
<td>DISCONNECT ED</td>
<td>DISCONNECT ED</td>
<td>DISCONNECT ED</td>
</tr>
<tr>
<td>digit 0</td>
<td>DISCONNECT ED</td>
<td>DIALING</td>
<td>CONNECTED</td>
<td>INVALID NUMBER</td>
<td>TIMED OUT OCCURRED</td>
</tr>
<tr>
<td>digit 9</td>
<td>DISCONNECT ED</td>
<td>DIALING</td>
<td>CONNECTED</td>
<td>INVALID NUMBER</td>
<td>TIMED OUT OCCURRED</td>
</tr>
<tr>
<td>timeout</td>
<td>FAILURE</td>
<td>TIMED OUT OCCURRED</td>
<td>FAILURE</td>
<td>TIMED OUT OCCURRED</td>
<td>TIMED OUT OCCURRED</td>
</tr>
<tr>
<td>dialed number valid</td>
<td>FAILURE</td>
<td>CONNECTED</td>
<td>FAILURE</td>
<td>FAILURE</td>
<td>FAILURE</td>
</tr>
<tr>
<td>dialed number invalid</td>
<td>FAILURE</td>
<td>INVALID NUMBER</td>
<td>FAILURE</td>
<td>FAILURE</td>
<td>FAILURE</td>
</tr>
</tbody>
</table>
State-Transition Testing: Example 2

State Chart with FAILURE state
State-Transition Testing  
(state-transition coverage)  

vs.  

White-Box Testing  
(control-flow-coverage)
Stack Example

```java
public class Stack {
    int[] values = new int[3];
    int size = 0;

    void push(int x) {
        if (size >= values.length) {
            resize();
        }
        values[size++] = x;
    }

    int pop() {
        if (size > 0) {
            return values[size--];
        }
        throw new EmptyStackException();
    }

    private void resize() {
        int[] tmp = new int[values.length * 2];
        for (int i = 0; i < values.length; i++) {
            tmp[i] = values[i];
        }
        values = tmp;
    }
}
```

diagram:
- Push
- Pop

push(elem1)
pop() -> elem1
pop() -> exception

push(elem1)
...
push(elem4) -> resize()
pop() -> elem4
Stack Example

```java
public class Stack {
    int[] values = new int[3];
    int size = 0;

    void push(int x) {
        if (size >= values.length) {
            resize(); }
        values[size++] = x;
    }

    int pop() {
        if (size > 0) {
            return values[size--]; }
        throw new EmptyStackException();
    }

    private void resize() {
        int[] tmp = new int[values.length * 2];
        for (int i = 0; i < values.length; i++) {
            tmp[i] = values[i]; }
        values = tmp;
    }
}
```

100% branch coverage with:

```java
Stack stack0 = new Stack();
try {
    stack0.push(int0);
    stack0.push(int0);
    stack0.push(int0);
    stack0.pop();
} catch(EmptyStackException e) {
}

Stack stack0 = new Stack();
int int(0) = -55;
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.pop();
```
Stack Example – CFGs

```java
public class Stack {
    int[] values = new int[3];
    int size = 0;

    void push(int x) {
        if (size >= values.length) {
            resize(); }
        values[size++] = x;
    }
    int pop() {
        if (size > 0) {
            return values[size--]; }
        throw new EmptyStackException();
    }
    private void resize() {
        int[] tmp = new int[values.length * 2];
        for (int i = 0; i < values.length; i++) {
            tmp[i] = values[i]; }
        values = tmp;
    }
}
```
Stack Example – CFGs

```java
public class Stack {
    int[] values = new int[3];
    int size = 0;
    
    void push(int x) {
        if (size >= values.length) {
            resize();
        }
        values[size++] = x;
    }
    
    int pop() {
        if (size > 0) {
            return values[size--];
        }
        throw new EmptyStackException();
    }
    
    private void resize() {
        int[] tmp = new int[values.length * 2];
        for (int i = 0; i < values.length; i++) {
            tmp[i] = values[i];
        }
        values = tmp;
    }
}
```

McCabe:
Push(): 1+1=2 or 5-5+2*1=2
Pop(): 1+1=2 or 5-5+2*1=2
Resize(): 1+1=2 or 4-4+2*1=2

Total: 2+2+2=6 or 14-14+2*3=6

2nd version:

```
#edges-#nodes+2*#programs
(program==method)
```
### Stack Example – CFGs

```java
public class Stack {
    int[] values = new int[3];
    int size = 0;

    void push(int x) {
        if (size >= values.length) {
            resize();
        }
        values[size++] = x;
    }

    int pop() {
        if (size > 0) {
            return values[size--];
        }
        throw new EmptyStackException();
    }

    private void resize() {
        int[] tmp = new int[values.length * 2];
        for (int i = 0; i < values.length; i++) {
            tmp[i] = values[i];
        }
        values = tmp;
    }
}
```

McCabe:
- Push(): 1+1=2 or 5-5+2*1=2
- Pop(): 1+1=2 or 5-5+2*1=2
- Resize(): 1+1=2 or 4-4+2*1=2

Total: 2+2+2=6 or 14-14+2*3=6

2nd version:

```java
private int f() {
    int tmp[] = new int[values.length];
    for (int i = 0; i < values.length; i++) {
        tmp[i] = values[i];
    }
    values = tmp;
}
```
public class Stack {
    int[] values = new int[3];
    int size = 0;

    void push(int x) {
        if (size >= values.length) {
            resize();
        }
        values[size++] = x;
    }
    int pop() {
        if (size > 0) {
            return values[size--];
        }
        throw new EmptyStackException();
    }
    private void resize() {
        int[] tmp = new int[values.length * 2];
        for (int i = 0; i < values.length; i++) {
            tmp[i] = values[i];
        }
        values = tmp;
    }
}
public class Stack {
    int[] values = new int[3];
    int size = 0;

    void push(int x) {
        if (size >= values.length) {
            resize();
        }
        values[size++] = x;
    }

    int pop() {
        if (size > 0) {
            return values[size--];
        } else {
            throw new EmptyStackException();
        }
    }

    private void resize() {
        int[] tmp = new int[values.length * 2];
        for (int i = 0; i < values.length; i++) {
            tmp[i] = values[i];
        }
        values = tmp;
    }
}

McCabe: 22-18+2*1=6

#edges-#nodes+2*#programs (program==class)
Stack Example – CFG (class)

```java
public class Stack {
    int[] values = new int[3];
    int size = 0;

    void push(int x) {
        if (size >= values.length) {
            resize();
        }
        values[size++] = x;
    }

    int pop() {
        if (size > 0) {
            return values[size--];
        }
        throw new EmptyStackException();
    }

    private void resize() {
        int[] tmp = new int[values.length * 2];
        for (int i = 0; i < values.length; i++) {
            tmp[i] = values[i];
        }
        values = tmp;
    }

    // McCabe:
    // 22-18+2*1=6

    // #edges-#nodes+2*#programs (program==class)
}
```

McCabe: 22-18+2*1=6
Stack Example

```
Stack stack0 = new Stack();
try {
    stack0.pop();
} catch (EmptyStackException e) {
}
```

100% branch coverage with:

```
Stack stack0 = new Stack();
try {
    stack0.pop();
} catch (EmptyStackException e) {
}
```

```
Stack stack0 = new Stack();
int int0 = -55;
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.pop();
```
Stack Example

Stack

Stack()

push(x)

for

ture
false

pop()

true
false

100% branch coverage with:

Stack stack0 = new Stack();
try {
    stack0.pop();
} catch(EmptyStackException e) {
}

Stack stack0 = new Stack();
int int0 = -55;
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.pop();
Stack Example

Stack

Stack()

if (size=0)

true

false

pop()

push(x)

if

true

false

resize()

for

true

false

Stack Example

100% branch coverage with:

Stack stack0 = new Stack();
try {
    stack0.pop();
} catch (EmptyStackException e) {
}

Stack stack0 = new Stack();
int int0 = -55;
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.pop();
Stack Example

100% branch coverage with:

```
Stack stack0 = new Stack();  // T1
try {
    stack0.pop();
} catch(EmptyStackException e) {
}

Stack stack0 = new Stack();  // T2
int int0 = -55;
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.pop();
```
Stack Example

100% branch coverage with:

```java
Stack stack0 = new Stack();
try {
    stack0.pop();
} catch (EmptyStackException e) {
}
```

```java
Stack stack0 = new Stack();
int int0 = -55;
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.pop();
```
Stack Example

100% branch coverage with:

```
Stack stack0 = new Stack();
try {
    stack0.pop();
} catch (EmptyStackException e) {
}
```

```
Stack stack0 = new Stack();
int int(0) = -55;
stack0.push(int(0));
stack0.push(int(0));
stack0.push(int(0));
stack0.push(int(0));
stack0.pop();
```
Stack Example

```java
Stack stack0 = new Stack();
try {
    stack0.pop();
} catch (EmptyStackException e) {
}
```

100% branch coverage with:

```
Stack stack0 = new Stack();
for (int x = 0; x < 4; x++) {
    stack0.push(x);
}
stack0.pop();
```

```
Stack stack0 = new Stack();
try {
    stack0.pop();
} catch (EmptyStackException e) {
}
```

```
Stack stack0 = new Stack();
int int0 = -55;
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.pop();
```
Stack Example

100% branch coverage with:

Stack stack0 = new Stack();
try {
    stack0.pop();
} catch (EmptyStackException e) {
}

Stack stack0 = new Stack();
int int0 = -55;
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.pop();
Stack Example

```java
public class Stack {
    int[] values = new int[3];
    int size = 0;

    void push(int x) {
        if (size >= values.length) {
            resize(); }
        values[size++] = x;
    }

    int pop() {
        if (size > 0) {
            return values[size--]; }
        throw new EmptyStackException();
    }

    private void resize() {
        int[] tmp = new int[values.length * 2];
        for (int i = 0; i < values.length; i++) {
            tmp[i] = values[i];
        }
        values = tmp;
    }
}
```

Specification:
- When Stack has been created, it is empty
- When elements are added (push), the stack is not empty
- When Stack is full and an element added, it will be resized
- When the last element has been removed (pop) from Stack, it is empty
- When Stack is empty and an element shall be removed, an error occurs.
State Diagram for Stack Example

- **non_exist**
  - **Stack()**
  - **push(x)**
    - **not_empty** (size = 0)
      - **pop()**
        - **error**
      - **push(x)**
        - **not_empty** (0 < size <= length)
          - **[size = 1] pop() / size--**
          - **[size > 1] pop() / size--**
        - **not_empty** (size = length)
          - **push(x)**
            - **resize(); size++**

**Note:** resize() is a private method

length := length*2
Stack stack0 = new Stack();
try {
    stack0.pop();
} catch(EmptyStackException e) { }

Note: resize() is a private method
length := length*2
non_exist

Stack() → empty
(size = 0)

not_empty
(0 < size <= length)

pop() → error

[size < length] push(x) / size++

[size = length] push(x) / resize(); size++

Note: resize() is a private method

length := length*2

Stack stack0 = new Stack();
int int0 = -510;
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.pop();
T3
(to cover all state-transitions)

non_exist

Stack()

empty
(size = 0)

pop()

error

not_empty

(push(x)

size < length)

push(x) / size++

size = 1

pop() / size--

[size > 1] pop() / size--

size = length

push(x) /
resize(); size++

Note: 
resize() is a private method
length := length*2
Merge T1 / T2 / T3 → T1*

Stack stack0 = new Stack();
try {
    stack0.pop();
} catch (EmptyStackException e) {
}

Stack stack0 = new Stack();
int int(0) = -55;
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.pop();
stack0.pop();
stack0.pop();
stack0.pop();
try {
    stack0.pop();
} catch (EmptyStackException e) {
}

Stack stack0 = new Stack();
int int(0) = -55;
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.pop();
stack0.pop();
stack0.pop();
stack0.pop();
try {
    stack0.pop();
} catch (EmptyStackException e) {
}
Merge T1 / T2 / T3 ➔ T1*

Stack stack0 = new Stack();  \hspace{1cm} T1
try {
    stack0.pop();
} catch(EmptyStackException e) {
}

Stack stack0 = new Stack();  \hspace{1cm} T2
int int(0) = -55;
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.pop();

Stack stack0 = new Stack();  \hspace{1cm} T3
int int(0) = -55;
stack0.push(int0);
stack0.pop();

NON_EXIST -> Stack() -> EMPTY -> pop() -> ERROR
[length = 5]

NON_EXIST -> Stack() -> EMPTY -> push() ->
NOT_EMPTY -> push() -> NOT_EMPTY -> push() ->
NOT_EMPTY -> push() -> NOT_EMPTY -> pop() ->
NOT_EMPTY
[length = 13]

NON_EXIST -> Stack() -> EMPTY -> push() ->
NOT_EMPTY -> pop() -> EMPTY
[length = 7]
Merge T1 / T2 / T3 ➔ T1*

NON_EXIST -> Stack() -> EMPTY -> push() ->
NOT_EMPTY -> push() -> NOT_EMPTY -> push() ->
NOT_EMPTY -> push() -> NOT_EMPTY -> pop() ->
NOT_EMPTY -> pop() -> NOT_EMPTY -> pop() ->
NOT_EMPTY -> pop() -> EMPTY -> pop() -> ERROR

[length = 21]

Stack stack0 = new Stack();
int int0 = -55;
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.pop();
stack0.pop();
stack0.pop();
stack0.pop();
try {
    stack0.pop();
} catch (EmptyStackException e) { }
Structure of Lecture 5

• State-Transition Testing
• Random Testing
• Certification Testing
• Metamorphic Testing
• Lab 5
Random Testing (RT) is …

• “a black-box software testing technique where programs are tested by generating random, independent inputs.

• Results of the output are compared against software specifications to verify that the test output is pass or fail.

• In case of absence of specifications the exceptions of the language are used … .”

Source: https://en.wikipedia.org/wiki/Random_testing
Random Testing – Definitions

Random Testing (RT) is …

Monkey Testing
Random Number Generators

Random Number Generator (RNG)
- By default should draw from uniform distribution; each draw should be independent from the previous

*NB: Sometimes, though, it’s better to draw from a known distribution (e.g., usage profiles)*

Examples of RNGs
- True random number generators, e.g., atmospheric noise (see [www.random.org](http://www.random.org))
- Pseudo-random number generators, e.g., using linear congruential formula (LCF)
Linear Congruential Generators

LCF:

\[ X_{(n+1)} = (a \times X_n + c) \mod m \]

- \( m \): \( 0 < m \) – the “modulus”
- \( a \): \( 0 < a < m \) – the “multiplier”
- \( c \): \( 0 \leq c < m \) – the “increment”
- \( X_0 \): \( 0 \leq X_0 < m \) – the “seed”
Linear Congruential Generators

In Java: java.util.Random

```java
public int nextInt() {
    return next(32);
}

synchronized protected int next(int bits) {
    seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
    return (int)(seed >>> (48 - bits));
}
```
Linear Congruential Generators

In Java: java.util.Random

```java
public int nextInt() {
    return next(32);
}
```

```java
synchronized protected int next(int bits) {
    seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
    return (int)(seed >>> (48 - bits));
}
```

Called n=10 times
Max. 32 bits
Linear Congruential Generators

In Java: `java.util.Random`

```java
public int nextInt() {
    return next(31);
}
```

```java
synchronized protected int next(int bits) {
    seed = (seed * 0x5DEECE66DL + 0xB) & ((1L << 48) - 1);
    return (int)(seed >>> (48 - bits));
}
```

Called \(n=10\) times
Max. 31 bits & pos.
Linear Congruential Generators

In Java: `java.util.Random`

```java
int n = 10;
int m = 10; // modulus

Random ran_a = new Random();
// pseudo-random numbers between 0 and m-1
for (int i = 0; i < n; i++) {
    int r_a = ran_a.nextInt(m);
    System.out.println("r_a is: "+r_a);
}
```

Pseudo-random numbers between 0 and m-1
Example: MyAbs()

```java
private static int myAbs(int x) {
    if (x > 0) {
        return x;
    }
    else {
        return x; // fault: should be '-x'
    }
}
```
Example: MyAbs()

```java
private static int myAbs(int x) {
    if (x > 0) {
        return x;
    }
    else {
        return x; // fault: should be '-x'
    }
}
```

Failure Rate: FR = |failing tests| / |all possible tests| = 0.5 (roughly)*

FR is the probability of a test case to trigger a failure (failing test)

* assuming valid input data
Example: MyAbs()

```java
private static int myAbs(int x) {
    if (x > 0) {
        return x;
    }
    else {
        return x;  // fault: should be '-x'
    }
}
```

Failure Rate: $FR = \frac{|\text{failing tests}|}{|\text{all possible tests}|} = 0.5$ (roughly)*

FR is the probability of a test case to trigger a failure (failing test)

* assuming valid input data
Example: MyAbs()

```java
int n = 10;
int result;
Random random = new Random();
for (int i=0; i<n; i++) {
    int x = random.nextInt();
    result = myAbs(x);
    System.out.print("x is: "+x+" abs(x) is:"+result);
    if (result<0) {System.out.println(" >>> Error!");}
    else {System.out.println(" ");}
}

private static int myAbs(int x) {
    if (x > 0) {
        return x;
    }
    else {
        return x; // should be '-x'
    }
}
```

Failure Rate: FR = 0.5 (roughly)
FR is the probability of a test case to trigger a failure (failing test)
Example: MyAbs()

Known Oracle

```java
private static int myAbs(int x) {
    if (x > 0) {
        return x;
    }
    else {
        return x; // should be '-x'
    }
}
```

Failure Rate: FR = 0.5 (roughly)
FR is the probability of a test case to trigger a failure (failing test)
Example: MyAdd()

```java
private static int myAdd(int x, int y) {
    if ( x - 1 <= 0 ) {
        if ( y - 1 <= 0 ) {
            if ( x + y - 2 == 0 ) {
                System.out.println(">>> Crash!!!!");
            }
        }
    }
    return(x + y);
}
```

**Question:**
If we sample x and y from [-10,9], what is the expected failure rate?
Example: MyAdd()

```java
private static int myAdd(int x, int y) {
    if ( x - 1 <= 0 ) {
        if ( y - 1 <= 0 ) {
            if ( x + y - 2 == 0 ) {
                System.out.println(">>>> Crash!!!!");
            }
        }
    }
    return (x + y);
}
```

Failure Rate: FR = 1 / (20*20) = 0.0025*

* assuming valid input data and range [-10,9]
Example: MyAdd()

```java
private static int myAdd(int x, int y) {
    if (x - 1 <= 0) {
        if (y - 1 <= 0) {
            if (x + y - 2 == 0) {
                System.out.println(">>> Crash !!!");
            }
        }
    }
    return (x + y);
}
```

Failure Rate: FR = 1 / (20*20) = 0.0025*

* assuming valid input data and range [-10,9]
RT Challenges

• What if we have complex data structures?
  – How to generate data for those?

• What if we need to execute sequences of test steps per test?
  – How to configure test sequences?

• What if test execution is slow?
  – How to select test cases from the generated ones?
RT Challenges

• What if we have complex data structures?
  – How to generate data for those?

  One approach:
  
  - Represent data structure as bit strings
  - Then use bit strings from RNG

  Problem: might generate much invalid input
RT Challenges

- What if we have complex data structures?
  - How to generate data for those?

- What if we need to execute sequences of test steps per test?
  - How to configure test sequences?

Usually, tools do this (see Lab 5):

- Issue: Unclear whether many ‘short’
  test sequences are better than few long
  test sequences
RT Challenges

One approach:

- Select most diverse test inputs

Motivation:

- \(X_1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}\)
- \(X_2 = \{-2345, 12, 342, -4443, 2, 3495437, -222223, 24, 99343256, -524474\}\)

Which one is better? Equal probability of finding a failure if no further assumptions are made → However, often failures cluster in certain areas

- Use distance measures, e.g., Euclidean distance, Hammond distance

- What if test execution is slow?
  - How to select test cases from the generated ones?
RT Challenges

Example: Diversity between test sequences
- Container c = new Container();
- c.insert(5);
- c.insert(2);
- c.remove(5);
- assertEquals(c.size(),1);

Convert to string: (n, -, ins, 5, ins, 2, rem, 5) -> s1 = “A_BCBDEC”
- Assume s2 = “A_BFEF__” → Hammond distance equals 5 (= # differences)

• What if test execution is slow?
  – How to select test cases from the generated ones?
RT Tools

- **Randoop** - generates sequences of methods and constructor invocations for the classes under test and creates JUnit tests from these.

- **QuickCheck** - a famous test tool, originally developed for Haskell but ported to many other languages, that generates random sequences of API calls based on a model and verifies system properties that should hold true after each run.

- **Simulant** - a Clojure tool that runs simulations of various agents (e.g., users with different behavioral profiles) based on a statistical model of their behavior, recording all the actions and results into a database for later exploration and verification.

- **AutoTest** - a tool integrated to EiffelStudio testing automatically Eiffel code with contracts based on the eponymous research prototype.

- **York Extensible Testing Infrastructure** (YETI) - a language agnostic tool which targets various programming languages (Java, JML, CoFoJa, .NET, C, Kermeta).

- **GramTest** - a grammar based random testing tool written in Java, it uses BNF notation to specify input grammars.

(Source: Wikipedia)
RT Summary

- Cheap, easy to understand method to produce test data
  - Enhances BBT (i.e., ECP testing)
  - Complements WBT
- Might be useful for reliability testing & test stop criteria
- Actually used in industry / tools exist

- Downsides:
  - Requires an oracle (or can only do crash testing)
  - Statement coverage is usually low (randoop: 50-70%)
Fuzzing

Definition:

• Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.

  (source: Wikipedia)

• Recommended online source: https://www.fuzzingbook.org
  – The Fuzzing Book - Tools and Techniques for Generating Software Tests
  – by Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler
Structure of Lecture 5

• State-Transition Testing
• Random Testing
• Certification Testing
• Metamorphic Testing
• Lab 5
RT and Reliability Certification

• How to decide that a component (SUT) has sufficient reliability?

• Example:

Use Random Testing (RT) and count occurrence of failures
What is Reliability?

- **Reliability definitions (Wikipedia):**
  - The idea that an item is fit for a purpose with respect to time
  - The capacity of a designed, produced, or maintained item to perform as required over time
  - The capacity of a population of designed, produced or maintained items to perform as required over time
  - The resistance to failure of an item over time
  - The probability of an item to perform a required function under stated conditions for a specified period of time
  - The durability of an object
Reliability Certification Testing Process

5 Steps:

1. Define the reliability objective
2. Define the usage model and usage profile (operational profile)
3. Specify test cases
4. Execute certification test
5. Certify software component
Reliability Objective $\lambda_{obj}$

- Usually, the **reliability objective** $\lambda_{obj}$ is defined as the **desired maximal level of failure intensity** ($\lambda_F$) encountered during operation
  - Failure intensity ($\lambda_F$) is the inverse of Mean-Time-Between-Failure (MTBF)

- In the context of certification testing, **failure intensity** is measured in terms of number of failures per test intensity (or test time or test effort) unit
  - Example test intensity units: e.g. CPU hour, test person hour, number of test cases, etc.
Reliability Objective $\lambda_{\text{obj}}$ – Examples

- Typical values of reliability objectives are listed below; they are derived from the estimated impact (damage expressed in terms of $\$, and in terms of number of deaths) induced by a failure (Musa, 1998).

<table>
<thead>
<tr>
<th>Failure impact</th>
<th>Typical failure intensity target value [failure/h]</th>
<th>Time between failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hundreds of deaths, more than $10^9$ $$ cost</td>
<td>$10^{-9}$</td>
<td>114,000 years</td>
</tr>
<tr>
<td>One or two deaths, more than $10^6$ $$ cost</td>
<td>$10^{-6}$</td>
<td>114 years</td>
</tr>
<tr>
<td>Around 1000 $$ cost</td>
<td>0.001</td>
<td>6 weeks</td>
</tr>
<tr>
<td>Around 100 $$ cost</td>
<td>0.01</td>
<td>100 hours</td>
</tr>
<tr>
<td>Around 10 $$ cost</td>
<td>0.1</td>
<td>10 hours</td>
</tr>
<tr>
<td>Around 1 $$ cost</td>
<td>1</td>
<td>1 hours</td>
</tr>
</tbody>
</table>
Reliability Demo Chart

- Reliability goals are often stated in terms of Failure Intensity Objectives (FIO)
- Usually: Failure Intensity represents the number of Failures observed in a defined time period.
- Using a Reliability Demonstration Chart is an efficient way of checking whether the FIO ($\lambda_{obj}$) is met or not.
- It is based on collecting failure data.
  - **Vertical axis:** failure number ($n$)
  - **Horizontal axis:** expected number of failures (or: normalized failure data ($T_n$), i.e., failure time $\times \lambda_{obj}$)

$$\text{Observed number of failures} = \text{Expected number of failures}$$

Musa (1977)
How to Define Reject, Continue, Accept Regions?

- The reject, continue, accept regions for a defined reliability objective (FIO) are based on sequential sampling theory.
- Procedure:
  1. Select the discrimination ratio $\gamma$ with which the certification test will be performed;
  2. Select the supplier (or developer) risk $\alpha$, i.e. the probability of falsely deciding that the reliability objective is not met when it is;
  3. Select the consumer (or customer) risk $\beta$, i.e. the probability of falsely deciding that the reliability objective is met when it is not.
How to Define Reject, Continue, Accept Regions?

\[ T_n = \frac{B}{1-\gamma} - \frac{\ln \gamma}{1-\gamma} n \quad \text{Boundary between reject and continue regions} \]

\[ T_n = \frac{A}{1-\gamma} - \frac{\ln \gamma}{1-\gamma} n \quad \text{Boundary between accept and continue regions} \]

(\( \gamma \) is the discrimination ratio)

\[ A = \ln \frac{\beta}{1-\alpha} \quad B = \ln \frac{1-\beta}{\alpha} \]
Reliability Demo Chart – Effects of $\alpha$, $\beta$ and $\gamma$

- When risk levels ($\alpha$ and $\beta$) decrease, ...

  or

- When discrimination ratio ($\gamma$) decreases, ...

- ... the system will require more testing before reaching the *Accept* or *Reject* regions

  - i.e., the *Continue* region gets wider.
RDC: Example

- Consumer risk $\beta = 0.05$
- Supplier risk $\alpha = 0.05$
- Discrimination ratio $\gamma = 2$
RDC: Example /2

- Consumer risk $\beta = 0.01$
- Supplier risk $\alpha = 0.01$
- Discrimination ratio $\gamma = 2$
RDC: Example

- Consumer risk \( \beta = 0.001 \)
- Supplier risk \( \alpha = 0.001 \)
- Discrimination ratio \( \gamma = 2 \)
RDC: Example

- Consumer risk \( \beta = 0.1 \)
- Supplier risk \( \alpha = 0.1 \)
- Discrimination ratio \( \gamma = 1.2 \)
Example 1

<table>
<thead>
<tr>
<th>Failure number</th>
<th>Measure (million tests)</th>
<th>Normalized Measure (= expected Failure number)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1875</td>
<td>0.75</td>
</tr>
<tr>
<td>2</td>
<td>0.3125</td>
<td>1.25</td>
</tr>
<tr>
<td>3</td>
<td>1.25</td>
<td>5</td>
</tr>
</tbody>
</table>

\[ \lambda_{\text{obj}} = \frac{4 \text{ failures}}{1 \text{ million tests}} \]

\[ \alpha = 0.1 \]

\[ \beta = 0.1 \]

\[ \gamma = 2 \]
Example 2

<table>
<thead>
<tr>
<th>Failure number</th>
<th>Measure (CPU hour)</th>
<th>Normalized Measure (= expected Failure number)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>0.8</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
<td>1.9</td>
</tr>
<tr>
<td>3</td>
<td>60</td>
<td>6</td>
</tr>
</tbody>
</table>

\[
\lambda_{\text{obj}} = 0.1 \text{ failures / CPU hour}
\]

\[
\alpha = 0.05
\]

\[
\beta = 0.05
\]

\[
\gamma = 2
\]
**Example 3**

We have developed a program for a Web server with a target failure intensity of 1 failure/1,000,000 tests. The program runs for 50 hours, handling 10,000 tests per hour on average, with no failures occurring. How confident are we that the program has met its objective? Can we release the software now?

\[
\lambda_{\text{obj}} = \frac{1\text{ failure}}{10^6\text{ tests}} \\
\alpha = 0.1 \quad \beta = 0.1 \quad \gamma = 2
\]
Example 3

<table>
<thead>
<tr>
<th>Failure number</th>
<th>Measure (tests)</th>
<th>Normalized Measure (= expected Failure number)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ?</td>
<td>500,000</td>
<td>0.5</td>
</tr>
</tbody>
</table>

$\lambda_{obj} = 1 \text{ failure} / (10^6 \text{ tests})$

$\alpha = 0.1 \quad \beta = 0.1 \quad \gamma = 2$
Certification Testing Summary

- Well-established method in systems development
- Can be combined with RT and Statistical Testing
- Works best for Robustness Testing (Crash Testing)

Challenges:
- Results are sensitive to the choice of the test intensity measure
- Works better for systems that have not too high reliability requirements
Structure of Lecture 5

- State-Transition Testing
- Random Testing
- Certification Testing
- Metamorphic Testing
- Lab 5
Part 1 is shown next
Part I: Introduction to Metamorphic Testing

- Introduction
- State of the art
- Lessons learned
- Challenges
Introduction

Input → System Under Test (SUT) → Output
Introduction

Input → System Under Test (SUT) → Output

Correct?
**Introduction**

**Test oracle**
Mechanism to decide whether a test output is correct or not.

Input → System Under Test (SUT) → Output

Correct?
Introduction

Oracle problem
Sometimes it is not feasible to check the correctness of a test output.

Scientific calculations
Artificial intelligence
Simulation and modelling
Introduction
The oracle problem - Examples

\[ \text{shortestPath}(G, s, d) \rightarrow \{e, g, h, t, x, z\} \]

Source = s
Destination = d

Introduction
The oracle problem - Examples

Introduction
The oracle problem - Examples

Introduction
The oracle problem - Examples

Introduction
Metamorphic testing - Examples

Source test case

Graph G
Source s
Destination d

\[ \text{shortestPath}(G, s, d) \]

\{e, g, h, t, x, z\}
Introduction
Metamorphic testing - Examples

Source test case

Graph G
Source s
Destination d

\[ |\text{shortestPath}(G,s,d)| = |\text{shortestPath}(G,d,s)| \]

shortestPath(G,s,d)

\{e,g,h,t,x,z\}
Introduction
Metamorphic testing - Examples

Source test case

Graph G
Source s
Destination d

\[ \text{shortestPath}(G, s, d) \]

\{e, g, h, t, x, z\}

Follow-up test case

Graph G
Source d
Destination s

\[ |\text{shortestPath}(G, s, d)| = |\text{shortestPath}(G, d, s)| \]

\[ \text{shortestPath}(G, d, s) \]
Introduction
Metamorphic testing - Examples

Source test case

Graph G
Source s
Destination d

\[ \text{shortestPath}(G, s, d) \]

\{e, g, h, t, x, z\}

Follow-up test case

Graph G
Source d
Destination s

\[ \text{shortestPath}(G, d, s) \]

\[ \text{shortestPath}(G, s, d) \]

\{z, x, g, e\}
Introduction

Metamorphic testing - Examples

Source test case

Q₁ = “Software”

↓

flickr

↓

Search for software

↓

View all 1,272,925
Introduction
Metamorphic testing - Examples

Source test case

Q₁ = “Software”

Follow-up test case

Q₂ = “Software”, size=large

If Q₂ ≡ Q₁ AND size=large
then Count(Q₂) ≤ Count(Q₁)

View all 1,272,925
**Introduction**

Metamorphic testing - Examples

Source test case

\( Q_1 = \text{“Software”} \)

[Image of a search bar with the query “software” and the result “View all 1,272,925”]

Follow-up test case

\( Q_2 = \text{“Software”, size=large} \)

[Image of a search bar with the query “software” and the result “View all 1,353,878”]

If \( Q_2 \equiv Q_1 \) AND size=large then \( \text{Count}(Q_2) \leq \text{Count}(Q_1) \)
Introduction

Source test case

\[ X_1 \]

\[ \downarrow \]

\[ P \]

\[ \downarrow \]

\[ O_1 \]

Metamorphic relation

\[ R (x_1, x_2, o_1, o_2) \]

Follow-up test case

\[ X_2 \]

\[ \downarrow \]

\[ P \]

\[ \downarrow \]

\[ O_2 \]
Introduction

Source test case

\[ X_1 \]

\[ \Downarrow \]

\[ P \]

\[ \Downarrow \]

\[ O_1 \]

Metamorphic relation

\[ R \left( x_1, x_2, o_1, o_2 \right) \]

\[ \Downarrow \]

\[ P \]

\[ \Downarrow \]

\[ O_2 \]

Follow-up test case_1

Follow-up test case_2

Follow-up test case_n
Introduction
Metamorphic testing process

1. Identification of metamorphic relations.
Introduction
Metamorphic testing process

1. Identification of metamorphic relations.
2. Generation/Selection of source test cases.
Introduction
Metamorphic testing process

1. Identification of metamorphic relations.
2. Generation/Selection of source test cases.
3. Generation of follow-up test cases.
Introduction
Metamorphic testing process

1. Identification of metamorphic relations.
2. Generation/Selection of source test cases.
3. Generation of follow-up test cases.
4. Checking of metamorphic relations.
Oh, I get it. This is about **alleviating the oracle problem**. Is that it?

Yes! but MT can also support **test data generation**!
Introduction
Test data generation

|\text{shortestPath}(G,s,d)|
|\text{shortestPath}(G,d,s)|

Metamorphic relation

\begin{align*}
|\text{shortestPath}(G,k,t)| & = |\text{shortestPath}(G,t,k)| \\
|\text{shortestPath}(G,C,A)| & = |\text{shortestPath}(G,A,C)| \\
|\text{shortestPath}(G,2,41)| & = |\text{shortestPath}(G,41,2)|
\end{align*}

Graph database
**Introduction**

Test data generation

If $Q_2 \equiv Q_1$ AND size=large then $\text{Count}(Q_2) \leq \text{Count}(Q_1)$

- **MT 1**
  - $Q_1 = \text{“dog”}$
  - $Q_2 = \text{“dog”, size = large}$
  - $\text{Count}(Q_2) \leq \text{Count}(Q_1)$

- **MT 2**
  - $Q_1 = \text{“wind”}$
  - $Q_2 = \text{“wind”, size = large}$
  - $\text{Count}(Q_2) \leq \text{Count}(Q_1)$

- **MT 3**
  - $Q_1 = \text{“thing”}$
  - $Q_2 = \text{“thing”, size = large}$
  - $\text{Count}(Q_2) \leq \text{Count}(Q_1)$

- **MT 4**
  - $Q_1 = \text{“money”}$
  - $Q_2 = \text{“money, size = large}$
  - $\text{Count}(Q_2) \leq \text{Count}(Q_1)$

Random word generator
Part I: Introduction to Metamorphic Testing

- Introduction
- State of the art
- Lessons learned
- Challenges
A Survey on Metamorphic Testing

Sergio Segura, Member, IEEE, Gordon Fraser, Member, IEEE, Ana B. Sanchez, and Antonio Ruiz-Cortés

Abstract—A test oracle determines whether a test execution reveals a fault, often by comparing the observed program output to the expected output. This is not always practical, for example when a program’s input-output relation is complex and difficult to capture formally. Metamorphic testing provides an alternative, whereby correctness is determined by checking an individual concrete output, but by applying a transformation to a test input and observing how the program output “morphs” into a different one as a result. Since the introduction of strict metamorphic relations in 1998, many contributions on metamorphic testing have been made, and the technique has been successful applications in a variety of domains, ranging from web services to computer graphics. This article provides a comprehensive survey on metamorphic testing. It summarizes the research results and application areas, and analyses common practice in empirical studies of metamorphic testing as well as the main open challenges.

1 INTRODUCTION

Software testing is an essential but costly activity applied during software development to detect faults in programs. Testing consists of executing a program with test inputs, and to detect faults there needs to be some procedure by which testers can decide whether the output of the program is correct or not. A so-called test oracle. If the test oracle consists of comparing an expected output value with that of test cases. Metamorphic testing does not only alleviate the oracle problem, but it can also be highly automated.

The introduction of metamorphic testing can be traced back to a technical report by Chen et al. [5] published in 1998. However, the use of identity relations to check program outputs can be found in earlier articles on testing of numerical programs [6], [7], and fault tolerance [8]. Since its introduction.

output behaviour. The prototypical example is that of a program that computes the sine function: What is the exact value of sin(12)? Is an observed output of 0.5063 correct? A mathematical property of the sine function states that sin(12) = sin(12 - r), and we can use this to test whether sin(12) = sin(12 - r), without knowing the concrete values of either sine calculation. This is an example of a metamorphic relation: an input transformation that can be used to generate new test cases from existing test data, and an output relation, that compares the outputs produced by a pair

119 papers
Jan 1998 - Nov 2015
State of the art

Domains

- Embedded systems: 16%
- Web services and applications: 10%
- Computer graphics: 12%
- Simulation and modelling: 12%
- Machine learning: 7%
- Bioinformatics: 5%
- Variability and decision support: 7%
- Components: 3%
- Compilers: 3%
- Numerical programs: 4%
- Others: 21%
State of the art

Subject programs
State of the art
Types of faults

295 real bugs in 36 different tools
Example: DeepTest

Autonomous vehicle controlled via a Deep Neural Network (DNN):
- DNN input = pictures from a camera; DNN outputs = steering angles.
- Goal: Use MRs to verify the correctness of the outputs
- MRs stated that the car should behave similarly for variations of the same input (for example, the same scene under different lighting conditions).
- Using these MRs, generate realistic synthetic images based on seed images. These synthetic images mimic real-world phenomena such as camera lens distortions and different weather conditions.
- MT + a notion of neuron coverage (the number of neurons activated) helped find a large number of corner case inputs yielding erroneous behavior.

Source: https://deeplearningtest.github.io/deepTest
(Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray, ICSE 2018)
Part I: Introduction to Metamorphic Testing

- Introduction
- State of the art
- Lessons learned
- Challenges
Lessons learned

Lesson learned

Metamorphic testing requires good knowledge of the problem domain.
Lessons learned

Different metamorphic relations can have different fault-detection capability.

- MR₁
- MR₂
- MR₃
Lesson learned

Metamorphic relations should be diverse so they exercise different parts of the program.
Lessons learned

Metamorphic relations can be combined.

MR₁ + MR₃ + MR₆

MR₁, MR₂, MR₃, MR₄, MR₅, MR₆, ..., +
The automated discovery of metamorphic relations seems feasible in certain domains.
Part I: Introduction to Metamorphic Testing

- Introduction
- State of the art
- Lessons learned
- Challenges
Metamorphic Testing: A Review of Challenges and Opportunities

TSONG YUEH CHEN and FEI-CHING KUO, Swinburne University of Technology
HUAI LIU, Victoria University
PAK-LOK POON, RMIT University
DAVE TOWEY, University of Nottingham Ningbo China
T. H. TSE, The University of Hong Kong
ZHI QUAN ZHOU, University of Wollongong

Metamorphic testing is an approach to both test case generation and test result verification. A central element is a set of metamorphic relations, which are necessary properties of the target function or algorithm in relation to multiple inputs and their expected outputs. Since its first publication, we have witnessed a rapidly increasing body of work examining metamorphic testing from various perspectives, including metamorphic relation identification, test case generation, integration with other software engineering techniques, and the validation and evaluation of software systems. In this paper, we review the current research of metamorphic testing and discuss the challenges yet to be addressed. We also present visions for further improvement of metamorphic testing and highlight opportunities for new research.

CCS Concepts: →Software and its engineering: →Software verification and validation; Software text-

ACM Computing Surveys
To appear
Challenges

**Challenge**

Systematic guidelines for the construction of good metamorphic relations.
Challenges

Challenge

Generation of likely metamorphic relations.

Program → Metamorphic Relations? → Likely

MR1

MR2

MR3
Challenges

Challenge
Non-functional metamorphic testing.
Challenges

Non-functional metamorphic testing.
Challenges

Challenge

Provide tools to foster the use of the technique.
Structure of Lecture 5

• State-Transition Testing
• Random Testing
• Certification Testing
• Metamorphic Testing
• Lab 5
Lab 5 – Random Testing (with Randoop)

HW 5 (week 30: Mar 23 & 24) – Random Testing with Randoop (9 points)

Lab 5 Instructions & Tools

Submission Deadlines:
• Tuesday Labs: → Monday, 29 Mar, 23:59
• Wednesday Labs: → Tuesday, 30 Mar, 23:59

• Penalties apply for late delivery: 50% penalty, if submitted up to 24 hours late; 100 penalty, if submitted more than 24 hours late
Lab 5 – Random Testing (with Randoop)

Lab 5 (week 30: Mar 23 & 24) – Random Testing with Randoop (9 points)

Goals:
- Understand how to use Randoop
- Trigger failures
- Play with Randoop parameters
- Discussion / Pro’s & Con’s

Instructions

Randoop

NextDate Code
POS Code

Several SUT versions
Several Randoop settings
Next Weeks

• Quiz 5 (in Moodle!):
  • Opens at end of lecture – closes on Monday at 17:30am!

• Lab 4:
  – Basic White-Box Testing

• Lab 5:
  – Random Testing (with Randoop)

• Lecture 6:
  – Test Levels, Test Tools, Test Automation

Next Week!

After next Week!