The debugging part uses Andreas Zeller’s slides. This is extra. The “injecting” part is about mutation testing; this is discussed in Ch 5 of the book.
Mutants

• *Mutation*: changing a valid artifact, usually by a minimalistic amount, to produce an invalid version of it. The result: *mutant* (Def 5.47, different formulation)

• Application:
  – negative testing, to produce invalid inputs.
  – seeding errors in a program

• To *systematically* generate mutants, we introduce *mutation operators* (Def 5.46)
Example

If an input is described by a BNF grammar, we can systematically mutate the grammar to produce mutants for negative tests.

\[
\begin{align*}
\text{NLpostcode} &\rightarrow \text{Area Street} \\
\text{Area} &\rightarrow \text{FirstDigit Digit Digit Digit} \\
\text{Street} &\rightarrow \text{Letter Letter} \\
\text{FirstDigit} &\rightarrow 1 \mid 2 \ldots \\
\text{Digit} &\rightarrow 0 \mid 1 \mid 2 \ldots \\
\text{Letter} &\rightarrow a \mid b \mid c \ldots \mid A \mid B \mid C \ldots
\end{align*}
\]
Some coverage concepts for grammar-based mutants generation

• To define how much mutants to generate.
• Imagine a set of mutation operators can be applied on BNF production rules.
• (C 5.33) *Mutation Operator Coverage (MOC)*: for every mutop $\delta$, TR contains 1x mutant produced by $\delta$.
• (C5.34) *Mutation Production Cov*: for every production rule $r$ and every mutop $\delta$ applicable to $r$, TR contains one mutant produced by $\delta(r)$. 
Let’s mutate a program to “simulate” programming mistakes, resulting syntactically valid mutants, but containing mistakes.

We can use this to test your tests, to see how “effective” they are → called *mutation test*.

Considered to be very strong. Tools:

– Pitest, Major for Java
– NinjaTurtles (beta) for C#
– MuCheck for Haskell
Example

<table>
<thead>
<tr>
<th>Original Method</th>
<th>With Embedded Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>int Min (int A, int B)</td>
<td>int Min (int A, int B)</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>int minVal;</td>
<td>int minVal;</td>
</tr>
<tr>
<td>minVal = A;</td>
<td>minVal = A;</td>
</tr>
<tr>
<td>if (B &lt; A)</td>
<td>if (B &lt; A)</td>
</tr>
<tr>
<td>{</td>
<td>Δ1 minVal = B;</td>
</tr>
<tr>
<td>minVal = B;</td>
<td>if (B &gt; A)</td>
</tr>
<tr>
<td>}</td>
<td>Δ2 if (B &gt; A)</td>
</tr>
<tr>
<td>return (minVal);</td>
<td>Δ3 if (B &lt; minVal)</td>
</tr>
<tr>
<td>} // end Min</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>minVal = B;</td>
</tr>
<tr>
<td></td>
<td>Δ4 <strong>Bomb();</strong></td>
</tr>
<tr>
<td></td>
<td>Δ5 minVal = A;</td>
</tr>
<tr>
<td></td>
<td>Δ6 minVal = failOnZero (B);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>return (minVal);</td>
</tr>
<tr>
<td></td>
<td>} // end Min</td>
</tr>
</tbody>
</table>

(mutations generated by the tool Mothra)
Typical Tooling Setup

This is a simpler variation of the scheme in Fig. 5.2
Killing a mutant

\[ P(x) \{ \text{if } (x < 0) \{ y = x ; \text{return } x \} \}
\]
\[ y = 1 ; \text{return } 0 \} \]

\[ P(x) \{ \text{if } (x \leq 0) \{ y = x ; \text{return } x \} \}
\]
\[ y = 1 ; \text{return } 0 \} \]

- (D 5.50) Let \( P \) be the original program. A test \( t \) of \( P \) \textit{weakly kills} a mutant \( P' \) of \( P \) if executing \( t \) on them results in different (internal) states, immediately after the mutation spot.

- (D 5.50) The test \( t \) \textit{strongly kills} \( P' \) if executing it on \( P \) and \( P' \) produces different test outcomes.
Mutation-based coverage criteria

- (C5.32) Given a set $M$ of mutants, the TR is simply $M$: you have to “kill” every mutant in $M$.
- OA remark that in practice strong vs weak do not make much difference → careful, this depends on the oracles used. In particular if you use partial oracles, e.g. in property-based testing, they may actually make much difference.
- The strength depends on the choice of mutops. See 5.2.2 and 5.3.2.
- Preferably you want mutops that simulate realistic errors.
- Also, try to minimize the mutops, or else mutation testing becomes too expensive.
Mutation operators

• (D5.51) Given a set $O_2$ of mutation operators; a subset $O_1$ is effective if test-cases designed specifically to kill mutants produced by $O_1$ will (very likely) also kill mutants from $O_2/O_1$.

• The smallest set of effective mutops? Hard to know, and very situation specific.

• Still, there have been plenty of research to see if there are some general patterns. For mutations over imperative constructs, see 5.2.2. Mutations on OO aspects, see 5.3.2.
Mutops on imperative constructs (5.2.2)

• **ABS (Absolute Value Insertion)** Modify each arithmetic (sub)expression by using functions `abs()`, `negAbs()`, and `failOnZero()`.

• **AOR (Arithmetic Operator Replacement)** Replace each occurrence of arithmetic operators `+`, `−`, `*`, `/`, `%` by each other; and in addition, by `leftOp`, and `rightOp`.

• **SVR (Scalar Variable Replacement)** Replace each variable reference by another (type compatible and in-scope) variable.

• **BSR (Bomb Statement Replacement)** Replace each statement by `Bomb()`.

• More... see book.
OO-related Mutops

- 5.3.3 additionally lists 20 mutops targeting typical OO aspects, e.g.:
  - Inheritance: method overriding, variables shadowing, constructors overriding works differently
  - Polymorphism
  - Use of static members
- I will only show some, as examples
# OO-related Mutops

## AOC – Argument Order Change

<table>
<thead>
<tr>
<th>Point3D</th>
<th>void set (int x, int y, char c);</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>void set (char a, int x, int y);</td>
</tr>
</tbody>
</table>

```java
Point3D p;
p.set (1, 2, 't');
Δ p.set ('t', 1, 2);
```

## ANC – Argument Number Change

<table>
<thead>
<tr>
<th>Point3D</th>
<th>void set (int x, int y, int z);</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>void set (int x, int y);</td>
</tr>
</tbody>
</table>

```java
Point3D p;
p.set (1, 2, 3);
Δ p.set (2, 3);
```
OO-related Mutops

• OMD (*Overriding Method Deletion*): delete an overriding method.
• OMM (*Overridden Method Moving*): move a call to a super.method to the first or last position, or up and down one statement.

```java
m() {
    x ++ ;
    y = y/x ;
    super.m();
}
```
OO-related Mutops

HVD – Hiding Variable Deletion

HVI – Hiding Variable Insertion

point

int x;
int y;

colorpoint

int x;
// int x;
int y;
// int y;

Δ1

Δ2

Δ1 int x;
Δ2 int y;

point

int x;
int y;

colorpoint

int x;
// int x;
int y;
// int y;

Δ1

Δ2
OO-related Mutops

ATC – Actual Type Change

Point p ;
  p = new Point();
Δ p = new Point3D();

DTC – Declared Type Change

Point p ;
Δ Point3D p ;
p = PointFactory();
Debugging

• Suppose we observe a program “fails”. Debugging: find the root cause of the failure (the error made in the program).
• Then we can fix the program.
• Very time consuming, especially at the system-level.
• How about automation?
Simplification

- Debugging = find an item of some property, in a given search space.
- It can at least be made easier if the "search space" is shrunk (simplification).
- Can we simplify systematically?
- E.g. binary search comes to mind....
Let’s First Re-express the Problem

• More abstractly, so that we can later apply the solution to different concrete situations.
• A test-case is abstractly represented by a configuration $c$, which is made up of “units” $\delta_1, \delta_2, \ldots, \delta_n$ organized in some structure (e.g. a list of units, or a tree of units).
• For simplicity, here we assume $c$ is a set of units. So: $c = \{\delta_1, \delta_2, \ldots, \delta_n\}$
Configurations

- \( \text{test}(c) \) executes the configuration \( c \), resulting either √ (pass), ✗ (fail), or ? (the configuration is invalid/not allowed).
- Configurations can be compared to decide which one is “simpler”. Since we said a configuration is a set, we use \( c_1 \subseteq c_2 \).
- \textit{Minimalization:} if \( c_\times \) is the initial failing configuration, \( (\text{test}(c_\times)= \times ) \), find the smallest subset \( c \) of \( c_\times \) such that \( \text{test}(c)= \times \).
- Expensive...
Simplification

• *Simplification*: find a “relevant” configuration.

• If $c_\times$ is the initial failing configuration ($\text{test}(c) = \times$), a subset $c$ of $c_\times$ is *relevant* if $\text{test}(c) = \times$ and if removing any unit from $c$ causes $\times$ to disappear. That is: $\forall \delta: \delta \in c : \text{test}(c/\{\delta\}) \neq \times$

• Note that a relevant configuration does not have to be minimal.
Binary Strategy

• Let \( c_\star \) be the initial failing configuration.
• A binary \( simplify(c_\star) \) is defined as follows:
  1. \( simplify(c) = \)
  2. \( \text{if } |c| = 1 \text{ then return } c \quad \text{// minimal} \)
  3. \( \text{split } c \text{ to disjoint } c_1 \cup c_2 \)
  4. \( \text{if } test(c_1) = \times \text{ then return } simplify(c_1) \)
  5. \( \text{if } test(c_2) = \times \text{ then return } simplify(c_2) \)
  6. \( /* 4 \text{ and } 5 \text{ give } \checkmark \text{ or } ? */ \text{ return } c \)
• Yield a simpler config, but not necessarily relevant.
Delta Debugging-min Algorithm

• \textit{simplify}(c) = \textit{ddmin}(c,2) defined below

1. \textit{ddmin}(c,N) = \texttt{// N is split granularity}

2. \textbf{if} \ \mid c \mid = 1 \ \textbf{then return} \ c

3. \text{split c into disjoint } \ c_1 \cup \ldots \cup c_N

4. \textbf{if} for some \( k \), \textit{test}(c/c_k) = \times \\
\text{then return } \textit{ddmin} \left( \frac{c}{c_k} , \max(N-1,2) \right)

5. \textbf{else if} \ N < \mid c \mid \ \texttt{// increase granularity} \\
\text{then return } \textit{ddmin}(c, \min(2N, \mid c \mid))

6. \textbf{else return} c
### Example

<table>
<thead>
<tr>
<th>orig</th>
<th>ddmin([1..8], 2)</th>
<th>ddmin([1..8], 4)</th>
<th>ddmin([1,2,3,4,7,8], 3)</th>
<th>ddmin([1,2,7,8], 2)</th>
<th>ddmin([1,2,7,8], 4)</th>
<th>ddmin([2,7,8], 3)</th>
<th>return with c = [2,7,8]</th>
</tr>
</thead>
<tbody>
<tr>
<td>orig</td>
<td>✗</td>
<td>?</td>
<td>✗</td>
<td>?</td>
<td>✗</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>ddmin([1..8], 2)</td>
<td>✗</td>
<td>?</td>
<td></td>
<td>?</td>
<td>✗</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>ddmin([1..8], 4)</td>
<td>✗</td>
<td>?</td>
<td></td>
<td>✗</td>
<td>✗</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>ddmin([1,2,3,4,7,8], 3)</td>
<td>✗</td>
<td>?</td>
<td></td>
<td>✗</td>
<td>✗</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>ddmin([1,2,7,8], 2)</td>
<td>?</td>
<td>?</td>
<td></td>
<td>?</td>
<td>?</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>ddmin([1,2,7,8], 4)</td>
<td>?</td>
<td>?</td>
<td></td>
<td>?</td>
<td>?</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>ddmin([2,7,8], 3)</td>
<td>?</td>
<td>?</td>
<td></td>
<td>?</td>
<td>?</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

- orig
- ddmin([1..8], 2)
- ddmin([1..8], 4)
- ddmin([1,2,3,4,7,8], 3)
- ddmin([1,2,7,8], 2)
- ddmin([1,2,7,8], 4)
- ddmin([2,7,8], 3)

**Return** with c = [2,7,8]
Complexity dadmin

• Worst case, invoking *test* this number of times: \(( |c\_\times|^2 + 7 |c\_\times| ) / 2\)

• If
  – there is only one failure-inducing unit in the initial configuration, and
  – all configurations that include the unit would fail

• then the number of tests is: \(2 \log(|c\_\times|)\).
Localizing

• Let \( c \) be the initial failing configuration. Find subsets \( s \subseteq f \subseteq c \) such that:
  – \( s \) succeeds
  – \( f \) fails
  – their difference \( \Delta = f/s \) is “minimal”. That is, for any \( \delta \in \Delta : \)
    \[
    \text{test}(s \cup \{ \delta \}) \neq \checkmark \\
    \text{test}(f / \{ \delta \}) \neq \times
    \]

• If \( |\Delta| = 1 \), it contains the problem unit. If \( |\Delta| > 1 \), it may contain units whose addition or removal lead to an invalid configuration.
Idea

- Define $\text{test} (\emptyset) = \checkmark$
- Adapt $\text{ddmin}$ to operate on a pair input configurations: $s$ and $f$
- Split $\Delta = f/s$ to $\Delta_1 \cup \ldots \cup \Delta_N$
- Check the outcome of:
  - $\text{test} (s \cup \Delta_k)$
  - $\text{test} (f / \Delta_k)$
- Well, now we have more outcome combinations...
Possible outcomes of tests

If for some partition $\Delta_k$ of $\Delta$ the outcome is one of these:

<table>
<thead>
<tr>
<th>test$(s \cup \Delta_k)$</th>
<th>$f := s \cup \Delta_k$</th>
<th>$s := s \cup \Delta_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>test$(f / \Delta_k)$</td>
<td>$f := f / \Delta_k$</td>
<td>$s := f / \Delta_k$</td>
</tr>
</tbody>
</table>

Else, (so, for any partition $\Delta_k$ of $\Delta$ the outcome is none of the above), increase split granularity.
The dd Algorithm

• \( localize(c) = dd(\emptyset,c,2) \) defined below

1. \( dd(s,f,N) = \)
2. \( \Delta := f/s \)
3. \( \text{if } |\Delta| = 1 \text{ then return } (s,f) \)
4. \( \text{split } \Delta \text{ into } \Delta_1 \cup \ldots \cup \Delta_N \)
5. \( \text{if for some } k, \text{ test}(s \cup \Delta_k) = \times \text{ then return } dd \left( s, s \cup \Delta_k, 2 \right) \)
6. \( \text{if for some } k, \text{ test}(s \cup \Delta_k) = \checkmark \text{ then return } dd \left( s \cup \Delta_k, f, \max(N-1,2) \right) \)
7. \( \text{if for some } k, \text{ test}(f/\Delta_k) = \times \text{ then return } dd \left( s, f/\Delta_k, \max(N-1,2) \right) \)
8. \( \text{if for some } k, \text{ test}(f/\Delta_k) = \checkmark \text{ then return } dd \left( f/\Delta_k, f, 2 \right) \)
9. \( \text{if } N<|\Delta| \quad \text{// increase granularity} \)
   \( \text{then return } dd( f,d, \min(2N, |\Delta|) ) \)
10. \( \text{else return } (s,f) \)