What is Program Mutation?

- Suppose that program P has been tested against a test set T and P has not failed on any test case in T. Now suppose we do the following:

\[
\begin{align*}
P & \quad \text{Changed to} \quad P' \\
\end{align*}
\]

- What behavior do you expect from \( P' \) against tests in T?

- A technique for assessing the goodness of tests
  - Provide criteria for test assessment and enhancement
  - Mutation testing is a code-based test assessment and improvement technique.
  - Can be extended to architecture (e.g., Statecharts) and design (e.g., SDL)

What is Program Mutation? (cont.)

- \( P' \) is known as a mutant of P.

- There might be a test t in T such that \( P(t) \neq P'(t) \).
  - In this case we say that t distinguishes \( P' \) from P. Or, that t has killed \( P' \).

- There might not be any test t in T such that \( P(t) \neq P'(t) \).
  - In this case we say that T is unable to distinguish P and \( P' \). Hence \( P' \) is considered live in the test process.

What is Program Mutation? (cont.)

- If there does not exist any test case t in the input domain of P that distinguishes P from \( P' \) then \( P' \) is said to be equivalent to P.
- If \( P' \) is not equivalent to P but no test in T is able to distinguish it from P then T is considered inadequate.

- A non-equivalent and live mutant offers the tester an opportunity to generate a new test case and hence enhance T.
  - Testing using mutation can often discover subtle flaws in the program under test, when testers try to show that mutants are incorrect solutions.
Example

Original program

```
begin
int x, y;
input (x, y);
if (x < y)
then
output (x+y);
else
output (x/y);
end
```

What is Program Mutation? (cont.)

- **First-order mutant**
  - A mutant generated by introducing only a simple change

- **Higher-order mutant**
  - Generated by introducing two or more simple changes
  - An n-th-order mutant can be created from another (n-1)th-order mutant

Example

- **2nd-order**

```
begin
int x, y;
input (x, y);
if (x < y - 1)
then
output (x/y);
else
output (x+y);
end
```

Mutation and Program Behavior

- **Correct Version**

```
1 enum dangerlevel [none, moderate, high, veryHigh];
2 procedure checkTemp (currentTemp, maxTemp);
3 float currentTemp[2], maxTemp; int highCount=0;
4 enum dangerLevel danger;
5 danger-none;
6 if (currentTemp[2]>maxTemp) highCount+=1;
7 if (highCount>=maxTemp) highCount=highCount+1;
8 if (highTemp[2]>maxTemp) highCount=highCount+1;
9 if (currentTemp[2]>maxTemp) highCount=highCount+1;
10 if (currentTemp[2]>maxTemp) highCount=highCount+1;
11 if (currentTemp[2]>maxTemp) highCount=highCount+1;
12 if (currentTemp[2]>maxTemp) highCount=highCount+1;
13 if (currentTemp[2]>maxTemp) highCount=highCount+1;
14 if (highCount>=3) danger=moderate;
15 if (highCount>=5) danger=high;
16 if (highCount<=2) danger=veryHigh;
17 return(danger);
18 }
```

- **Strong and Weak Mutations**

  - **Strong mutation**
    - Use external observations (Observe return values and side effects after procedure termination)
    - A mutant can behave similar to its parent

  - **Weak mutation**
    - Use internal observations (Observe during executions at different level)
    - Can be performed in various ways

Mutation with drastic behavior change

Mutation with NO behavior change

No difference?

Depending on the point of observation!
Test Adequacy using Mutation

- Given a test set $T$ for program $P$ that must meet requirements $R$, a test adequacy assessment procedure proceeds as follows.
  - **Example**
    \[
    f(x, y) = \begin{cases} 
    x + y & \text{if } x < y, \\
    x * y & \text{otherwise.}
    \end{cases}
    \]
    $T_F = \{ t_1 : < x = 0, y = 0 >, \\
    t_2 : < x = 0, y = 1 >, \\
    t_3 : < x = 1, y = 0 >, \\
    t_4 : < x = -1, y = -2 > \}.$
  - **Step 1:** Create a set $M$ of mutants of $P$.
    - Let $M = \{M_0, M_1, \ldots, M_k\}$. ($k$ mutants)

Test Adequacy using Mutation (cont.)

- **Example:**
  - Step 1: Create a set $M$ of mutants of $P$.
  - Let $M = \{M_0, M_1, \ldots, M_8\}$. ($k$ mutants)
  - Step 2: Identify mutants killed by $1^+$ test cases
    - For each mutant $M_i$, **find if there exists a $t \in T$ such that $M_i(t) \neq P(t)$**. If such a $t$ exists then $M_i$ is considered **killed** and removed from further consideration.
Test Adequacy using Mutation (cont.)

- **Step 3:**
  - Suppose that \( k_1 \leq k \) mutants have been killed – i.e., \((k-k_1)\) mutants are live.
  - **Case 1:** \((k-k_1)=0\): T is adequate with respect to mutation.
  - **Case 2:** \((k-k_1)>0\) then we compute the mutation score (MS) as follows:

\[
MS = \frac{k_1}{(k-e)}
\]

- \( e \) is the number of equivalent mutants.
- Note: \( e \leq (k-k_1) \).

Example: Equivalent mutants

Is \( M_2 \) equivalent to \( P \)?

Mutated

\[
f_M(x, y) = \begin{cases} 
  x + y & \text{if } x < y, \\
  x & \text{otherwise.}
\end{cases}
\]

\[
g_M(x, y) = \begin{cases} 
  x + y & \text{if } x < y + 1, \\
  x & \text{otherwise.}
\end{cases}
\]

What test do we need?

- \( \eta : x = 0, y = 0 \) Satisfies \( C_2 \) but not \( C_1 \)

- New test \( \rho : x = 1, y = 1 \) Satisfies \( C_1 \) and \( C_2 \)

Conditions for the test case

Test Enhancement using Mutation

- Enhance a test set \( T \) after having assessed its adequacy with respect to mutation.
  - **Step 1** (MS=1): Use other technique, or a different set of mutants to help enhance \( T \).
  - **Step 2** (MS<1): Repeat creating new test case(s) that distinguish 1* live mutants until all live mutants are distinguished.
Error Detection using Mutation

- As with any test enhancement technique, there is no guarantee that tests derived to distinguish live mutants will reveal a yet undiscovered error in P. Nevertheless, empirical studies have found to be the most powerful of all formal test enhancement techniques.

Example
- Consider the following function `foo` that is required to return the sum of two integers `x` and `y`. Clearly `foo` is incorrect.

```c
int foo(int x, y) {
    return (x-y);  // This should be return (x+y)
}
```

Error Detection using Mutation (cont.)

- Let us evaluate the adequacy of `T` using mutation.
  - Suppose that the following three mutants are generated from `foo`.
    ```c
    M1: int foo(int x, y) {
        return (x+y);
    }
    M2: int foo(int x, y) {
        return (x-0);
    }
    M3: int foo(int x, y) {
        return (0+y);  // Note that M1 is obtained by replacing the - operator by a + operator, M2 by replacing y by 0, and M3 by replacing x by 0.
    }
    ```
  - Now suppose that `foo` has been tested using a test set `T` that contains two tests:
    ```
    T={ t1: <x=1, y=0>, t2: <x=-1, y=0> }
    ```
  - First note that `foo` behaves perfectly fine on each test in, i.e. `foo` returns the expected value for each test case in `T`.
  - Also, `T` is adequate with respect to all control and data flow based test adequacy criteria.
Error Detection using Mutation (cont.)

- Next, we execute each mutant against tests in T until the mutant is distinguished or we have exhausted all tests. Here is what we get.

\[ T = \{ t_1: <x=1, y=0>, t_2: <x=-1, y=0> \} \]

<table>
<thead>
<tr>
<th>Test</th>
<th>foo(t)</th>
<th>M1(t)</th>
<th>M2(t)</th>
<th>M3(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>t2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

|   | Live | Live | Killed |

| M1: int foo(int x, y){ return [x+y]; } |
| M2: int foo(int x, y){ return [x-0]; } |
| M3: int foo(int x, y){ return [0+y]; } |

After executing all three mutants we find that two are live and one is distinguished.

Computation of mutation score requires us to determine if any of the live mutants is equivalent to the program.
- Consider entire input domain for equivalence check

\[ MS = k_y/(k-e) \]?

<table>
<thead>
<tr>
<th>Test</th>
<th>foo(t)</th>
<th>M1(t)</th>
<th>M2(t)</th>
<th>M3(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>t2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

|   | Live | Live | Killed |

Let us examine the following two live mutants.

- Let us focus on M1. A test that distinguishes M1 from foo must satisfy the following condition: \( x-y \neq x+y \) implies \( y \neq 0 \).
- Hence, we get a test case \( t_3: <x=1, y=1> \).
- Executing foo on \( t_3 \) gives us \( foo(t_3) = 0 \). However, according to the requirements we must get \( foo(t_3) = 2 \). Thus, \( t_3 \) distinguishes M1 from foo and also reveals the error.

Guaranteed Error Detection

- Sometimes, there exists a mutant \( P' \) of program P such that any test \( t \) that distinguishes \( P' \) from P also causes P to fail. More formally:
  - Let \( P' \) be a mutant of P and \( t \) a test in the input domain of P.
  - We say that \( P' \) is an error revealing mutant if the following condition holds for any \( t \) such that \( P'(t) \neq P(t) \).
    - \( P(t) \neq R(t) \), where \( R(t) \) is the expected response of P based on its requirements.

Is M1 in the previous example an error revealing mutant?

What about M2?
Distinguishing a Mutant

- A test case $t$ that distinguishes a mutant $m$ from its parent program $P$ must satisfy the following three conditions:
  - Condition 1: **Reachability**: $t$ must cause $m$ to follow a path that arrives at the mutated statement in $m$.
  - Condition 2: **Infection**: If $S_{in}$ is the state of the mutant upon arrival at the mutant statement and $S_{out}$ the state soon after the execution of the mutated statement, then $S_{in} \neq S_{out}$.
  - Condition 3: **Propagation**: If difference between $S_{in}$ and $S_{out}$ must propagate to the output of $m$ such that the output of $m$ is different from that of $P$.

Equivalent Mutants

- The problem of deciding whether or not a mutant is equivalent to its parent program is undecidable. Hence, there is no way to fully automate the detection of equivalent mutants.
- The number of equivalent mutants can vary from one program to another. However, empirical studies have shown that one can expect about 5% of the generated mutants to the equivalent to the parent program.
- Identifying equivalent mutants is generally a manual and often time consuming--as well as frustrating--process.

A Misconception

- **Misconception**: Any "coverage" based technique, including mutation, will not be able to detect errors due to missing path.

- Consider the following programs.

  Program under test
  ```c
  int foo (int x, y){
  int p=0;
  if(x<y)
    p=p+1;
  return(x+p*y)   ----- Missing else
  }
  ```

  Correct program
  ```c
  int foo (int x, y){
  int p=0;
  if(x<y)
    p=p+1;
  else
    p=p-1;
  return(x+p*y)
  }
  ```

A Misconception (cont.)

- Suggest at least one error-revealing mutant $M$ of $foo$ that is guaranteed to reveal the error.

  - Suppose $T$ is decision adequate for $foo$. Is $T$ guaranteed to reveal the error?

  - Suppose $T$ is def-use adequate for $foo$. Is $T$ guaranteed to reveal the error?