جهش و آزمون نرم‌افزاری

محمد تنهایی

Applying Syntax-based Testing to Programs

Syntax-based criteria **originated** with programs and have been used most with programs

BNF criteria are most commonly used to test compilers

Mutation testing criteria are most commonly used for unit testing and integration testing of classes

BNF Testing for Compilers (۵.۲.۱)

آزمون کامپایلرها دشوار است

میلیون‌ها خط کد درست

کامپایلرها باید برنامه‌های غلط را شناسایی و رد کنند

می‌توان از معیار BNF برای بررسی همه ویژگی‌های زبان مورد نظر کامپایلر استفاده کرد.

این کار خیلی دشوار و پیچیده است.

Program-based Grammars (۵.۲.۲)

مهمترین استفاده از روش آزمون مبتنی بر syntax ، **دستکاری برنامه‌ها** است.

هر عملگر برخی از اجزای رشته ورودی برنامه را **دستکاری** می‌کند و یک جهش می‌سازد.

جهش باید به درستی **کامپایل** شود. یعنی برنامه باید **نحو درستی** داشته باشد.

جهش آزمون نیست، بلکه ابزاری برای یافتن آزمون است.

زمانی که جهش‌ها ایجاد شدند، باید برخی آزمونه‌ها را یافت که این جهش‌ها را **بکشد**.

به این کار اصلاحا **killing mutants** گفته می‌شود.

Killing Mutants

Given a mutant $m \in M$ for a ground string program P and a test t, t is said to kill m if and only if the output of t on P is different from the output of t on m.

اگر عملگرهای جهش درست طراحی شده باشند، خروجی آزمون باید خیلی قوی باشد.

برای هر نوع زبان، نوع عملگرهای جهش متفاوت است.

وظیفه آزمون‌گر، افزودن آزمون تا زمانی است که همه جهش‌ها کشته شوند

- **جهش مرده**: یک آزمون جهش را کشته است.
- **جهش سقط شده!**: کلا از اول جهش مشکل داشته (کامپایل نمی شود)
- **جهش سر راست**: هر نوع آزمونی جهش را می‌کشد.
- **جهش برابر**: هیچ آزمونی نمی‌تواند جهش را بکشد.

Program-based Grammars

Original Method

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
             minVal = B;
         }
         return (minVal);
} // end Min
```

Program-based Grammars

Original Method

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
             minVal = B;
         }
         return (minVal);
} // end Min
```

With Embedded Mutants

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
Δ 1     minVal = B;
        if (B < A)
Δ 2     if (B > A)
Δ 3     if (B < minVal)
        {
                minVal = B;
Δ 4             Bomb ();
Δ 5             minVal = A;
Δ 6             minVal = failOnZero (B);
        }
        return (minVal);
} // end Min
```

Program-based Grammars

Original Method

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
             minVal = B;
         }
         return (minVal);
} // end Min
```

With Embedded Mutants

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
Δ 1    minVal = B;
        if (B < A)
Δ 2    if (B > A)
Δ 3    if (B < minVal)
        {
                minVal = B;
Δ 4            Bomb ();
Δ 5            minVal = A;
Δ 6            minVal = failOnZero (B);
        }
        return (minVal);
} // end Min
```

6 mutants

Each represents a separate program

Program-based Grammars

Original Method

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
             minVal = B;
         }
         return (minVal);
} // end Min
```

6 mutants

Each represents a separate program

With Embedded Mutants

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
Δ 1   minVal = B;
        if (B < A)
Δ 2   if (B > A)
Δ 3   if (B < minVal)
        {
                minVal = B;
Δ 4           Bomb ();
Δ 5           minVal = A;
Δ 6           minVal = failOnZero (B);
        }
        return (minVal);
} // end Min
```

Replace one variable with another

Program-based Grammars

Original Method

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
            minVal = B;
         }
         return (minVal);
} // end Min
```

With Embedded Mutants

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
Δ 1  minVal = B;
        if (B < A)
Δ 2  if (B > A)
Δ 3  if (B < minVal)
        {
                minVal = B;
Δ 4          Bomb ();
Δ 5          minVal = A;
Δ 6          minVal = failOnZero (B);
        }
        return (minVal);
} // end Min
```

Replace one variable with another

Changes operator

6 mutants

Each represents a separate program

Program-based Grammars

Original Method

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
        if (B < A)
        {
            minVal = B;
         }
         return (minVal);
} // end Min
```

With Embedded Mutants

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
Δ 1     minVal = B;
        if (B < A)
Δ 2     if (B > A)
Δ 3     if (B < minVal)
        {
                minVal = B;
Δ 4             Bomb ();
Δ 5             minVal = A;
Δ 6             minVal = failOnZero (B);
        }
        return (minVal);
} // end Min
```

Replace one variable with another

Changes operator

Immediate runtime failure … if reached

6 mutants

Each represents a separate program

Program-based Grammars

Original Method

```
int Min (int A, int B)
{
     int minVal;
     minVal = A;
     if (B < A)
     {
          minVal = B;
     }
     return (minVal);
} // end Min
```

With Embedded Mutants

```
int Min (int A, int B)
{
        int minVal;
        minVal = A;
Δ 1   minVal = B;
        if (B < A)
Δ 2   if (B > A)
Δ 3   if (B < minVal)
        {
              minVal = B;
Δ 4         Bomb ();
Δ 5         minVal = A;
Δ 6         minVal = failOnZero (B);
        }
        return (minVal);
} // end Min
```

Replace one variable with another

Changes operator

Immediate runtime failure … if reached

Immediate runtime failure if B==0 else does nothing

6 mutants

Each represents a separate program

Syntax-Based Coverage Criteria

The RIP model from chapter 1:

Reachability : The test causes the faulty statement to be reached (in mutation – the mutated statement)

Infection : The test causes the faulty statement to result in an incorrect state

Propagation : The incorrect state propagates to incorrect output

The RIP model leads to two variants of mutation coverage ...

Syntax-Based Coverage Criteria

Mutation Coverage (MC) : For each $m \in M$, TR contains exactly one requirement, to kill m.

The RIP model from chapter 1:

Reachability : The test causes the faulty statement to be reached (in mutation – the mutated statement)

Infection : The test causes the faulty statement to result in an incorrect state

Propagation : The incorrect state propagates to incorrect output

The RIP model leads to two variants of mutation coverage ...

Syntax-Based Coverage Criteria

1) **Strongly Killing Mutants:**

Given a mutant $m \in M$ for a program P and a test t, t is said to strongly kill m if and only if the output of t on P is different from the output of t on m

2) **Weakly Killing Mutants:**

Given a mutant $m \in M$ that modifies a location l in a program P, and a test t, t is said to weakly kill m if and only if the state of the execution of P on t is different from the state of the execution of m immediately on t after l

Weakly killing satisfies reachability and infection, but not propagation

Weak Mutation

"Weak mutation" is so named because it is **easier** to kill mutants under this assumption

Weak mutation also requires **less analysis**

Some mutants can be killed under weak mutation but not under strong mutation (**no propagation**)

In practice, there is little difference

Weak Mutation

Weak Mutation Coverage (WMC) : For each $m \in M$, TR contains exactly one requirement, to weakly kill m.

"Weak mutation" is so named because it is **easier** to kill mutants under this assumption

Weak mutation also requires **less analysis**

Some mutants can be killed under weak mutation but not under strong mutation (**no propagation**)

In practice, there is little difference

Mutation Example

Mutant 1 in the Min( ) example is:

- The complete test specification to kill mutant 1:

- Reachability : *true* // Always get to that statement
- Infection : $A \neq B$
- Propagation: $(B < A) = false$ // Skip the next assignment
- Full Test Specification : $true \wedge (A \neq B) \wedge ((B < A) = false)$

$$\equiv (A \neq B) \wedge (B \geq A)$$

$$\equiv (B > A)$$

- (A = 5, B = 7) will kill mutant 1.

Mutation Example

Mutant 1 in the Min( ) example is:

```
    minVal = A;
Δ 1 minVal = B;
    if (B < A)
    minVal = B;
```

- The complete test specification to kill mutant 1:

- Reachability : *true* // Always get to that statement
- Infection : $A \neq B$
- Propagation: $(B < A) = false$ // Skip the next assignment
- Full Test Specification : $true \wedge (A \neq B) \wedge ((B < A) = false)$

$$\equiv (A \neq B) \wedge (B \geq A)$$

$$\equiv (B > A)$$

- (A = 5, B = 7) will kill mutant 1.

Equivalent Mutation Example

Mutant 3 in the Min() example is equivalent:

- The infection condition is “(B < A) != (B < minVal)”
- However, the previous statement was “minVal = A”
  - Substituing, we get: “(B < A) != (B < A)”
- Thus no input can kill this mutant

Equivalent Mutation Example

Mutant 3 in the Min() example is equivalent:

minVal = A;
if (B < A)
Δ 3 if (B < minVal)

- The infection condition is "(B < A) != (B < minVal)"
- However, the previous statement was "minVal = A"
  - Substituing, we get: "(B < A) != (B < A)"
- Thus no input can kill this mutant

Strong Versus Weak Mutation

```
1    boolean isEven (int X)
2    {
3        if (X < 0)
4            X = 0 - X;
Δ 4          X = 0;
5        if (float) (X/2) == ((float) X) / 2.0
6            return (true);
7        else
8            return (false);
9        }
```

Strong Versus Weak Mutation

```
1    boolean isEven (int X)
2    {
3        if (X < 0)
4            X = 0 - X;
Δ 4          X = 0;
5        if (float) (X/2) == ((float) X) / 2.0
6            return (true);
7        else
8            return (false);
9        }
```

Reachability : $X < 0$

Strong Versus Weak Mutation

```
1     boolean isEven (int X)
2     {
3         if (X < 0)
4             X = 0 - X;
Δ 4           X = 0;
5         if (float) (X/2) == ((float) X) / 2.0
6             return (true);
7         else
8             return (false);
9         }
```

Reachability : X < 0

Infection : X != 0

Strong Versus Weak Mutation

```
1     boolean isEven (int X)
2     {
3         if (X < 0)
4             X = 0 - X;
Δ 4           X = 0;
5         if (float) (X/2) == ((float) X) / 2.0
6             return (true);
7         else
8             return (false);
9         }
```

Reachability : X < 0

Infection : X != 0

(X = -6) will kill mutant 4
under weak mutation

Strong Versus Weak Mutation

```
1     boolean isEven (int X)
2     {
3         if (X < 0)
4             X = 0 - X;
Δ 4           X = 0;
5         if (float) (X/2) == ((float) X) / 2.0
6             return (true);
7         else
8             return (false);
9         }
```

Reachability : X < 0

Infection : X != 0

(X = -6) will kill mutant 4 under weak mutation

Propagation :

((float) ((0-X)/2) == ((float) 0-X) / 2.0)

!= ((float) (0/2) == ((float) 0) / 2.0)

That is, X is not even …

Thus (X = -6) does *not* kill the mutant under strong mutation

پایان