Inferring Mutant Utility from Program Context

René Just*, Bob Kurtz†, Paul Ammann†

*University of Massachusetts, Amherst
†George Mason University

July 12, 2017
Automatic program repair
Automatic program repair
Automatic program repair

Goal: generate **mutants** that **improve** the functional correctness of the original program.
Mutation-based test generation
Mutation-based test generation
Mutation-based test generation

Goal: generate strong tests using hard-to-detect mutants.
Selecting a set of effective mutants

Goals:
1. Generate mutants that **improve functional correctness**.
2. Generate mutants that are **hard to detect**.
Selecting a set of effective mutants

Goals:
1. Generate mutants that improve functional correctness.
2. Generate mutants that are hard to detect.

Problem:
- Many mutants are non compilable, trivially crashing, or equivalent → useless and costly mutants.

Existing strategies:
- Selective mutation (e.g., pattern-based mutation).
- Program-independent and no better than random.

Gopinath et al., ICSE’16, Kurtz et al., FSE’16
Selecting a set of effective mutants

Goals:
1. Generate mutants that improve functional correctness.
2. Generate mutants that are hard to detect.

Problem:
• Many mutants are non compilable, trivially crashing, or equivalent → useless and costly mutants.

Existing strategies:
• Selective mutation (e.g., pattern-based mutation).
• Program-independent and no better than random.

Hypothesis: Program context matters!

Gopinath et al., ICSE’16, Kurtz et al., FSE’16
public double getAbsAvg(double[] nums) {
    double sum = 0;
    for (int i = 0; i < nums.length; ++i) {
        if (nums[i] < 0) {
            sum -= nums[i];
        } else {
            sum += nums[i];
        }
    }
    return sum / nums.length;
}
public double getAbsAvg(double[] nums) {
    double sum = 0;
    for (int i = 0; i < nums.length; ++i) {
        if (nums[i] < 0) {
            sum -= nums[i];
        } else {
            sum += nums[i];
        }
    }
    return sum / nums.length;
}
Program context: Parent context

Original program

```java
public double getAbsAvg(double[] nums) {
    double sum = 0;

    for (int i = 0; i < nums.length; ++i) {
        if (nums[i] < 0) {
            sum -= nums[i];
        } else {
            sum += nums[i];
        }
    }

    return sum / nums.length;
}
```

Mutation operator

\[lhs < rhs \quad \rightarrow \quad lhs \neq rhs\]
public double getAbsAvg(double[] nums) {
    double sum = 0;
    for (int i = 0; i < nums.length; ++i) {
        if (nums[i] < 0) {
            sum -= nums[i];
        } else {
            sum += nums[i];
        }
    }
    return sum / nums.length;
}
public double getAbsAvg(double[] nums) {
    double sum = 0;
    for (int i = 0; i < nums.length; ++i) {
        if (nums[i] < 0) {
            sum -= nums[i];
        } else {
            sum += nums[i];
        }
    }
    return sum / nums.length;
}
public double getAbsAvg(double[] nums) {
    double sum = 0;
    for (int i = 0; i < nums.length; ++i) {
        if (nums[i] < 0) {
            sum -= nums[i];
        } else {
            sum += nums[i];
        }
    }
    return sum / nums.length;
}
public double getAbsAvg(double[] nums) {
    double sum = 0;
    for (int i = 0; i < nums.length; ++i) {
        if (nums[i] < 0) {
            sum -= nums[i];
        } else {
            sum += nums[i];
        }
    }
    return sum / nums.length;
}
Program context: Children context

Original program

```java
public double getAbsAvg(double[] nums) {
    double sum = 0;
    for (int i = 0; i < nums.length; ++i) {
        if (nums[i] < 0) {
            sum -= nums[i];
        } else {
            sum += nums[i];
        }
    }
    return sum / nums.length;
}
```

Mutation operator

lhs < rhs $\rightarrow$ lhs $\leq$ rhs

Context: kind of operands (identifier vs. operator vs. literal)
Program context: Data type context

Original program

```java
public double getAbsAvg(double[] nums) {
    double sum = 0;

    for (int i = 0; i < nums.length; ++i) {
        if (nums[i] < 0) {
            sum -= nums[i];
        } else {
            sum += nums[i];
        }
    }

    return sum / nums.length;
}
```

Mutation operator

0 \rightarrow -1
public double getAbsAvg(double[] nums) {
    double sum = 0;
    for (int i = 0; i < nums.length; ++i) {
        if (nums[i] < 0) {
            sum -= nums[i];
        } else {
            sum += nums[i];
        }
    }
    return sum / nums.length;
}
Program context: Data type context

Original program

```java
public double getAbsAvg(double[] nums) {
    double sum = 0;
    for (int i = 0; i < nums.length; ++i) {
        if (nums[i] < 0) {
            sum -= nums[i];
        } else {
            sum += nums[i];
        }
    }
    return sum / nums.length;
}
```

Mutation operator

Context: data type (double vs. int)
Program context: Summary

Mutation operator effectiveness differs, even within a single method.

Different dimensions of program context
- **Parent context**: Kind of lexically enclosing statement(s).
- **Data type context**: Data types of operators and operands.
- **Children context**: Kind of operands.
Modeling program context using the AST
Modeling program context using the AST

- Parent context
Modeling program context using the AST

- Parent context
- Data type context

```
boolean(int,int)
```
Modeling program context using the AST

- Parent context
- Data type context
- Children context
Mutant utility

1. **Equivalence**: equivalent mutants have low utility.
2. **Triviality**: trivially crashing mutants have low utility.
3. **Dominance**: dominator mutants have high utility.
# Mutant utility

1. **Equivalence**: equivalent mutants have **low utility**.
2. **Triviality**: trivially crashing mutants have **low utility**.
3. **Dominance**: **dominator mutants** have **high utility**.

<table>
<thead>
<tr>
<th>Mutant</th>
<th>Test</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$t_1$</td>
<td>$t_2$</td>
<td>$t_3$</td>
<td>$t_4$</td>
</tr>
<tr>
<td>$m_1$</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$m_2$</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$m_3$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$m_4$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$m_5$</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$m_6$</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$m_7$</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>$m_8$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$m_9$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{10}$</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>$m_{11}$</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{12}$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{13}$</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$m_{14}$</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Diagram:

- $m_4, m_7, m_9, m_{14}$
- $m_8, m_{13}$
- $m_2, m_5$
- $m_{11}, m_{12}$
- $m_3, m_6$
Mutant utility

1. **Equivalence**: equivalent mutants have low utility.
2. **Triviality**: trivially crashing mutants have low utility.
3. **Dominance**: dominator mutants have high utility.

**Good**: dominator mutants  
**Bad**: trivial mutants  
**Ugly**: equivalent mutants
Is program context predictive of mutant utility?

Determining ground truth (equivalence, triviality, dominance)
- Approximations using extensive test suites.
- 95+% statement coverage.

Selected subjects: 97 unique classes (4 real-world projects)
- 15,000 test cases
  - 64 test cases cover each mutant, on average
  - 23 test cases detect each mutant, on average

80,000 generated mutants (129 mutation operators)

http://defects4j.org
Recall the high-level goal

**Good**: dominator mutants

**Bad**: trivial mutants

**Ugly**: equivalent mutants
Expected mutant utility: context-based vs. random

Context-based selection
Expected mutant utility: context-based vs. random

Context-based selection

Random selection

Full experimental details in the paper.
Future work: what’s next?

More complex program context models
- Scope and visibility
- Control and data flow

Train effective machine learning classifiers

Integrate into downstream techniques
Inferring mutant utility from program context

Program context: Parent context

Original program
public double getAbsAvg(double[] nums) {
    double sum = 0;
    for (int i = 0; i < nums.length; ++i) {
        if (nums[i] < 0) {
            sum -= nums[i];
        } else {
            sum += nums[i];
        }
    }
    return sum / nums.length;
}

Mutation operator
\[ \text{lhs} \leq \text{rhs} \]  \[ \text{lhs} = \text{rhs} \]

equivalent  non-equivalent

Context: kind of lexically enclosing statement (for vs. if)

Modeling program context using the AST
- Parent context
- Data type context
- Children context

Expected mutant utility: context-based vs. random

Context-based selection
Random selection

Future work: what’s next?
More complex program context models
- Scope and visibility
- Control and data flow
Train effective machine learning classifiers
Integrate into downstream techniques

Automatic program repair