Advances in Mutation Testing Research for C++

Pedro Delgado-Pérez

TAROT: Intro Talk

June 2015

Index

Index

Mutation testing

A brief definition

- A fault injection testing technique. $x > 1 \rightarrow x < 1$
- Involves inserting simple syntactic changes in the program using mutation operators.
- Mutation operators are based on typical mistakes.
- This modification creates a new version called mutant.

Goals

1. Measure how good is a test suite detecting faults affecting the program.
2. Improve the test suite through the results of the mutants.

Mutation testing

A brief definition

- A fault injection testing technique. $x > 1 \rightarrow x < 1$
- Involves inserting simple syntactic changes in the program using mutation operators.
- Mutation operators are based on typical mistakes.
- This modification creates a new version called mutant.

Goals

1. Measure how good is a test suite detecting faults affecting the program.
2. Improve the test suite through the results of the mutants.

Mutation testing

Mutant classification

1. **Dead:** The output of the original program and the mutant is different.
2. **Alive:** The change has not been detected:
    - Equivalence: The change cannot be detected by any input.
    - A new test case is needed to detect the change.
3. **Invalid:** The mutant does not comply with the grammar rules.

Mutation testing

Mutant classification

1. **Dead:** The output of the original program and the mutant is different.
2. **Alive:** The change has not been detected:
   - Equivalence: The change cannot be detected by any input.
   - A new test case is needed to detect the change.
3. **Invalid:** The mutant does not comply with the grammar rules.

Index

Mutation testing research

History

- First ideas in 1970s.
- Early years: around procedural languages $\rightarrow$ traditional operators
- From 1990s onwards: around other kind of languages and domains.

Mutation tools developed

- *Mothra* - FORTRAN
- *MuJava* - Java
- *GAmera* - WS-BPEL
- ...

Mutation testing research

Motivation

- One of the most important programming languages → 4*th* position in TIOBE index.
- Research regarding C++ was pending.
- Obtaining results about the usefulness of this technique in C++.

Possible reasons

- Complexity of the language.
- The technique to inject mutations in the code.
- Dependency analysis of source code files.

Mutation testing research

Motivation

- One of the most important programming languages $\rightarrow$ 4*th* position in TIOBE index.
- Research regarding C++ was pending.
- Obtaining results about the usefulness of this technique in C++.

Possible reasons

- Complexity of the language.
- The technique to inject mutations in the code.
- Dependency analysis of source code files.

The C++ programming language

Achievements

1. State of the art [1].
2. Definition of a set of class-level mutation operators [2].
3. Implementation of class mutation operators.

P. Delgado-Pérez, I. Medina-Bulo and J. J. Domínguez-Jiménez.
Analysis of the development process of a mutation testing tool for the C++ language.
In *The Ninth International Multi-Conference on Computing in the Global Information Technology, ICCGI 2014*. Seville, Spain, 2014.

P. Delgado-Pérez, I. Medina-Bulo, J. Domínguez-Jiménez, A. García-Domínguez and F. Palomo-Lozano.
Class mutation operators for C++ object-oriented systems.
Annals of Telecommunications, April 2015.
ISSN 0003-4347.

Index

Categories

Summary

- Definition of 37 operators at the class level.
- Operators grouped into 7 categories.
- Adapted and new operators.

1. Access control
2. Inheritance
3. Polymorphism and dynamic binding
4. Method overloading
5. Exception handling
6. Object and member replacement
7. Miscellany

Categories

Summary

- Definition of 37 operators at the class level.
- Operators grouped into 7 categories.
- Adapted and new operators.

1. Access control
2. Inheritance
3. Polymorphism and dynamic binding
4. Method overloading
5. Exception handling
6. Object and member replacement
7. Miscellany

Example

Example "Inheritance" block: IOD (Overriding method deletion)

Original:

```
class A {                          class B: public A{
    ... ...                            ... ...
    int method(){... ...};             int method(){... ...};
};                                  };
```

Mutant:

```
class A {                          class B: public A{
    ... ...                            ... ...
    int method(){... ...};             /*IOD*/
};                                 };
```

Example

Example “Inheritance” block: IOD (Overriding method deletion)

```
Original:
    class A {                         class B: public A{
        ... ...                           ... ...
        int method(){... ...};            int method(){... ...};
    };                                 };

Mutant:
    class A {                         class B: public A{
        ... ...                           ... ...
        int method(){... ...};            /*IOD*/
    };                                };
```

Example

Example “Inheritance” block: IOD (Overriding method deletion)

Original:

```
class A {                          class B: public A{
    ... ...                            ... ...
    int method(){... ...};             int method(){... ...};
};                                  };
```

Mutant:

```
class A {                          class B: public A{
    ... ...                            ... ...
    int method(){... ...};             /*IOD*/
};                                 };
```

Example

Example "Polymorphism" block: PVI (*virtual* modifier insertion)

Original:

```
class A {                          class B: public A{
    ... ...                            ... ...
    int method(){... ...};             int method(){... ...};
};                                  };
```

Mutant:

```
class A {
    ... ...
    virtual int method(){... ...};
};
```

Example

Example “Polymorphism” block: PVI (*virtual* modifier insertion)

Original:

```
class A {                        class B: public A{
    ... ...                          ... ...
    int method(){... ...};           int method(){... ...};
};                               };
```

Mutant:

```
class A {
    ... ...
    virtual int method(){... ...};
};
```

Index

Mutation operator implementation

AST transformations

- **Abstract Syntax Tree (AST):** Simplified structure of the code.
- Language elements are represented with different kind of nodes.
- Traversal of the AST through pattern matching.
- Useful to determine the mutation locations and transform the code.

Mutation operator implementation

Steps

1. Creation of the pattern using the DSL in Clang.
2. The source code is converted to the form of AST.
3. AST is traversed searching for every mutation target.
4. The nodes retrieved are analyzed, ensuring that the injection of a fault is possible at that point.
5. Depending on the nature of the operator, one or more variants can be introduced in each location.
6. The mutation is inserted.
7. The source code containing the mutant is saved.

Mutation operator implementation

Steps

1. Creation of the pattern using the DSL in Clang.
2. The source code is converted to the form of AST.
3. AST is traversed searching for every mutation target.
4. The nodes retrieved are analyzed, ensuring that the injection of a fault is possible at that point.
5. Depending on the nature of the operator, one or more variants can be introduced in each location.
6. The mutation is inserted.
7. The source code containing the mutant is saved.

Mutation operator implementation

Steps

1. Creation of the pattern using the DSL in Clang.
2. The source code is converted to the form of AST.
3. AST is traversed searching for every mutation target.
4. The nodes retrieved are analyzed, ensuring that the injection of a fault is possible at that point.
5. Depending on the nature of the operator, one or more variants can be introduced in each location.
6. The mutation is inserted.
7. The source code containing the mutant is saved.

Mutation operator implementation

Steps

1. Creation of the pattern using the DSL in Clang.
2. The source code is converted to the form of AST.
3. AST is traversed searching for every mutation target.
4. The nodes retrieved are analyzed, ensuring that the injection of a fault is possible at that point.
5. Depending on the nature of the operator, one or more variants can be introduced in each location.
6. The mutation is inserted.
7. The source code containing the mutant is saved.

Mutation operator implementation

Steps

1. Creation of the pattern using the DSL in Clang.
2. The source code is converted to the form of AST.
3. AST is traversed searching for every mutation target.
4. The nodes retrieved are analyzed, ensuring that the injection of a fault is possible at that point.
5. Depending on the nature of the operator, one or more variants can be introduced in each location.
6. The mutation is inserted.
7. The source code containing the mutant is saved.

Mutation operator implementation

Steps

1. Creation of the pattern using the DSL in Clang.
2. The source code is converted to the form of AST.
3. AST is traversed searching for every mutation target.
4. The nodes retrieved are analyzed, ensuring that the injection of a fault is possible at that point.
5. Depending on the nature of the operator, one or more variants can be introduced in each location.
6. The mutation is inserted.
7. The source code containing the mutant is saved.

Mutation operator implementation

Steps

1. Creation of the pattern using the DSL in Clang.
2. The source code is converted to the form of AST.
3. AST is traversed searching for every mutation target.
4. The nodes retrieved are analyzed, ensuring that the injection of a fault is possible at that point.
5. Depending on the nature of the operator, one or more variants can be introduced in each location.
6. The mutation is inserted.
7. The source code containing the mutant is saved.

Mutation operator implementation

Steps

1. Creation of the pattern using the DSL in Clang.
2. The source code is converted to the form of AST.
3. AST is traversed searching for every mutation target.
4. The nodes retrieved are analyzed, ensuring that the injection of a fault is possible at that point.
5. Depending on the nature of the operator, one or more variants can be introduced in each location.
6. The mutation is inserted.
7. The source code containing the mutant is saved.

Index

Can we insert the mutation?

1. Check appropriate conditions for the mutation.
2. Reduce unproductive mutations.

Example “Inheritance” block: IOD (Overriding method deletion)

Original:

```
class A {                          class B: public A{
    ... ...                            ... ...
    int method(){... ...};             int method(){... ...}
};                                 };
```

Goals

- Prevent errors in the syntax of the code.
- Avoid *noise* and *silence* in the pattern matching.

Can we insert the mutation?

1. Check appropriate conditions for the mutation.
2. Reduce unproductive mutations.

Example “Inheritance” block: IOD (Overriding method deletion)

Original:

```
class A {                          class B: public A{
    ... ...                            ... ...
    int method(){... ...};             int method(){... ...}
};                                 };
```

Goals

- Prevent errors in the syntax of the code.
- Avoid *noise* and *silence* in the pattern matching.

Can we insert the mutation?

1. Check appropriate conditions for the mutation.
2. Reduce unproductive mutations.

Example “Inheritance” block: IOD (Overriding method deletion)

Original:

```
class A {                              class B: public A{
    ... ...                                ... ...
    int method(){... ...};                 int method(){... ...}
};                                     };
```

Goals

- Prevent errors in the syntax of the code.
- Avoid *noise* and *silence* in the pattern matching.

Is the mutation valuable?

1. Check appropriate conditions for the mutation.
2. Reduce unproductive mutations.

Unproductive mutants

- Those mutants which do not help the purpose of mutation testing.
- Mutants:
  1. Invalid mutants.
  2. Equivalent mutants
  3. Trivial mutants.
- Detect situations always producing an unproductive mutant.

Is the mutation valuable?

1. Check appropriate conditions for the mutation.
2. Reduce unproductive mutations.

Unproductive mutants

- Those mutants which do not help the purpose of mutation testing.
- Mutants:
  1. Invalid mutants.
  2. Equivalent mutants
  3. Trivial mutants.
- Detect situations always producing an unproductive mutant.

Is the mutation valuable?

1. Check appropriate conditions for the mutation.
2. Reduce unproductive mutations.

```
Original:
     class A {                        class B: public A{
       ... ...                          ... ...
       virtual int method() = 0;          int method(){... ...};
     };                               };
Mutant:
     class A {                        class B: public A{
       ... ...                          ... ...
       virtual int method() = 0;       /*IOD*/
     };                               };
```

Is the mutation valuable?

1. Check appropriate conditions for the mutation.
2. Reduce unproductive mutations.

Original:

```
class A {                          class B: public A{
  ... ...                            ... ...
  virtual int method() = 0;           int method(){... ...};
};                                 };
```

Mutant:

```
class A {                          class B: public A{
  ... ...                            ... ...
  virtual int method() = 0;         /*IOD*/
};                                 };
```

Index

Conclusion

Summary

- Goal: apply mutation testing to C++.
- First step: definition of 37 class mutation operators.
- Second step: automation of mutation operators.
- Third step: evaluation and improvement of operators.

Future work

- Measure quality metrics.
- Improve the mutation tool:
  - Test coverage.
  - New standards of the language.
- Evolutionary Mutation Testing.
- Contribution of class operators.

Conclusion

Summary

- Goal: apply mutation testing to C++.
- First step: definition of 37 class mutation operators.
- Second step: automation of mutation operators.
- Third step: evaluation and improvement of operators.

Future work

- Measure quality metrics.
- Improve the mutation tool:
  - Test coverage.
  - New standards of the language.
- Evolutionary Mutation Testing.
- Contribution of class operators.

Conclusion

Summary

- Goal: apply mutation testing to C++.
- First step: definition of 37 class mutation operators.
- Second step: automation of mutation operators.
- Third step: evaluation and improvement of operators.

Future work

- Measure quality metrics.
- Improve the mutation tool:
  - Test coverage.
  - New standards of the language.
- Evolutionary Mutation Testing.
- Contribution of class operators.

Conclusion

Summary

- Goal: apply mutation testing to C++.
- First step: definition of 37 class mutation operators.
- Second step: automation of mutation operators.
- Third step: evaluation and improvement of operators.

Future work

- Measure quality metrics.
- Improve the mutation tool:
  - Test coverage.
  - New standards of the language.
- Evolutionary Mutation Testing.
- Contribution of class operators.

Conclusion

Summary

- Goal: apply mutation testing to C++.
- First step: definition of 37 class mutation operators.
- Second step: automation of mutation operators.
- Third step: evaluation and improvement of operators.

Future work

- Measure quality metrics.
- Improve the mutation tool:
  - Test coverage.
  - New standards of the language.
- Evolutionary Mutation Testing.
- Contribution of class operators.

Thank you for your attention

TAROT-2015

Pedro Delgado-Pérez
pedro.delgado@uca.es