Software Testing and Validation

MUJAVA: AN AUTOMATED CLASS MUTATION SYSTEM

Group 12
João Vítor 99246
Guilherme Gago 100471
Guilherme Garrilha 110827

Introduction

- Object-Oriented (OO) Programming offers many advantages in regards to software development, and solutions to old problems;

- However, newer OO languages arise with newer problems, that require newer solutions.

Introduction

- Early work was focused on testing data abstraction and state behavior;
- Subsequent work looked into class testing, and issues regarding how many objects should be instantiated, or the order of classes to be tested;
- Recent work goes into integration of OO software and complete-class testing.

Context

- Researchers have started looking into ways to test **Polymorphism** and **Inheritance**.

```
class Shape
{
    public void draw() {
    }
}
class Square extends Shape {
   public void draw() {      // <-- overridden method
   }
   .    // other methods or variables declaration
 }
class Circle extends Shape {
   public void draw() {      // <-- overridden method
   }
    .    // other methods or variables declaration
}
class Shapes {
   public static void main(String[] args) {
      Shape a = new Square();       // <-- upcasting Square to Shape
      Shape b = new Circle();       // <-- upcasting Circle to Shape
      a.draw();          // draw a square
      b.draw();          // draw a circle
   }
}
```

```
class Maths {
    int addition(int a, int b) {
        return a + b;
    }
    int subtraction(int a, int b) {
        return a - b;
    }
}

public class Calculation extends Maths{
    int multiplication(int a, int b) {
        return a * b;
    }

    public static void main(String[] args) {
        Calculation calculation = new Calculation();
        System.out.println("Addition : "+ calculation.addition(20, 10));
        System.out.println("Subtraction : "+ calculation.subtraction(20, 10));
        System.out.println("Multiply : "+ calculation.multiplication(20, 10));
    }
}



OUTPUT:
Addition : 30
Subtraction : 10
Multiply : 200
```

Mutation testing!

- What is it?
- **Mutation testing:** fault-based technique that measures the effectiveness of test cases.
- Based on the assumption that a program will be well tested if the majority of simple faults are detected and removed.

Mutation testing!

- Simple faults are introduced into the program, creating a set of faulty versions, called mutants.
- Created from the original program by applying mutation operators, describing syntactic changes to the programming languages
- Test cases are used to execute the mutants with the goal to produce an incorrect output
- A test case that distinguishes the program from one+ mutants is considered effective

Mutation testing!

- One glaring issue is: it involves many executions of programs!
- Cost is a serious issue! :(

Mutation testing!

- **Do fewer**: run fewer mutant programs without going into intolerable loss in effectiveness

- **Do smarter**: distribute computational expense over several machines and/or factor the expense over several executions by retaining state information between runs

- **Do faster**: focuses on ways to generate and run mutant programs as quickly as possible

Mutation testing!

- This paper focuses on a faster approach for OO programming!
- It mainly focuses on reducing compilation times

Mutation testing for OO Programs

A major difference in OO testing, is the software changing levels at which testing will be performed out, they can be classified into 4 levels:

- **Intra-method level**: faults occur when the functionality of a method is implemented incorrectly.
- **Inter-method level**: faults are made on the connections between pairs of methods of a single class
- **Intra-class level**: tests are constructed for a single class, with the purpose of testing the class as a whole
- **Inter-class level**: when two+ classes are tested in combination to look for faults in how they’re integrated

Mutation testing for OO Programs

- New mutation operators that handle new types of faults
- Introduced by OO-specific features
- OO mutation systems should be able to extract information and execute programs from an OO standpoint
- i.e, classes and references to user defined types must be handled, like control data, inheritance and polymorphism relationships amongst components
- The paper focused on studying inter-class, OO, operators

Based on language features common to all OO languages

Language feature	Operator	Description
Access control	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	ISK	*super* key word deletion
	IPC	Explicit call of a parent's constructor deletion
Polymorphism	PNC	*new* method call with child class type
	PMD	Instance variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PRV	Reference assignment with other comparable type
Overloading	OMR	Overloading method contents change
	OMD	Overloading method deletion
	OAO	Argument order change
	OAN	Argument number change
Java-specific features	JTD	*this* keyword deletion
	JSC	*static* modifier change
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
Common programming mistakes	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

Based on language features common to all OO languages

Java specific language features

Common OO programming mistakes

Language feature	Operator	Description
Access control	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	ISK	*super* key word deletion
	IPC	Explicit call of a parent's constructor deletion
Polymorphism	PNC	*new* method call with child class type
	PMD	Instance variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PRV	Reference assignment with other comparable type
Overloading	OMR	Overloading method contents change
	OMD	Overloading method deletion
	OAO	Argument order change
	OAN	Argument number change
Java-specific features	JTD	*this* keyword deletion
	JSC	*static* modifier change
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
Common programming mistakes	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

Designed for faulty behavior when the class is integrated with other classes, modified or inherited from

Incorrect use can lead to a number of fails.

Allows the behavior of an object reference to differ depending on the actual type.

Language feature	Operator	Description
Access control	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	ISK	*super* key word deletion
	IPC	Explicit call of a parent's constructor deletion
Polymorphism	PNC	*new* method call with child class type
	PMD	Instance variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PRV	Reference assignment with other comparable type
Overloading	OMR	Overloading method contents change
	OMD	Overloading method deletion
	OAO	Argument order change
	OAN	Argument number change
Java-specific features	JTD	*this* keyword deletion
	JSC	*static* modifier change
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
Common programming mistakes	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

Allows two or more methods of the same class to have the same name as long as they have different argument signatures

Mutation testing is language dependent, mutation operators need to reflect language-specific features

Capture typical mistakes made when writting OO software.

Language feature	Operator	Description
Access control	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	ISK	*super* key word deletion
	IPC	Explicit call of a parent's constructor deletion
Polymorphism	PNC	*new* method call with child class type
	PMD	Instance variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PRV	Reference assignment with other comparable type
Overloading	OMR	Overloading method contents change
	OMD	Overloading method deletion
	OAO	Argument order change
	OAN	Argument number change
Java-specific features	JTD	*this* keyword deletion
	JSC	*static* modifier change
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
Common programming mistakes	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

Mutation tools for OO Testing

- Mutants are created from a test program using an automated mutation system.
- Test cases are added to try to “kill” the output of the original program from the mutant programs
- Mutation operators for Java change type and/or data structure declarations
- Must access information in a program from a OO standpoint

Reflection

- One of the first implementations of OO mutation was suggested using **Reflection**
- Reflection is the ability for a program to look upon itself, i.e, their structure and behavior, and manipulate it.

Why Reflection is good for OO

- Natural way to implement mutation analysis
- Extracts OO related info about a class by providing an object that represents a logical structure of the class definition, therefore parsing the program
- Provides an Application Programming Interface, API, to easily change the behavior of the program during execution
- Allows objects to be instantiated and methods to be invoked dynamically

Why Reflection is good for OO

- All of this allows Java programs to perform various functions
- Such as asking for the class of a given object, finding the methods in that class, and invoking said methods
- Java does not support full reflective capabilities!
- It only supports introspection, i.e, it can only look upon the data structures but not alter program behavior

A solution... Bytecode!

- Previous tools suffer from a very serious drawback, their horrendous low performance!
- Slow because they use an inefficient way to create mutant programs! Creating a copy of the original source code and changing for each mutation!
- To solve this, this paper proposed an approach that generates mutants directly from the bytecode!

Bytecode translation

- Bytecode translation inspects and modifies the intermediate representation of Java programs, bytecode.
- It handles bytecode directly, so it can process an off-the-shelf program, or a library without supplied source code
- It can be performed on demand at load time, when the Java Virtual machine loads a class file

Overview of a Java OO Mutation System

- This paper makes use of a mutation testing system that uses mutant schemata generation (MSG) and bytecode translation
- MSG is used to generate one metamutant program at the source level that incorporates many mutants
- MuJava works directly on the bytecode, thus requires two compilations
- The original source code, and the comp. of the metamutants generated with MSG.

Overview of a Java OO Mutation System

- Allows for faster performance than previous mutation systems
- So we can conclude MSG and bytecode translation can be significantly faster
- Reminder : Bytecode translation allows the structure of the bytecode to be changed directly, not requiring additional comp. times

Overview of a Java OO Mutation System

- The MSG method is adapted for class mutation operators that change the behavior of the program
- Creates a “meta” version of the test program that contains all mutants and requires a single compilation
- A new method based on bytecode translation is introduced for class mutation operators
- It changes the structure of the program

Overview of a Java OO Mutation System

- Bytecode translation allows the struct. of the bytecode to be changed directly, no need for further comp.
- And thus, mutants that change the behavior of the program are called behavioral mutants
- And the ones that change the structure are structural mutants

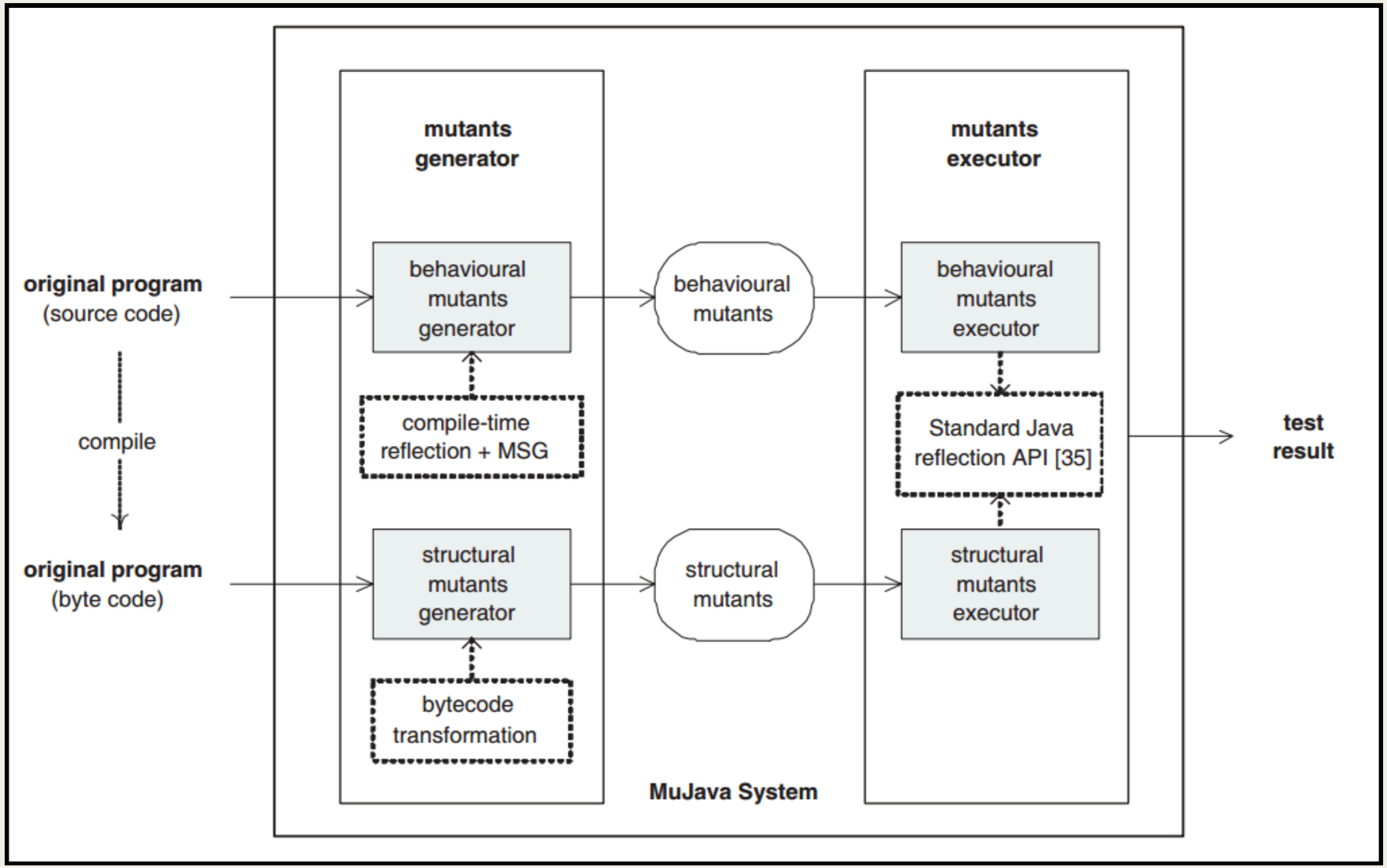
Generating and running behavioral mutants

- Non-OO mutants are all behavioral, there are tecnhiques that can be used for existing mutant generation
- This paper makes use of MSG, which as we've seen before, was found to be significantly faster for intra-class mutation operators
- MuJava however is focused on inter-class mutation.

MSG Method

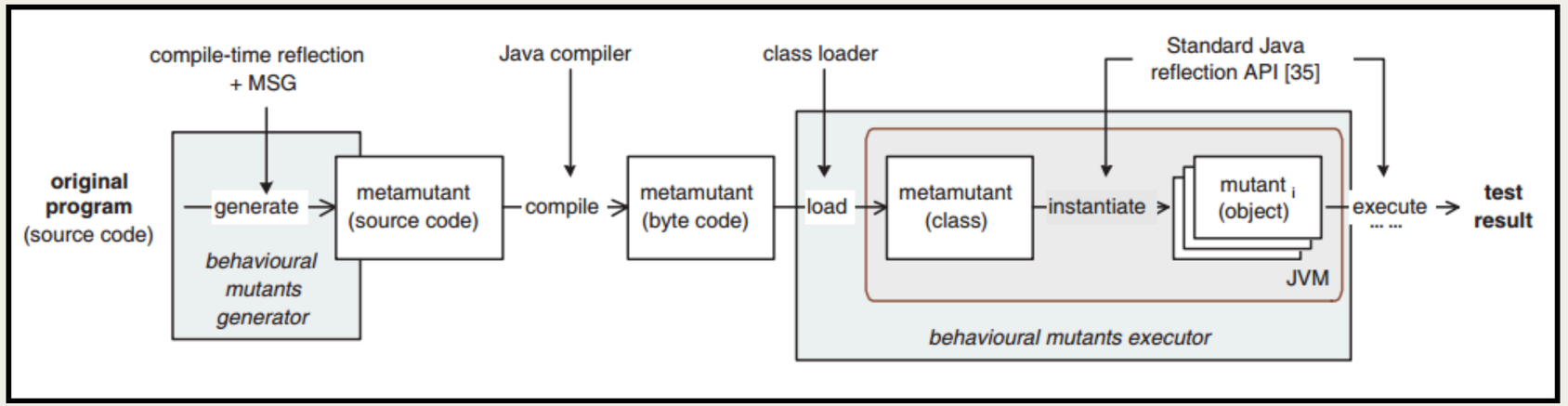
- Encodes all mutants into a specially parameterized program, that is, a metamutant.
- It is derived from a program under test, call it P
- This metamutant is then compiled used the same compiler used to compile P, running at compile-speeds.
- While running, the metamutant has the ability to function as any of the mutant programs of P.

Structure of MuJava



MSG Method

- For a program P, each mutant of P is formed as a result of a single modification to some statement in P
- Each P mutant only differs from the original by one mutated statement
- It is dictated by the set of mutation operators used



Metaprocedures

- Functions that correspond to an abstract entity in the schema
- A statement that has been changed to a generic form, is said to have been metamutated
- Metamutation is a synctactically valid change that embodies other changes
- When generating a metamutant of P, a list of mutant descriptors is produced.
- This list details other operations that can be used at each change stage to alter the program.
- Using this list the metamutant is dynamically instantiated to the function as any mutants of P

Notes for using the MSG method for OO programming languages

- Object references.

Notes for using the MSG method for OO programming languages

- Object references.

```
void f_oan (A obj, int p1, int p2, char p3)
{
    switch (mutantID)
    {
        case 1:  obj.f (p1, p3);    break;  // a.f(2, 'c');
        case 2:  obj.f (p2, p3);    break;  // a.f(3, 'c');
        default: obj.f (p1, p2, p3);        // original code
    }
}
```

Notes for using the MSG method for OO programming languages

- Object references.
- Polymorphism.

Notes for using the MSG method for OO programming languages

- Object references.
- Polymorphism.

```
private A pnc()
{
    switch (mutantID)
    {
        case 1 :  return (new B());      // mutated statement 1
        case 2 :  return (new C());      // mutated statement 2
        default : return (new A());      // original code
    }
}
```

Notes for using the MSG method for OO programming languages

- Object references.
- Polymorphism.
- Instantiation overhead.

Notes for using the MSG method for OO programming languages

- Object references.
- Polymorphism.
- Instantiation overhead.

```
if (MethodID == 3)
{
     if (StatementID == 8)
          {metamutated form of the 8th statement;}
     else
          {original form of the 8th statement;}

     ... ...
}
else
{original form of the 3rd method;}
```

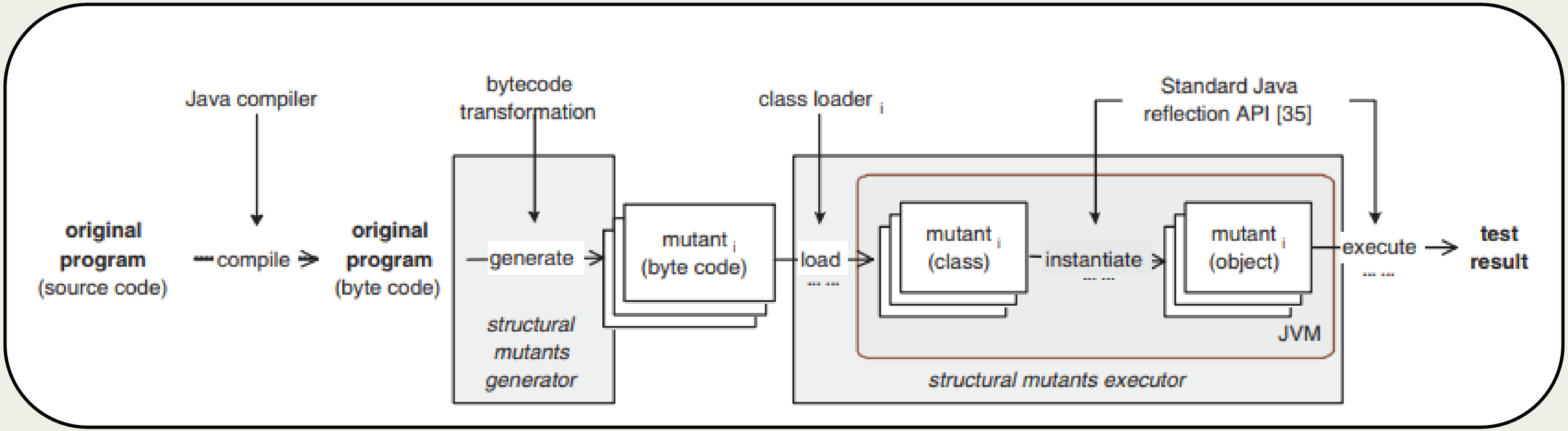
Notes for using the MSG method for OO programming languages

- Object references.
- Polymorphism.
- Instantiation overhead.
- Number of mutants.

BCEL

- A package of classes that describe ‘static’ constraints on class files.
- A package to dynamically generate or modify bytecode.
- Various code examples and utilities, a tool to convert class files into HTML, and a converter from class files to the Jasmin assembly language.

Generating structural mutants



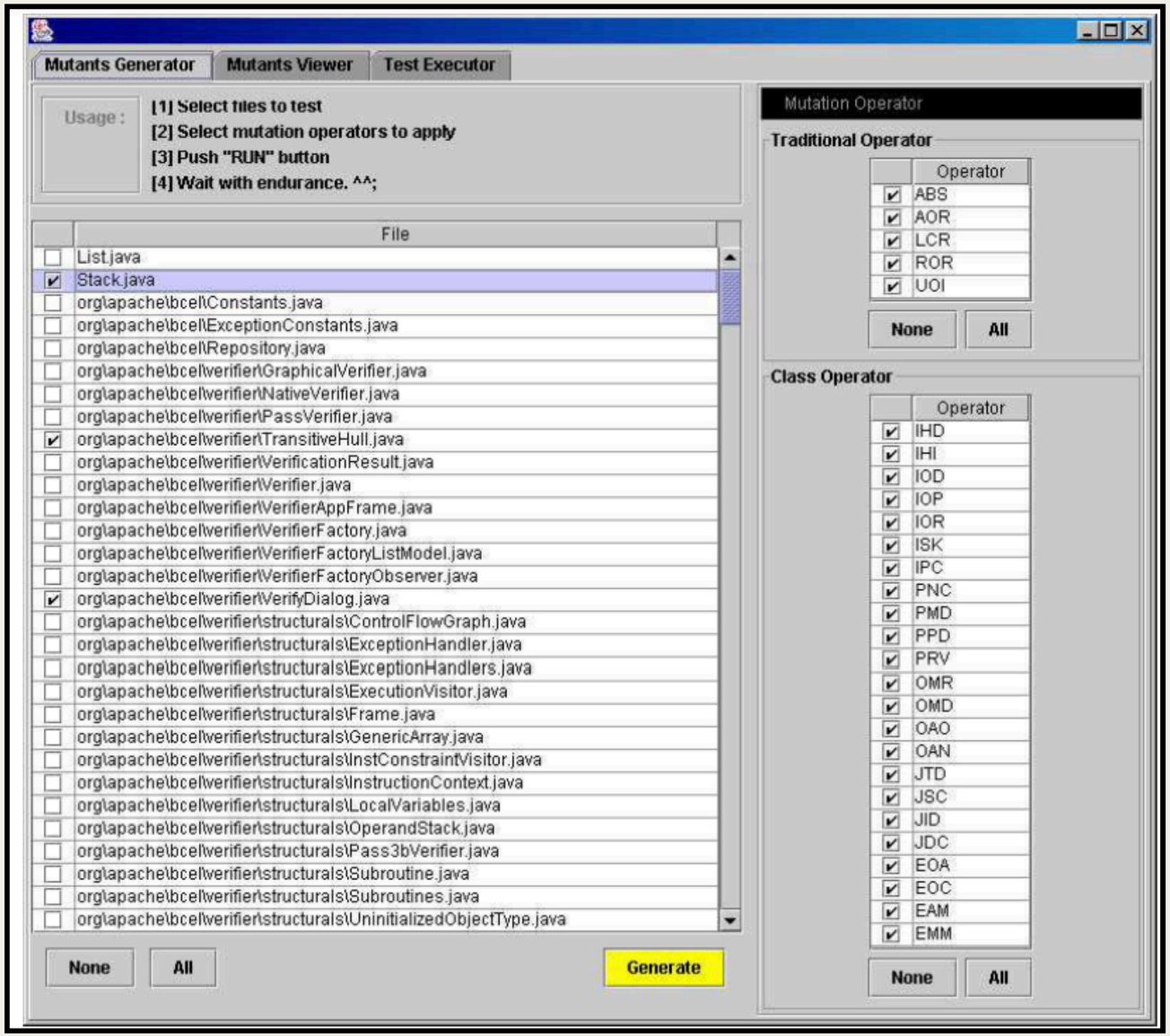
The MuJava tool

Major functions:

- Generate mutants.
- Analyse mutants.
- Run test cases supplied by the tester.

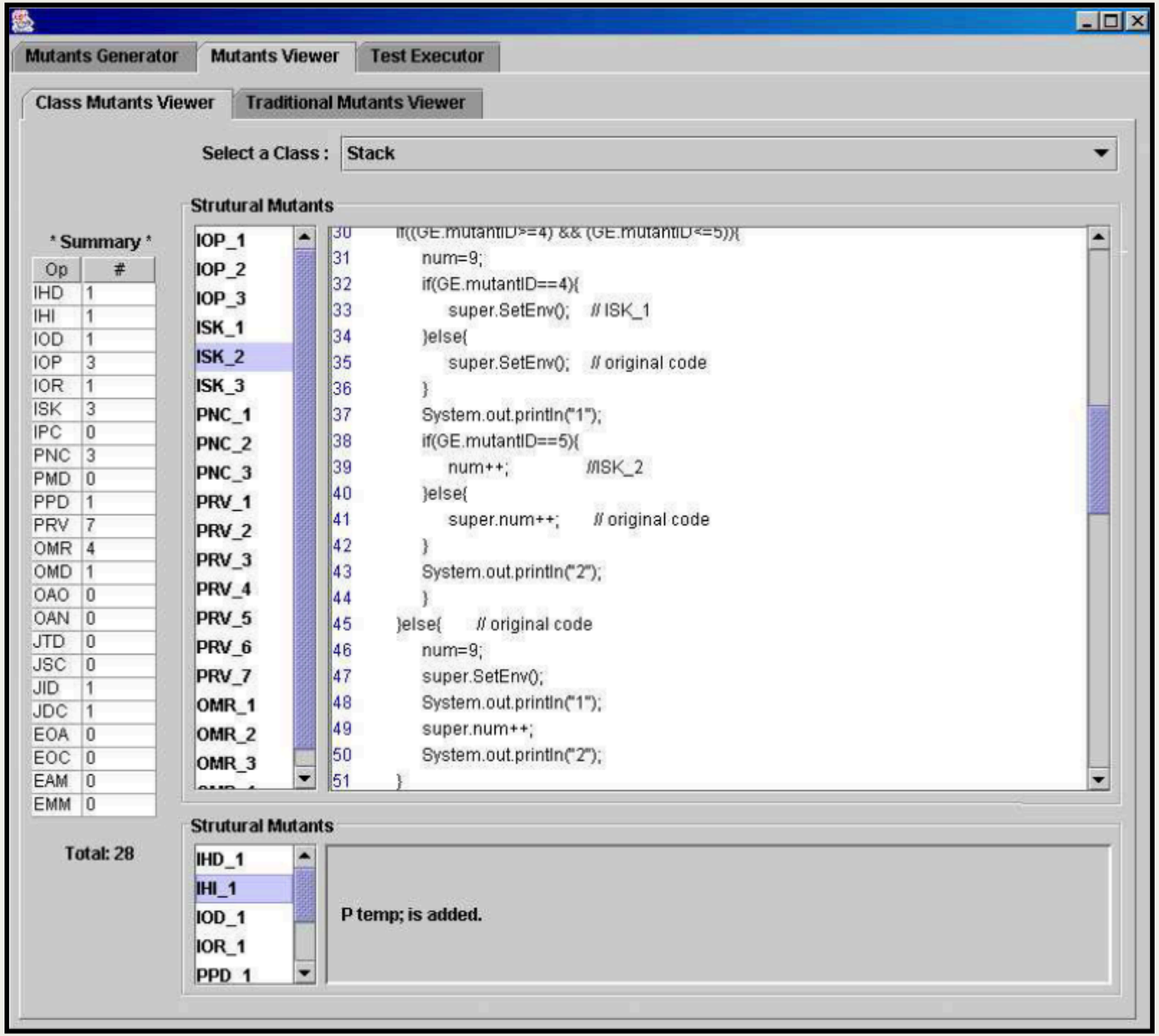
The MuJava tool

Testers can select files and mutation operators to generate mutants. Generated mutants are displayed in the 'Mutants Viewer' tabs.



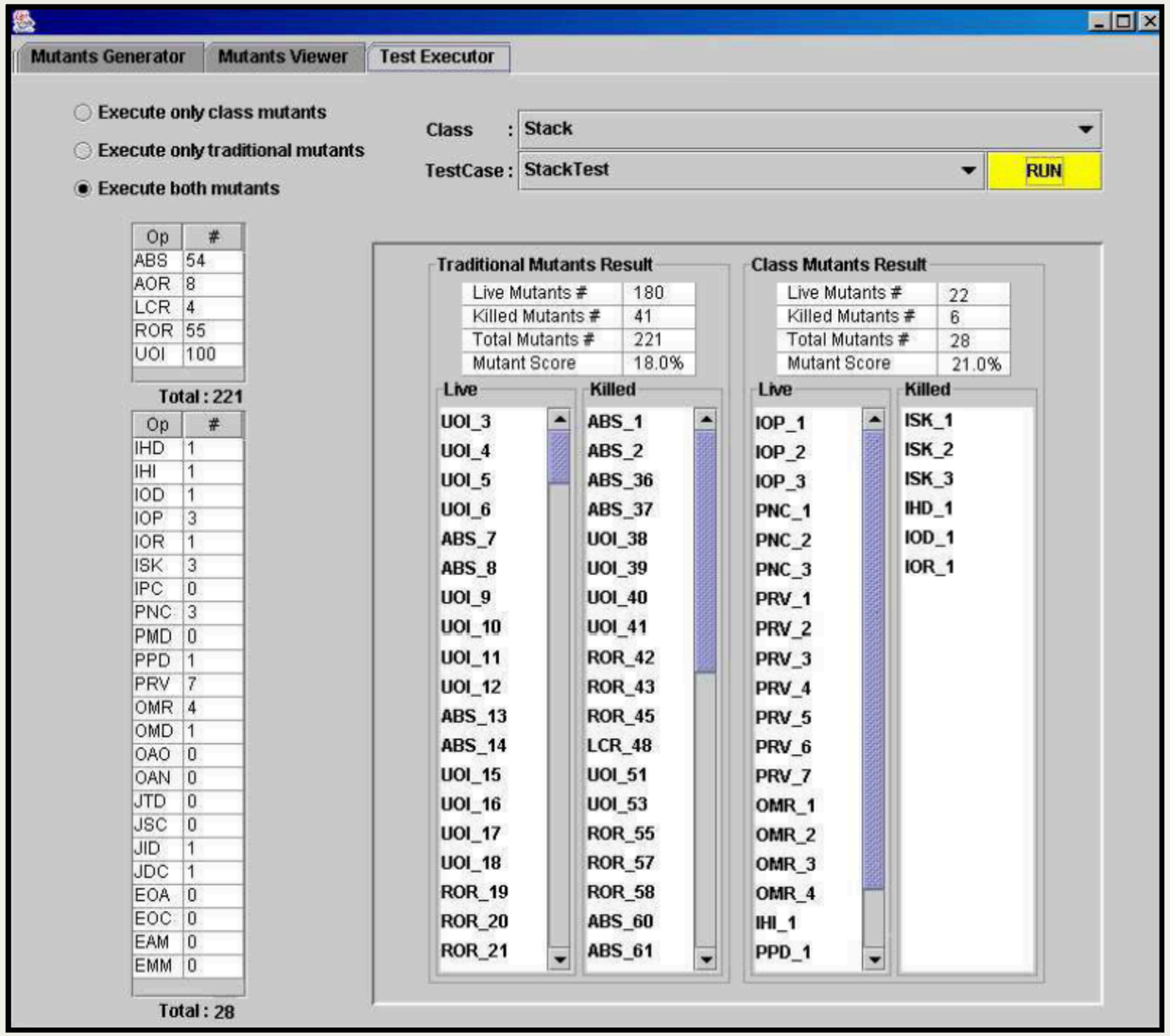
The MuJava tool

Shows changes in the source code due to mutants. Structural mutants show brief descriptions as they are generated from bytecode.



The MuJava tool

Executes mutants against the test set and displays the mutation score, along with information about live and dead mutants



Experiental Performance Evaluation

- Mutation tool combining MSG and bytecode translation outperforms one using compile-time reflection?
- Number of mutants created by each operator.
- Performance of MSG versus bytecode translation.

Experimental subjects

- The BCEL system was chosen because it is widely used and for its convenient size.
- 264 classes generated 3812 inter-class mutants, with 1.53 times more behavioral mutants than structural.
- 7 classes with varying sizes and at least 8 structural mutants were chosen out of the 264.

Experimental procedure

- Test sets, that satisfy branch coverage for all methods, were created to kill all class mutants for each class.

- The same set of tests and mutants were used to compare MuJava with a system that separately compiles each mutant.

- Both mutant generation and test execution time was compared.

Experimental results

- **MSG/bytecode translation -** mutant generation and test execution times were 9.3 times and 2.1 times faster than with separate compilation.

- **Just MSG (behavioral mutants only) -** mutant generation and test execution times were 6.6 times and 7.5 times faster than with separate compilation.
- Reloading the class each time while testing is costly.

Experimental results

- **Just bytecode translation -** mutant generation was 44 times faster than with separate compilation. Test execution time was the same.

- MSG method is, over all, faster than bytecode translation. But, since MSG can’t generate structural mutants, bytecode translation is the best option for those.

Conclusion/Summary

- This work goes over the mutation operators required for inter-class testing in OO software. They generate either behavioral or structural mutants.

- It then describes how the already existing MSG method was adapted to generate behavioral mutants.

Conclusion/Summary

- And introduces a new method,that utilizes bytecode translation, to generate structural mutants.
- With these two methods combined, both generating mutant classes and testing them becomes much faster.