Journal of Network and Innovative Computing.
ISSN 2160-2174 Volume 3 (2015) pp. 047–059
© MIR Labs, www.mirlabs.net/jnic/index.html

A novel approach to the design and implementation of mutation operators for Object-Oriented programming language

Qianqian Wang[1], **Hirohide Haga**[2]

[1]IBM China,
2F, Building 10, 399 Keyuan, Zhang Jiang High Tech Park, 201203, Shanghai, China
wangqqw@cn.ibm.com

[2]Faculty of Science and Engineering, Doshisha University,
1-3, Miyakotani, Tatara Kyotanabe, 610-0321, Japan
hhaga@mail.doshisha.ac.jp

Abstract: **Software testing allows programmers to determine and guarantee the quality of software system. It is one of the essential activities in software development process. Mutation analysis is a branch of software testing. It is classified as a fault-based software testing technology. Unlike other software testing technologies, mutation analysis assesses the quality of software test cases and therefore improves the efficiency of software testing by measuring and improving the quality of test cases set. Mutation analysis works by generating mutants, which describe the faults that may occur in the software programs. Mutants are generated by applying mutation operators on original software program. Mutation operator is a rule that specifies the syntactic variations of strings.**

There have been several works about mutation analysis support system for conventional languages such as C and FORTRAN. However, there are a few for object-oriented languages such as C++ and Java. This article aims to propose a novel approach to design and implement mutation operators for object-oriented programming language. The essential idea of proposed method is the usage of JavaML as the intermediate representation to implement mutation operators: it first converts the original program into JavaML document; then implements mutation operator for JavaML document and gets the mutated JavaML document with the help of DOM – a tool to process JavaML document– finally it converts the mutated JavaML document into mutant program. Typical six mutation operators are implemented. Implementation detail will be given in this article.

Keywords: Software Testing, Mutation Analysis, Object Oriented Langauge, XML

I. Introduction

Software testing is one of the most important activities in software development process[1]. It observes the execution of programs to validate whether it behaves as intended and produces the expected results. Mutation analysis is a fault-based testing technology, which assesses the quality of test cases set and therefore improves the quality of programs[2, 3]. A high-quality test cases set can detect more possible faults in software and guarantee the quality of software.

In mutation analysis, mutation operators are used to generate mutated programs (hereinafter called *mutants*) from original program; test cases set is applied on both original program and mutants to get their behaviors and results. Then the behaviors and results are observed carefully to evaluate and improve the quality of test cases set.

The purpose of this article is to propose a novel approach to the design and implementation of mutation operators for Object-Oriented programming language[4]. The essential idea of this method is the usage of JavaML (Java Markup Language)[5] as an intermediate representation to implement mutation operator. Proposed method first converts the original source program into JavaML document. Then it applies mutation operators to converted JavaML document and gets the mutated JavaML document with the help of DOM (Document Object Model)[6, 7], – a tool to process JavaML document. Finally it makes reverse conversion from the mutated JavaML document into mutant program.

In this article, OO specific six mutation operators including AMC (access modifier change), SKD ("super" keyword deletion), SMD (static modifier deletion), TKD (deletion of " this " keyword), PCD (parent constructor deletion) and OMD (overloading method deletion) are implemented.

Section II introduces the concept of software testing, and its necessary components; test cases and coverage. Concept of mutation analysis is also introduced in this section. Section III describes the Extensible Markup Language (XML) and one of its special forms JavaML, which is used for representing Java source code in this article. A method called DOM (Document Object Model) is also introduced in this section to processing XML documents. Finally a novel approach to implement mutation operator with using JavaML is proposed. In section IV, which is the main part of this article, we will describe the details of the design and implementation of six selected mutation operators

II. Software testing and mutation analysis

A. *Software Testing*

The purpose of software testing is to guarantee that program is developed according to the specification. The program could be executed correctly under all possible circumstance. Software testing provides objective view of the correctness, completeness and quality for software products. It is an essential technology for all software developments
Software testing can be processed through a number of ways; and it is performed using testing cases. Therefore, test case is one of the main components of software testing activity.
Before introducing the concept of test case, the following definitions are necessary to be acquired [2]:

1. **Test Case Value:** the input values necessary to complete some execution of software under test.
2. **Prefix Values:** any inputs necessary to put the software into the appropriate state to receive the test case values.
3. **Postfix Values:** any inputs that need to be sent to the software after the test case values are sent.

A test case is composed of the test case value, expected results, prefix values, and postfix values necessary for a complete execution and evaluation of the software under test. A test cases set is simply a set of test cases.
The number of potential inputs for most programs is too large to be effectively covered. Thinking about a C parser, the number of the potential inputs to the C parser is a feasible sequence of program fragments of C programming language, and thus it is infinite and could not be explicitly enumerated. Therefore, we have to consider how to reduce the number of test cases without reducing the quality of software. One of the most popular methods to resolve this problem is the usage of "coverage measure". Coverage measure is a measure used to describe the degree to which the source code of program is tested by a particular test cases set. We will provide the formal definition of coverage and related terminology as follow:

- **Coverage:** given a set of test requirements *TR* for a coverage criterion *C*, a test cases set *T* satisfies *C* if and only if for every test requirement *tr* in *TR*, at least one test *t* in *T* exists such that *t* satisfies *tr*.
- **Coverage criterion:** a coverage criterion is a rule or collection of rules that impose test requirements on a test cases set.

Since we could not test all the possible test cases, coverage is used to evaluate the effectiveness and performance of test cases set [3]. Higher coverage of test cases set could be more efficient and have higher performance in software testing.

B. *Mutation Analysis*

Mutation analysis is a branch of software testing. Its purpose is to measure and improve the quality of test cases set. Mutation analysis is a fault-based software testing technology[3]. Unlike other software testing techniques, it measures the effectiveness and performance of test cases, and helps increase the coverage of test cases set. Mutation analysis is developed under two hypotheses[8].

- **Competent programmer hypothesis:** software programs are often different from the correct version of the programs in several ways. And most programs are nearly correct.
- **Coupling effect hypothesis:** complex fault of program is made up of simple faults.

Mutation analysis is composed of three parts: "original code", "mutation operator," and "mutant." The original code is a string in the program that is syntactically correct. In mutation analysis, mutation operator is applied on original code to get mutant.

C. *Mutants*

Mutant is also a program which is generated from the original program to be tested; it is the result of one application of a mutation operator to the original program. By applying mutation operator, original program is syntactically modified. Mutant is said to be *killed* if and only if the execution of at least one test case on the original program is different from the result of on the mutant.
There are mainly three kinds of mutant in mutation analysis:

- **Killed mutant:** a mutant that is killed by at least one test case.
- **Still-live mutant:** a mutant whose behavior is identical to original code for all prepared test cases. If we add further test case, still-live mutant may be killed.
- **Equivalent mutant:** a mutant which is functionally equivalent to the original program. The equivalent mutants produce the same result as the original program for any test cases. Therefore, no test case can kill them.

Mutation score is the percentage of non-equivalent mutants killed by the test cases set. Each mutant is generated by applying one mutation operator on the original program once.

D. *Mutation Operators*

Mutation operator is a rule that specifies syntactic variations of strings generated from a grammar. Mutation operators should be carefully designed for different programming language because of the different features of languages.
Taking mutation operator **AOR** (Arithmetic Operator Replacement) as an example, each occurrence of one of the arithmetic operators (+, −, ⋆, /) is replaced by other arithmetic operators. Following is an example of original program and its mutants.

Original		Mutants
		$x = a \underline{-} b$
$x = a \underline{+} b$	$\longrightarrow$	$x = a \underline{*} b$
		$x = a \underline{/} b$

The **AOR** operator is applied to the original code and three mutants are generated. There are common mutation operators which fit to various kinds of programming languages such as **AOR**. But some mutation operators are language-specific. In order to design mutation operators for specific languages, we have to consider the features of target language.

E. Process of Mutation Analysis

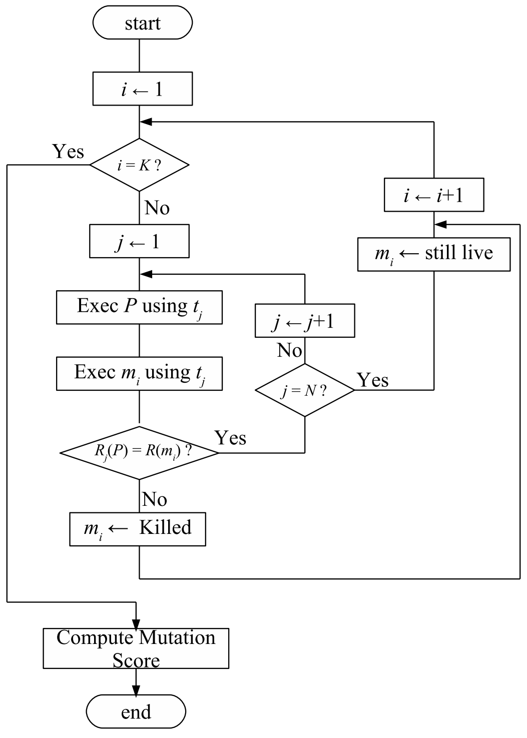

Figure. 1: Process of mutation analysis

Figure 1 describes the principle steps of mutation analysis. There are several steps in mutation analysis. They are:

1. Prepare original program *P* to be tested.
2. Prepare test cases set *T*. Each test case $t_j(1 \le j \le N)$ is included in *T*.
3. Generate mutants $m_i(1 \le i \le K)$. Generated mutants are included in mutants set *M*.
4. Eliminate equivalent mutants from *M*.
5. Execute *P* using test case t_j and get the result $R_j(P)$.
6. Execute m_i using test case t_j and get the result $R_j(m_i)$.
7. When $R_j(P) \neq R_j(m_i)$, m_i is marked as *killed mutant*.
8. If $\forall j(R_j(m_i) = R_j(P))$, m_i is marked as *still-live mutant*.
9. Mutation score *MS* will be computed using following formulae:

$$MS = \frac{\#\ of\ killed\ mutants}{\#\ of\ all\ mutants - \#\ equivalent\ mutants}$$

This *MS* indicates the *quality* or *effectiveness* of test cases set *T*. When almost all mutants are killed by test cases set *T*, $MS \cong 1$ and lower *MS* means that there are relatively many still-live mutants. There are three possibilities of still-live mutants.

a) mutated code is not executed for all test cases t_j,

b) mutated code generates identical result of for specific test case,

c) mutant is an equivalent mutant.

In case a) and b), further test cases must be added to the test cases set *T*. Therefore, by using *MS*, test engineers can estimate the quality or effectiveness of test cases set *T*.

III. Object-oriented Programming

A. Brief Introduction of Object Oriented Programming

Object-oriented programming (OOP) is a programming concept that enables the programmer to associate a set of procedures with each type of data structure[4]. This data structure is called ‘Object’. An object is considered as an item that can have several attributes and could perform a set of related activities, which is called as ”method.”
A class in OOP is simply a representation of a type of object. It is the “blue print” from which the individual instance objects are created. Class is composed of three things: a name, attributes, and methods. Following is an example of class definition.

Name	`Student`

Attributes	`Name: String`
	`Age: int`

Methods	`void Students()`
	`int getAge()`

Three features of object-oriented programming related to design mutation operators for OOP are as follow:

(1) **Encapsulation:** Encapsulation is the mechanism under which the details of a class are kept hidden from its user. The advantage of encapsulation comes when the implementation of the class changes but the interface remains the same.

(2) **Inheritance:** Inheritance is the mechanism under which specific classes are made from another, sometimes more general ones. The child or derived class inherits all attributes and methods of its parent or base class, and the child class can add attributes and methods of its own.

(3) **Polymorphism:** Polymorphism is a mechanism under which it is possible to assign different meaning or usage to an entity in different contexts. The entity can be a variable, method or object. Polymorphism enables a programmer to make use of an entity in several different forms without affecting the original identity of the entity.

B. Mutation Operators for OO Language

There are several proposals for the set of mutation operators of OOP[9, 10]. In this article, we introduce mutation operators for OO language from two levels.

- **Method level:** Method level operators modify the expressions by inserting, replacing or deleting the primitive operators. The method-level operators are usually classified into six types: arithmetic operators, relation operators, conditional operators, shift operators, logical operators, and assignment operators. Their brief descriptions are shown in Table 1.

Table 1: Method-level Mutation Operators for OO Language

Type	Operator	Description
Arithmetic	**AOR**	Arithmetic Operator Replacement
	AOI	Arithmetic Operator Insertion
	AOD	Arithmetic Operator Deletion
Relation	**ROR**	Relational Operator Replacement
Conditional	**COR**	Conditional Operator Replacement
	COI	Conditional Operator Insertion
	COD	Conditionial Operator Deletion
Shift	**SOR**	Shift Operator Replacement
Logical	**LOR**	Logical Operator Replacement
	LOI	Logical Operator Insertion
	LOD	Logical Operator Deletion
Assignment	**ASR**	Assignment Operator Replacement

Table 2: Class-level Mutation Operators for OO Language

Type	Operator	Description
Encapsulation	**AMC**	Access Modifier Change
Inheritance	**HVD**	Hiding variable deletion
	HVI	Hiding variable insertion
	OMD$_{\text{I}}$	Overriding method deletion
	OMM	Overridden method moving
	OMR	Overridden method rename
	SKI	Super keyword insertion
	SKD	Super keyword deletion
	PCD$_{\text{I}}$	Parent constructor deletion
Polymorphism	**ATC**	Actual type change
	DTC	Declared type change
	PTC	Parameter type change
	PCI	Type cast operator insertion
	PCD$_{\text{P}}$	Type cast operator deletion
	PCC	Cast type change
	RTC	Reference type change
	OMC	Overload method change
	OMC$_{\text{P}}$	Overloading method deletion
	OAN	Argument of overloading method change
Java-specfic	**JTI**	"This" keyword insertion
	JTD	"This" keyword deletion
	JSI	Static modifier insertion
	JSD	Static modifier deletion
	VID	Variable initialization deletion
	DCD	Default constructor deletion
	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Access method change
	EMM	Modifier method change

- **Class level:** The class-level operators make changes to syntax by inserting, deleting, or modifying the expressions in the object-oriented programs. Since this article chooses Java as an example language, the class-level mutation operators for Java programming language are classified into four types with their descriptions in Table 2 They are encapsulation, inheritance, polymorphism, and Java-specific.

IV. IST: Integrated Software Testing Environment

We are now developing the IST (*I*ntegrated *S*oftware *T*eating Environment). IST is an integrated environment for software testing activity. In IST, there is a program database which contains programs to be tested. Several testing support tools exist around this program database.

In program database, all programs to be tested are represented as XML form. This is mainly because of the flexibility of XML form. XML form is represented in text. Therefore, both machine and human can read and use it. XML form can include structural information. This means that syntax information can be embedded in XML form. Therefore, by representing all programs in a XML form, tools surrounding program database need not to parse the program when they execute specific functions. Figure 2 represents the conceptual structure of IST.

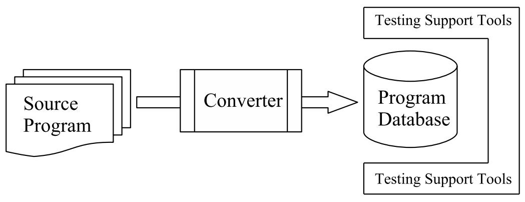

Figure. 2: Conceptual Structure of IST

V. JavaML and Mutation Operators

A. XML – Extensible Markup Language

JavaML is one instance of XML (Extensible Markup Language). XML is a markup language describing the structural information of documents. It is a subset of SGML, the Standard Generalized Markup Language [9]. The XML language is designed to represent the set of data in an easily understandable format both for machine and human being.

Arbitrary definitions of user-defined markup tags are allowed in XML, and thus making XML more flexible to be adapted for various application fields.

1) Syntax of XML

Figure 3 shows a simple XML document, which is sufficient for introducing the syntax of XML document. XML document consists of texts marked up with tags, which are enclosed in angle braces

```
1  <?xml version="1.0" encoding="GB2312"?>
2  <MyProduct>
3  <Product>
4    <ProductId>123</ProductId>
5      <Price type "= "dollar>870</Price>
6      <Size>1</Size>
7    </Product>
8    <Product>
9      <ProductId>124</ProductId>
10     <Price type "= "dollar>200</Price>
11     <Size>4</Size>
12   </Product>
13 </MyProduct>
```

Figure. 3: An example of XML document

a) **XML declaration:** Line 1 in Figure 3 is an XML declaration. It defines the version of XML and the character encoding used in the document. In this example, the version of XML is '1.0' and the encoding type is 'GB2312'.

```
<?xml version="1.0" encoding="GB2312"?>
```

b) **Element:** Elements in XML document have opening and closing tags with same name. Line 2 of this example is the root element of this XML document. It encloses all the other elements.

```
<MyProduct>……</MyProduct>
```

Each XML document contains only one root element, all elements could have child elements or sub elements. Figure 4 shows the relationship of all elements. In the example shown in Figure 3, the element enclosed by the tag `<Product>` and `</Product>` is the child element of the root element of this XML document; `<ProductId>`, `<Price>`, and `<Size>` are also the sub-children of the child element `<Product>`. Shortly saying, XML has a recursive structure.

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

Figure. 4: Elements of XML document

1. **Attribute:** XML elements may have attributes in the starting tag. These attributes provide additional information of elements. Each attribute is described as `"name=value"` pairs. In the above example, the element `<Price>` has attribute named `"type"` and its value is `"dollar"`, which describes the currency unit.

```
<Price type="dollar">870</Price>
```

2) Processing XML documents with DOM

Document Object Model (DOM) defines a standard model for accessing documents like HTML and XML[6] . DOM is divided into three independent parts:

a) Core DOM – standard model for any structured document

b) XML DOM – standard model for XML document

c) HTML DOM – standard model for HTML document

XML DOM defines the objects and properties of all XML elements, and the methods (interfaces) to access them.

3) DOM node and node tree

In XML DOM, everything from an XML document is regarded as node:

Entire XML document	$\longleftrightarrow$	Document node
XML element	$\longleftrightarrow$	Element node
Text content of XML element	$\longleftrightarrow$	Text node
Attribute	$\longleftrightarrow$	Attribute node
Comment	$\longleftrightarrow$	Comment node

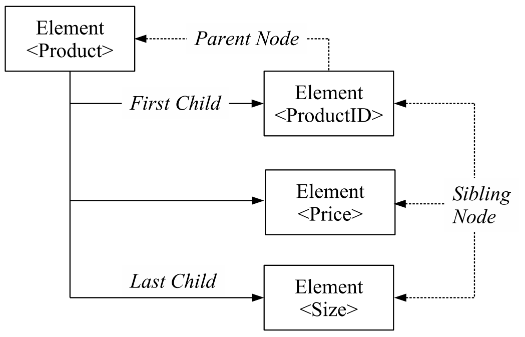

Figure. 5: Relationship of nodes

The relationship between each type of node is described in Figure 5:
The XML DOM represents the entire XML document as a tree-like data structure. This structure is suitable for processing by programs. Figure 6 is the DOM node tree representation of XML example document in Figure 3.

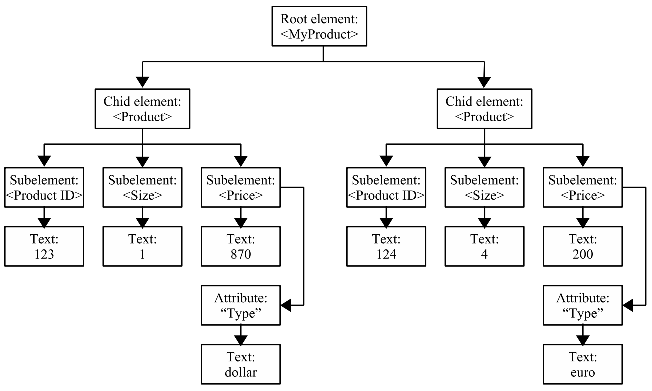

Figure. 6: DOM tree

4) DOM methods

DOM offers many methods (interfaces) to process XML document[7]; methods listed in Table 3 is most commonly used to processing XML document.
With the methods in Table 3, we could get the value of node, change, remove, replace, create and add nodes in the XML document.

B. JavaML

JavaML is an XML document for describing Java programming language. It provides a complete self-describing representation of Java source code in XML[6]. In our research, the converter‘ `JJmlt.jar` ’[11] is applied to convert Java source code into JavaML. Figure. 7 is a sample Java source code and Figure.8 is a corresponding JavaML representation generated by `JJmlt.jar`.
JavaML can represent the syntactic structure of Java source code directly by tags in XML document.
As JavaML is an XML document, it inherits all advantages from general XML document: it is easy to be parsed, and could be processed by plentiful tools.
The key point of our approach is to realize mutation operators in the domain of JavaML document rather than Java source

Table 3: Commonly used DOM methods

Method	Description
`loadXML`	Load the DOM tree of XML document into memory.
`getElementsByTagName`	Returns a collection of elements that have the specified name.
`getAttribute`	Gets the value of the attribute.
`getAttributeNode`	Gets the attribute node.
`createAttribute`	Creates a new attribute with the specified name.
`removeAttribute`	Removes or replaces the named attribute.
`setAttribute`	Sets the value of the named attribute.
`item(IXMLDOMNodeList)`	Allows random access to individual nodes within the collection.
`createNode`	Creates a node using the supplied type, name, and namespace.
`createTextNode`	Creates a text node that contains the supplied data.
`getNodeName`	Returns the name of a node
`hasChildNodes`	Provides a fast way to determine whether a node has children.
`appendChild`	Appends a new child node as the last child of the node.
`firstChild`	Returns the first child of the element
`lastChild`	Returns the last child of the element
`removeChild`	Removes the specified child node from the list of children and returns it.

```
package com.wqq.java.examples;
import com.wqq.example.Parent;

public class Child extends Parent{
    String name;
}
```

Figure. 7: Sample Java Source Code

code. We first get the corresponding JavaML document of the Java source code by using converter "JJmlt."[11] Then we apply mutation operators on JavaML document and get the mutated JavaML document. Finally the conversion from the mutated JavaML to Java source code will be executed. The converted Java source code is the desired mutant. To show this approach intuitively, we draw the schematic illustration in Figure. 9.

VI. Design and Implementing Mutation Operators on JavaML

Six operators are selected primarily to be implemented to verify the feasibility of proposed approach. Table 4 shows all the mutation operators explained in our experimental implementation.

Table 4: Experimentally implemented mutation operators

Mutation Operator	Brief Description
AMC	Access modifier change
SKD	Deletion of '`super` keyword
SMD	Static modifier deletion
TKD	Deletion of '`this`' keyword
PCD	Parent constructor deletion
OMD	Overload method deletion

```
<?xml version="1.0" encoding="UTF-8"?>
<java-source-program>
  <java-class-file name="testttt.java">
    <package-decl name="com.wqq.example"/>
    <import module="com.wqq.example.Parent"/>
      <class name="Child" visible="public">
        <superclass name="Parent"/>
        <field name="name">
          <type name="String"/>
        </field>
      </class>
  </java-class-file>
</java-source-program>
```

Figure. 8: Corresponding JavaML representation

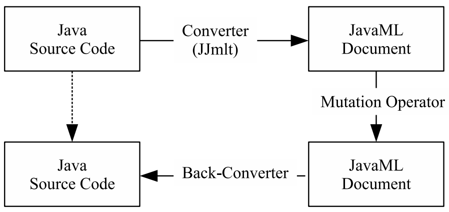

Figure. 9: Schematic illustration of proposed method

A. *Implementing AMC Operator*

Mutation operator **AMC** (Access Modifier Change) is designed to test the encapsulation feature in OO programming language.

By changing the access level for each instance variable and method to other access levels, mutation operator **AMC** helps software testers ensure the correctness of accessibility. The generated mutant would be killed only if the new access level denies access to another class or allows access that causes name conflict. Following is an example of original and its mutated codes.

Original code	Mutant
`public int x`	`private int x;`
	`protected int x;`

There are mainly three kinds of access modifier in Java. They are `public`, `private` and `protected`. These modifiers are used to set the access level for variables, methods, classes and constructors. Since the access modifier of constructor is always `public`, we only considers about that of methods other than constructor, variables, and classes.

The original code of Java to be mutated and its corresponding JavaML are as follows:

Original code	Corresponding JavaML

`public int x`	`<field name="x" visible="public">`
	`  <type name="int" primitive="true"/>`
	`</field>`

`private void nextLine()`	`<method name="nextLine"`
	`  visible="private" id="···">`
	`   ...`
	`</method>`

The **AMC** operator could be implemented by editing the value of attribute ' `visible` ' in node tag `<field>` and `<method>`. Figure 10 is a flowchart of **AMC** implementation.

By using DOM, we can edit XML document freely. DOM is applied to access the node and attribute in XML document. The core part of source code of this implementation is as

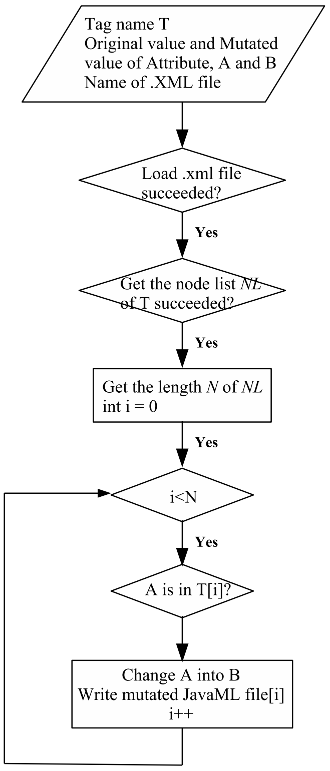
Tag name T
Original value and Mutated value of Attribute, A and B
Name of .XML file
Load .xml file succeeded?
Yes
Get the node list NL of T succeeded?
Yes
Get the length N of NL
int i = 0
Yes
i<N
Yes
A is in T[i]?
Change A into B
Write mutated JavaML file[i]
i++

Figure. 10: Flowchart of **AMC**

follow. Firstly, XML document is loaded by using the code shown below.

```
private DocumentBuilderFactory dbf;
private DocumentBuilder db;
private Document doc;
private Element element;
try{
    db=dbf.newDocumentBuilder();
    doc=db.parse(inputFileName);
    root = doc.getDocumentElement();
}
catch(ParserConfigurationException e)
    {e.printStackTrace();}
catch(SAXException e)
    {e.printStackTrace();}
catch(IOException e)
    {e.printStackTrace();}
```

With the codes of

```
NodeList nl = doc.getElementsByTagName(tagname);
int numOftag = nl.getLength();
```

the main program could get the node list with a given tag name.
The following codes first check the existence of attribute `visible`, and set the value of `visible` into the mutated value.

```
element = (Element) nl.item(i−1);
Boolean hasAttribute=element.hasAttribute ("visible");
if (hasAttr) {
    element.setAttribute("visible", desName);
}
```

Steps shown above are essential steps of implementing **AMC** operator. The original and mutated parts in JavaML are listed in Figure 11.

Tag name	Original JavaML	Mutated JavaML
`field`	`<field name="x" visible="public">`	`<field name="x" visible="public">`
`method`	`<method name="nLine" visible="private" id="···">`	`<method name="nLine" visible="protected" id="···">`

Figure. 11: Mutated result of **AMC** operator

B. Implementing SKD

Mutation operator **SKD** ("`super`" keyword deletion) is designed to test the inheritance feature in OO programming language. This operator deletes the "`super` keyword so that any references to the variable or method go to the overriding method or variable. The mutant can only be killed by test case that causes different behavior with the hiding variables or the overriding methods.

Original code	Mutant
`super.name = name1;`	`name = name1;`

The keyword `super` is referencing variable that is defined at the immediate parent class object. The keyword `super` is used to refer immediate parent class and/or instance variable and to invoke immediate parent class method.
In Java programs, there are five cases, under which the **SKD** operator could be applied:

1. Subclass has the member variable with same name of its immediate super class.
2. Subclass does not have the member variable with same name of its immediate super class.
3. Subclass has overridden method. The subclass invokes the super method within the overridden method.
4. Subclass overrides the method, and invokes the super method in other method.
5. Subclass does not contain the overridden method, and only invokes the super method.

In case 2) and 5), the mutant has no semantically difference with the original program, thus it is determined as equivalent mutant and will not be considered. In case 3), the mutant has syntax error, thus it will also not be considered in our implementation.
The original code of Java under mutation analysis and its corresponding JavaML are shown in Figure 12.
Mutation operator SKD is implemented by following the steps in Figure 13.
DOM is applied to access the node and attribute in XML document, the core source code of this implementation is as follows. Following code shows the process of modifying the child node of the parent node tag.

1. If the parent tag is `<field-set>`, the following source code work to delete its child node `<super>` and append new child node `<var-set>`.

   ```
   if (nod.getFirstChild().getNodeName() == "super") {
       Element newNode = doc.createElement("var−set");
       newNode.setAttribute("name",attrName);
       nod_parent. removeChild (nod);
       nod_parent.appendChild(newNode);
   }
   ```

2. If the parent tag is `<field-access>`, the following codes would delete its child node `<super>` and append new child node `<var-ref>`;

`super.comName=name;`	`<assignment-expr op = "=">` `<lvalue>` `<field-set field = comName">` `<super/>` `</field-set>` `</lvalue>` `<var-ref name = "name"` `idref="com.wqq.javaex.Child:frm-1"/>` `</assignment-expr>`
`System.out.println(` `super.comName);`	`<filed-access field="comName">` `<super>>` `</filed-access>`
`System.out.println(` `super.disName);`	`<filed-access field="comName">` `<send message="disName">` `<target>` `<super/>` `</target>` `<arguments>` `</arguments>` `<send>`

Figure. 12: Java source and corresponding JavaML for SKD

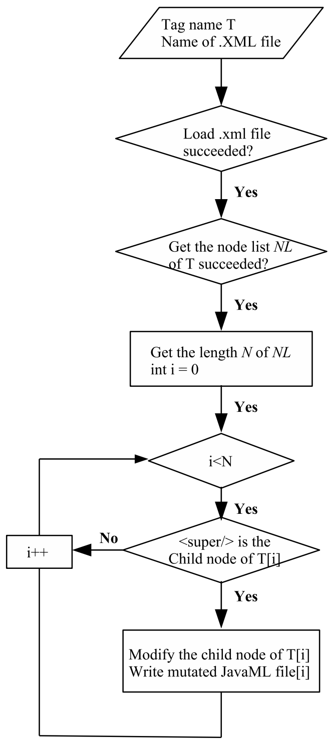
Tag name T
Name of .XML file
Load .xml file succeeded?
Yes
Get the node list NL of T succeeded?
Yes
Get the length N of NL
int i = 0
Yes
i<N
Yes
<super/> is the Child node of T[i]
No
i++
Yes
Modify the child node of T[i]
Write mutated JavaML file[i]

Figure. 13: Flowchart of **SKD**

```
if (nod.getFirstChild().getNodeName() == ”super”) {
   Element newNode = doc.createElement(”var−ref”);
   newNode.setAttribute(”name”,attrName);
   nod_parent.removeChild(nod);
   nod_parent.appendChild(newNode);
}
```

3. If the parent tag is `<target>`, this node will be deleted.

```
if (nod.getFirstChild().getNodeName() == ”super”) {
   nod_parent.removeChild(nod);
}
```

C. Implementing SMD Operator

Mutation operator **SMD**(Static Modifier Deletion) is designed to test the inheritance feature in OO programming language. It removes the static modifier.
The static modifier is referencing variable that is defined at the immediate parent class object. In Java, `static` is used to modify methods, variables, blocks and nested classes.

1. **Static method:** Static methods are the specific methods that do not need to access the state of object or only use static fields. Original code and its corresponding JavaML code is as follow:

Original code	JavaML
`static void g() { }`	`<method name="go"` `static="true" id="...">`

Since the mutant has syntax error, there is no need to use the `static` modifier on `main` method. The method beside the `main` method would be implemented by three steps:

- **Step1:** determine if the method is `main` method by checking the name of the method, go to step2 if the method is not `main` method.

- **Step2:** determine if the value of attribute `static` in the node tag `<method>` is ‘ true ’;
- **Step3:** delete the attribute `static` if its value is true.

2. **Static variable:** Static variable in Java is associated with the class rather than the objects of that class. Once the class is initiated, the storage location of the static variable is determined.

Original code	JavaML
`static int i`	`<field name="i" static="true">`

The static modifier is represented as attribute ’`static=true`’ in node tag `<field>`, thus the implementation could be realized by two steps:

- **Step1:** determine if the value of attribute ’static’ in the node tag `<field>` is ’true’;
- **Step2:** delete the attribute ’static’ if its value is true.

3. **Static block:** Static block in a class is a block that will be initiated when and only when the class is instanced.

Original Code	JavaML
	`<staic-initializer>`
`static {`	`<block>`
`System.out.println`	...
`"Value="+i);}`	`<block>`
	`</staic-initializer>`

Since deleting the static modifier of block does not cause any difference semantically, the mutant is regarded as equivalent mutant and need not to be implemented.

4. **Static Nested Block:** A static nested class is one which doesn’t implicitly have a reference to an instance of the outer class.

Original Code	JavaML
`public static`	`<class name="InnercCls"`
`class InnerCls{`	`<visible="public">`
`InnerCls() {}`	`static="ture">`
`}`	...

The implementation could be realized by two steps:

- **Step1:** determine if the value of attribute ‘ static ’ in the node tag `<class>` is ’true’;
- **Step2:** delete the attribute `static` if its value is true.

As analyzed above, mutation operator **SMD** is implemented following the steps in Figure 14.
DOM is applied to access the node and attribute in XML document, the core source code of this implementation is as follow.

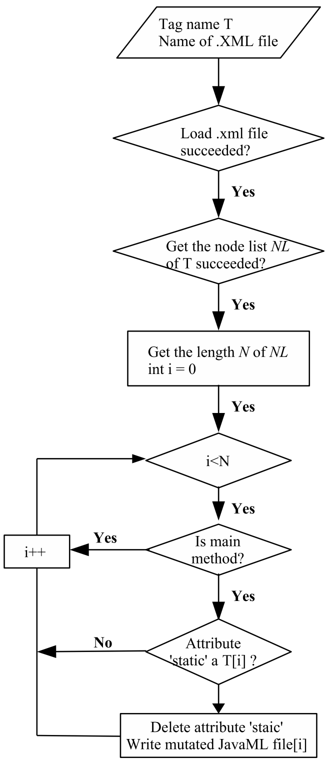

Figure. 14: Flowchart of **SMD**

- Following code shows the source code of ’`hasNode`’ method. This method checks the existence of input node tag. This method gets the node list and returns ‘ true ’ if the node exists.

```
public boolean hasNode(String NodeName) {
        this.nodeName = NodeName;
        nl = doc.getElementsByTagName(nodeName);
        numOfNode = nl.getLength();
        if (numOfNode != 0) {
                hasNode = true;
        }
        return hasNode;
}
```

- Next is to check whether the method is a `main` method or not. Following code conducts this task.

```
public boolean isMain(Node nod){
        Element ele = (Element) nod;
        boolean ismain = false;
        if(ele.hasAttribute(”name”)){
                ele.getAttribute(”name”);
                String tmpName = ele.getAttribute(”name
                        ”);
                if(tmpName.equals(”main”)){
                        ismain = true;
                }
        }
        return ismain;
}
```

- Below lists the detail of ’`hasAttr`’ method, this method checks the input node tag to check if it contains the input attribute.

```
public boolean hasAttr(String attrName, Node nod) {
        Element ele = (Element) nod;
        this.hasAttr = ele.hasAttribute(attrName);
        return hasAttr;
}
```

```
loadXML xml = new loadXML(fileName);
if (xml.hasNode(node)) {
        cntNode = xml.numOfNode;
        Node nod;
        cntAttr = 0;
        for (int i = 0; i < cntNode; i++) {
                nod = xml.nl.item(i);
                if(xml.hasAttr("static", nod)){
                        cntAttr++;
                        xml.setexistingplc(node);
                        if (xml.isMain(nod)) {
                                cntAttr−−;
                                continue;
                        }
                        else
                                xml.removeAttr("static", nod);
                }
        }
}
```

Figure. 15: Essential part of **SMD** implementation

- Finally, following `'remover'` method deletes the input attribute of the input node tag.

```
public void removeAttr(String attrName, Node nod) {
        if (hasAttr(attrName, nod)) {
                Element ele = (Element) nod;
                ele.removeAttribute(attrName);
        }
}
```

Figure 15 is an essential part of implementing **SMD** operator.

D. Implementing TKD Operator

The mutation operator **TKD**(”`this` keyword deletion) is designed to test the Java-specified feature. It deletes `this` keyword anywhere it occurs in the program. This operator checks if the member variables or methods are used correctly by replacing occurrences of `this.X` with `X`. The generated mutant could be killed by test case that shows different result when the member variables or methods are replaced by a method parameter.
The keyword `this` is referencing variable that is defined at the current class. There are two usages of `this` keyword:

1. (a) The keyword `this` is used to refer current class variable and/or instance variable.

Super class	Subclass
`String name;` `public Parent() {` `name="test";}`	`String name;` `public Chind(` `String name1){` `this.name=name1;}`

The original Java source code under mutation analysis and its corresponding JavaML are as follows:

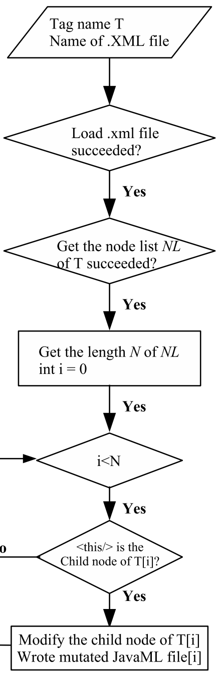
Tag name T
Name of .XML file
Load .xml file succeeded?
Yes
Get the node list NL of T succeeded?
Yes
Get the length N of NL
int i = 0
Yes
i<N
Yes
<this/> is the Child node of T[i]?
No
i++
Yes
Modify the child node of T[i]
Wrote mutated JavaML file[i]

Figure. 16: Flowchart of **TKD**

Original code	Coressponding JavaML
`"access"+this.name`	`...` `<binary-expr op="+">` `<field-access` `field="name">` `<this/>` `</field-access>` `<literal-string` `value="access"/>` `</binary-expr>` `...`

By converting the tags `<field-access>` and `<field-set>` into `<var-ref>` and `<var-set>`, we get the mutated JavaML document.

2. The keyword `this` is used to invoke current class method.
   The original Java source code under mutation analysis and its corresponding JavaML are as follows:

Original code	Corresponding JavaML
`this.detail();`	`<send meesage="detail">` `<target>` `<this>` `</target>` `<arguments>` `</arguments>` `</send>`

In this case, by deleting the node tag `<target>` directly, we would get the mutated JavaML document.

As analyzed above, mutation operator **TKD** is implemented following the steps in Figure 16.
Following codes show the program of checking and modifying the child nod of the parent node tag. If the parent tag is

`<field-set>`, the following codes(a) will delete its child node `<super>` and append new child node `<var-set>`; if the parent tag is `<field-ref>`, the following codes(b) will delete its child node `<this>` and append new child node `<var-ref>`; and if the parent tag is `<target>`, this node will be deleted.

(a) Check and modify child node `<field-set>`

```
if (nod.getFirstChild().getNodeName() == "this") {
   Element newNode = doc.createElement("var−set");
   newNode.setAttribute("name",attrName);
   nod_parent. removeChild (nod);
   nod_parent.appendChild(newNode);
}
```

(b) Check and modify child node `<field-ref>`

```
if (nod.getFirstChild().getNodeName() == "this") {
   Element newNode = doc.createElement("var−ref");
   newNode.setAttribute("name",attrName);
   nod_parent.removeChild(nod);
   nod_parent.appendChild(newNode);
}
```

(c) Check and modify child node `<target>`

```
if (nod.getFirstChild().getNodeName() == "this") {
   nod_parent.removeChild(nod);
}
```

E. Implementing PCD Operator

The mutation operator **PCD**(Parent Constructor Deletion) is designed to test the inheritance feature of Java. This operator deletes the calling to the parent class constructor. This causes the default constructor of the parent class to be called. The generated mutant could be killed by a test case for which the parent default constructor creates an incorrect initial state.

Original code	Mutant

`Class Child extends`
`  Parents {`
` Child(int a) {`
`  super(a);`
` }`
`}`
	`Class Child extends`
	`  Parents {`
	` Child(int a) {`
	` }`
	`}`

Parent constructor is invoked by using "`super()`" method in Java. Thus by deleting the parent constructor `super()`, we can implement this operator. The Java source code of parent constructor and its corresponding JavaML code are follow.

Source code	Corresponding JavaML

`super("");`	`<super-call>`
	` <arguments>`
	`  <literal-string`
	`   value=""/>`
	` <arguments>`
	`<super-call>`

By deleting the node `<super-call>`, we can get the mutated JavaML document.

Based on this analysis shown above, mutation operator **PCD** is implemented following the steps in Figure 17.

Following code is a detail of `removeTag` method, which deletes an assigned node tag `<super-call>` from its node list.

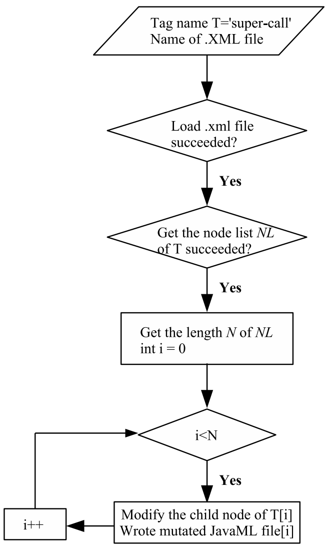
Tag name T='super-call'
Name of .XML file
Load .xml file succeeded?
Yes
Get the node list *NL* of T succeeded?
Yes
Get the length *N* of *NL*
int i = 0
i<N
Yes
Modify the child node of T[i]
Wrote mutated JavaML file[i]
i++

Figure. 17: Flowchart of **PCD**

```
public void removeTag() {
   for ( int i = numOfTag−1;i>=0;i−−) {
      Node nod=doc.getElementsByTagName(tagName).item(i);
      Node nod_parent = nod.getParentNode();
      Node nod_child = nod;
      nod_parent.removeChild(nod);
      setOutputFileName(i);
      writeXML();
      nod_parent.appendChild(nod_child);
   }
}
```

F. Implementing OMD Operator

The mutation operator **OMD**(Oveload Method Deletion) is designed to test the polymorphism feature of Java. This operator works by deleting each of the overloading methods one by one.

Original code	Corresponding JavaML

`public void details(){}`
`public void details(int n,`
` String st)){}`
	`//public void details(){}`
	`public void details(int n,`
	` String st) {}`

Overloading means that the same method name can be defined with different signature (the list of formal parameters or their types is different):

In JavaML, all methods are represented in node tag `<method>`. The attribute name in node tag `<method>` is same only when the methods are overloaded. Thus we separate the implementation of this operator into two steps:

- **Step1:** determine if a the node tag `<method>` is a overload method by detecting the existence of node tag `<method>` with the same value of attribute name,
- **Step2:** delete the node tag `<method>` if it is an overload method.

Based on the above analysis, the OMD operator is implemented based on the flowchart shown in Figure 18.

Following is a core part of this mutation operator implementation.

```
if (ele.hasAttribute("name")) {
   nameRecord = ele.getAttribute("name");
```

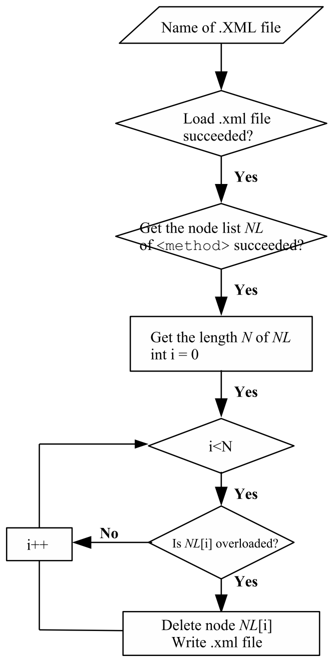
Name of .XML file
Load .xml file succeeded?
Yes
Get the node list NL of <method> succeeded?
Yes
Get the length N of NL int i = 0
Yes
i<N
Yes
Is NL[i] overloaded?
No
i++
Yes
Delete node NL[i] Write .xml file

Figure. 18: Flowchart of **OMD**

```
    int n = 0;
    System.out.println(”Method name is: ”+nameRecord);
    for (number = 0; number < length; number++) {
    if (number != checkPoint) {
        nodComp = nl.item(number);
        eleComp = (Element) nodComp;
        if (eleComp.hasAttribute(”name”)) {
            String tmpComp = eleComp.getAttribute(”name”);
            if (nameRecord.equals(tmpComp)) {
                isOverload = true;
                n++;}
            }
        }
    }
}
```

VII. Conclusion and Future Works

This article proposes a novel approach to implement mutation operators for object-oriented programming language. This approach is based on the technology of JavaML and DOM. JavaML document works as the intermediate representation to implement mutation operators and DOM is a tool to processing XML document.
The approach proposed in article first converts the original program into JavaML document. Then it implements mutation operator on JavaML document and gets the mutated JavaML document. And finally it converts the mutated JavaML document into mutant program. Six object-oriented mutation operators are designed for Java programs, and implemented correctly in the prototype. The mutated JavaML documents are generated. This method makes mutation analysis easier to be processed and has portability on the implementation of any mutation operator.
Future research requires for implementing other O-O mutation operators. It would be also important to get the evaluations of the mutants generated by each mutation operator: the percentage of dead mutants, still-born mutants, nonequivalent mutants and equivalent. By evaluating the mutants, mutation analysis improves the efficiency and quality of each mutation operator.

References

[1] A. Bertolino, “Software testing research: Achievements, challenges, dreams,” in *Proceedings of Future of Software Engineering (FOSE)*. IEEE Computer Society, 2007, pp. 85–103.

[2] A. James, H. Lionel, C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?” in *Proceedings of ICSE 2005*, 2005, pp. 402–411.

[3] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.

[4] D. Flanagan, *Java in a Nutshell*. O’Reilly Media, Inc, 2005.

[5] G. J. Badros, “JavaML: a markup language for Java source code,” *Computer Networks*, vol. 33, no. 1, pp. 159–177, 2000.

[6] R. H. Elliotte, *Processing XML with Java: A Guide to SAX, DOM, JDOM, JAXP, and TrAX*. Addison-Wesley Professional, 2002,

[7] M. Hall and L. Brown, *Core Web Programming*. Prentice Hall, 2001.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Hints on test data selection: Help for the Practicing Programmer,” *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[9] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, “The class-level mutants of MuJava,” in *Proceedings of the 2006 international workshop on Automation of software test*, 2006, pp. 78–84.

[10] Y.-S. Ma, Y.-R. Kwon and J. Offutt, “Inter-class mutation operators for Java,” in *Proceedings of the 13th International Symposium on Software Reliability Engineering*, 2002, pp. 352–363.

[11] H. Aman, “JJmlt-a Java-JavaML convertor,” http://se.cite.ehime-u.ac.jp/tool/JJmlt/.

Author Biographies

Qianqian Wang was born in Datong City, Shanxi Province, China in 1988. She received her BEng from XiDian University in 2011 and double MEng from Xidian University and Doshisha University in 2014 respectively. During her master period, she did research on image processing and software testing. She joined IBM (China) as a software test specialist after graduation. Currently she is working on storage testing.

Hirohide Haga was born in Kyoto, Japan in 1954. He received his BEng, MEng from Doshisha University and Ph.D. in Computer Science from Kyoto University in 1978, 1980, and 1995 respectively. In 1980 he joined to Hitachi, Ltd, and moved to Doshisha University in 1994. In 2001, he was a visiting scientist of the University of Oulu, Finland. From 2004 to 2005, he was a visiting professor of Cambridge University, UK. In 2011 and 2013, he was an invited professor of Ecole Centrale de Lille, France.

Currently he is a professor of the Faculty of Science and Engineering, Doshisha University, Kyoto, Japan. His research interests include software engineering, multi-agent computing, and digital gaming. He is a member of IEEE, ACM, BCS (British Computer Society), IPSJ (Information Processing Society of Japan), and IEICE (Institute of Electronics, Information and Communication Engineers, Japan).