Symbiosis

www.symbiosisonline.org
www.symbiosisonlinepublishing.com

Research article Journal of Computer Science Applications and Information Technology Open Access

Formalization of Mutation Operators

Pranshu Gupta*

Assistant Professor, Department of Mathematics and Computer Science, DeSales University, Center Valley, PA 18034, USA

Received: January 12, 2018; Accepted: January 25, 2018; Published: January 29, 2018

***Corresponding author:** *Pranshu Gupta, PhD, Assistant Professor, Department of Mathematics and Computer Science, Dooling Hall 222E, De-Sales University, 2755 Station Avenue, Center Valley, PA 18034, USA, Tel: (610) 282-1100 x2854; Email: Pranshu.Gupta@desales.edu*

Abstract

Software testing is a significant phase in any software development lifecycle irrespective of the type of software being developed. The main goal of software testing phase is to minimize the software faults in a system and increase its reliability. A software fault is an unintended mistake that causes failure of the system or any system component. Therefore, it is vital that the system is tested for most faults. One of the popular approaches to achieve a fault-free system is to induce, test and remove the faults from the system, commonly known as Mutation Testing. It uses mutation operators to induce and test faults in program. Furthermore, a software fault type (a textual description of a specific kind of fault that can occur in any program) encompasses one or more mutation operators. As the mutation operators induce a specific fault in the program, there is a need for a formal definition for each mutation operator representing a precise software fault that falls under an explicit software fault type. Literature research shows the lack of formalization of mutation operator definitions and as a consequence, it becomes difficult to induce a precise fault in a program. In this paper, the author illustrates a formal methodology to define mutation operator to induce a specific faults in programs written using object-oriented programming languages. The formalization of operators shows the need for new mutation operators that have not been defined for object-oriented programs.

Keywords: Mutation operator; Production rule; Software testing; Mutation Testing;

Introduction

A software fault is an unintended mistake that should be removed before the system is deployed in the working environment. The business and mission-critical software, e.g. airplane software or medical robotics software, if not tested completely and thoroughly may lead to undesirable events. In order to deploy a fault-free and most reliable system, the first step of mutation testing is to induce software faults in the system that represents commonly occurring errors. If these faults can be induced in a system, the program can be tested for unwanted behavior and subsequently the mistake can be removed from the program. Mutation Testing uses mutation operators to induce faults in a program. Mutation operators represent commonly occurring errors in the system and are tested for success or failure. If the program output changes by inducing the mutation operator, the test is a success otherwise if the output remains the same then the test is considered a failure because after inducing the fault the output should have changed to show the incorrect behavior of the system.

Before understanding mutation operators in detail, it is important to understand the concept of software fault type. A software fault type is a textual description of a specific kind of fault that can occur in any program. A fault type can have many instances in the program depending on where and what change is made. For example, "changing arithmetic operator" is a software fault type that can exist in both constructor and a method of a program, creating two distinct faults in a program. Each software fault type can be represented by one or more mutation operators. Each mutation operators induces a specific software fault in the program. Furthermore, a single mutation operator can induce multiple instances of a fault in a program and in consequence inducing multiple instances of the software fault type. A mutation operator definition describes how to insert an instance of a specific fault type in a program. Multiple faulty versions of the programs, namely "mutant" programs, are generated based on where and what changes are made in a program using the mutation operator. A mutation operator may not necessarily generate all instances of a software fault type. As the mutation operator should induce a specific fault in the program, there is a need for a formal definition that precisely defines the changes that will be made in the program by the mutation operator representing a specific software fault.

Literature research shows the lack of formalization of mutation operator definitions. It shows that mutation operators usually have been defined using a sample code or text only. In this paper, the author presents a grammar to formalize the mutation operator definitions that can be applied to a program to precisely induce a particular fault and analyze the behavior of the program

Mutation Operators

Formerly, mutation operators were created for non-OO fault types but as the concept of OO programming was introduced the same idea was applied to OO software fault types [5]. Traditional faults or non-OO faults are represented by mutation operators defined for a non-OO programming language such as C [1]. A few examples of the traditional fault types are a missing statement, incorrect arithmetic operations, incorrect Boolean expressions, and incorrect variable [1,10]. Examples of traditional mutation operators that model these fault types are ABS (Absolute value insertion), AOR (Arithmetic operator replacement), LCR (Logical

 * ***Corresponding author email:*** *Pranshu.Gupta@desales.edu*

connector replacement), ROR (Relational operator replacement), and UOI (Unary operator insertion). Traditional mutation operators can be used to induce fault instances in both non-OO and OO programs [1,10,15,20]. Traditional mutation operators are still valid for OO programs, but due to the properties of the OO programming, additional operators have been defined for the OO programs that represent faults specific to OO programming properties [9,14,19,21].

Kim et al., Ma et al., and Derezińska defined mutation operators for OO programs [6,11,13,14]. Specifically, Kim et al. introduced a Class Mutation method that targeted fault types that could occur in OO programs [11]. This research categorized fault types based on polymorphic types, method overloading, method overriding, field variable hiding, information hiding, static/dynamic stated of objects and exception handling. Further, Ma et al. defined more mutation operators for OO programs and evaluated mutation testing for OO programs using empirical studies [13,14]. Research by Ma et al. defines mutation operators specifically for Java language, but a majority of operators can be applied to other OO programming languages [13,14]. Offutt et al. discussed the faults and mutation operators based on OO properties, such as polymorphism and Inheritance [16,17].

Fault Types

Before inducing faults in a program, it is important to understand the fault in detail and its categorization. For example, changing arithmetic operators is a fault type that encompasses many specific faults such as changing subtraction to addition, addition to subtraction, multiplication to division and more. Hayes defined the fault taxonomy for OO systems by analyzing fault types and conventional methods applied to these fault types [9]. His research used fault types from Purchase & Winder, Firesmith and Offutt et al. [7,16,18]. Firesmith and Offutt et al. list OO fault types based on various OO program features such as class, attributes and methods as well as OO program properties such as inheritance, polymorphism, and method overloading [2,7,16]. Walkinshaw et al. mention a number of OO fault types collected from various sources [2,4,9,19]. Walkinshaw et al. state that little investigation has been done to find specifics of fault types likely to occur in OO projects; therefore, the development of testing techniques for OO software is difficult [19]. Lin et al. analyzed fault types related to the polymorphic property of OO programming. The goal of Lin et al. research was to provide a realization for OO fault types in polymorphism and also discuss specific categories of inheritance and polymorphism fault types [12]. In this paper, the focus is on OO fault types that are collected from various studies [7,16]. OO fault types already considered for the creation of a mutation operator from Chevalley and Firesmith are incorrect overloading methods implementation, super keyword misuse, this keyword misuse and faults from programming experience [7,21]. Other OO fault types considered by Offutt et al. are state visibility anomaly, state definition inconsistency, state definition anomaly, indirect inconsistent state definition, anomalous construction behavior; incomplete construction and inconsistent type use [16,17]. Table 1 shows fault types collected from literature and mapped to existing mutation operators [3,21,6,7,11,13,16,17]. The analysis of fault types and testing methods associated with these fault types is crucial.

Grammar

The goal of this paper is to formalize the definition of mutation operators such that faults can be precisely induced in the program. This section shows the grammar that was created to formalize the definition of mutation operators. A grammar is defined here for the framework known as the Mutation Operator Production Rule System (MOPRS). The grammar includes formally designed production rules (IF-THEN) using the keywords and operators that help in defining a precise mutation operator to be applied to the program. This mutant program represents a particular software fault in the program. The grammar consists of following components:

a) A finite set of production rules – These production rules are used to create a formal specification for the mutation operators that can be applied to a program to generate mutants for a specific fault. The symbol PR represents the production rule for the mutation operator. Figure 1 shows the grammar production rules with start symbol S.

b) A finite set of nonterminal symbols – The start symbol in this grammar is PR with nonterminal symbols A, B, C, E, F, G, H, K, L, and S1.

c) A finite set of terminal symbols – The keywords and the operators are the terminal symbols in this grammar. The language-specific keywords are also included in this set of terminal symbols. D, I, J, OP, OP1 and Z are terminal symbols.

Formalization of Mutation Operators

In this paper, the author has created new mutation operators as well as improved upon the definitions of the existing object-oriented mutation operators. This section presents the precise definition of the mutation operator using the MOPRS. Using this system, first, a precise definition of the mutation operator is formed in words that serve as formal text definition. Based on the precise definition a formal production rule is created that can be applied to the program. The next step is to create mutant programs by applying this production rule using the grammar (defined in the previous section) along with the specific components from the program such as methods, punctuation, classes and more. The mutant examples mentioned in the next section show the grammar keywords and the specific program components.

For this experiment, two programs were used – Geometric Objects and Banking. Geometric Objects program calculated the perimeter and area of various geometric objects and banking created different types of accounts and defines operations performed on these accounts. Even though these programs are simple but they addressed the properties of the object-oriented programming languages. These programs were written using Java programming language and therefore Java mutant programs were created but the MOPRS is generic and can be applied to any programming language. As a part of the research, a program was written to read these specifications of a mutation operator from

```
S -> PR1 | PR2 | PR3 | PR4
PR1 -> IF MATCH A THEN B
PR2 -> IF MATCHINCOMMENT E THEN B
PR3 -> IF MATCH K THEN H
PR4 -> IF MATCH L THEN F
S1 -> "' "(" VAR OP EXPRESSION'" ")" D
A -> 'assert S1' D
A -> "' "(" VAR "I" VAR OP VAR'" ")" D
A -> 'if S1' D
B -> FIND VAR "I" EXPRESSION1 "Z" C
B -> PARENT CLASSNAME FIND VAR "I" EXPRESSION1 "Z" C
B -> ASSOCIATEDMATCH CLASSNAME ASSOCIATEDFIND CLASSNAME FIND VAR "I"
EXPRESSION1 C
C -> INSERTAFTER VAR "I" EXPRESSION OP1 NUM "Z" D
C -> INSERTAFTER VAR "I" EXPRESSION "Z" D
E -> 'invariant S1' "Z" D
C -> CHILD CHILDNAME INSERTAFTER VAR I EXPRESSION OP1 NUM D
C -> ASSOCIATEDFIND CLASSNAME INSERTAFTER VAR I EXPRESSION OP1 NUM D
K -> "'EXPRESSION "J" VAR'" D
K -> EXPRESSION "J" METHODCALL "("")" D
L -> "'METHODNAME (PARAMETER) "{" METHODBODY '""}" D
L -> "'METHODNAME ("PARAMETERLIST") "{" METHODBODY '""}" D
F -> FIND "{" METHODBODY "}" G
G -> REPLACE METHODNAME"("")" D
G -> REPLACE METHODNAME PARAMETERLIST[N] D
H -> REPLACE EXPRESSION "J" VAR1 D
H -> REPLACE EXPRESSION "J" METHODCALL"("")" "Z" D
I -> =
J -> .
OP -> >
OP -> <
OP -> <=
OP -> >=
OP -> ==
OP1 -> -
OP1 -> +
OP1 -> *
OP1 -> /
Z -> ;
D -> ε
```

Figure 1: MOPRS Grammar

a text file and auto-generate the mutants in the program. This step was completed in order to test the feasibility of the system because each mutation operator generates multiple mutant programs.

Example 1: MIP2 (Method Incorrectly Performed)

This mutation operator is an improved specification of an existing the mutation operator known as OMR (Overloading method contents change).

Text definition: If there is an overloaded method with some parameters, then change the contents of the original method to the nth overloaded method call.

Production Rule: If there are N number of overloaded methods with some parameters, x(y, z), x(y), x (z), Then change the contents of x(y, z) to a method call for x(y) and x (z) one at a time.

Mutant 1

```
IF MATCH: "`METHODNAME ("PARAMETERLIST") "{" METHODBODY '"}""
METHODNAME: findPerimeter
THEN
FIND: "{" METHODBODY "}"
INSERTAFTER: NULL
INSERTBEFORE: NULL
REPLACEPARAMETER: PARAMETERLIST[0]
REPLACE: METHODNAME "(" PARAMETERLIST[0] ")" ";"
```

Mutant 2

```
IF MATCH: "`METHODNAME ("PARAMETERLIST") "{" METHODBODY '"}""
METHODNAME: findPerimeter
THEN
FIND: "{" METHODBODY "}"
INSERTAFTER: NULL
INSERTBEFORE: NULL
REPLACEPARAMETER: PARAMETERLIST[1]
REPLACE: METHODNAME "(" PARAMETERLIST[1] ")" ";"
```

These mutants are created using the grammar production rules shown in figure 1 when applied to the program. This not a complete list of mutants, more are possible using the same production rule. There is a possibility of multiple instances of the same mutant in the program.

Example 2: MIP4 (Method Incorrectly Performed)

This is an example of a new mutation operator in this category. As mentioned in the previous section, literature research shows that all faults have not been addressed by the mutation operators. Thus, this research also focused on generating new mutation operators for fault types.

Text definition: If there is another method call inside the body of the method, then change the call to another compatible method.

Production Rule: If there is a method call x() inside the body of method z. Then change the method call x() to y() in the body of the method z, where x and y are compatible.

Mutant 1

```
IF MATCH: EXPRESSION "." METHODCALL "("")"
THEN
FIND: NULL
INSERTAFTER: NULL
INSERTBEFORE: NULL
REPLACEMETHOD: METHODCALL1
REPLACE: EXPRESSION "." METHODCALL1"("")" ";"
```

A new mutation operator was created that does not exist in the current list of object-oriented mutation operators. Similar to the last example a formal production rule was defined and then applied to the program using the grammar shown in figure 1. As mentioned before, more mutants are possible for these operators and also multiple instances of the same mutant are possible.

Discussion

The previous section shows the formalization of the mutation operator. The grammar is applied to the program to create mutants. These mutants were tested and the erroneous behavior of the program helped in finding faults in the program. If the mutant was killed that meant that a specific fault encompassed by a specific fault type was found and could then be corrected, further increasing the reliability of the software. The fault type, method incorrectly performed, contains multiple mutation operators two of which are shown in the previous section. These two mutation operators specify two different faults. When forming this grammar, it was found that all faults are not represented by the existing set of mutation operators. The second example in the previous section shows the production rule for a newly created mutation operator.

This research created mutation operators for fault types such as assignment that violates a state invariant (IVS) and Method Incorrectly Performed (MIP). 11 mutation operators were specified in the first category and 4 for the second type. Also, a test suite was created for testing the programs induced with these mutation operators [8]. All the mutants for both fault types were killed by the test suite.

Conclusions

Both new and improved mutation operators were created for IVS and MIP fault types. The former fault type has not been addressed by any of the existing mutation operators and contained all new mutation operators for specific faults in this category. The other fault type (MIP) includes new mutation operators as well as improved specifications for existing mutation operator that is encompassed in this fault type. This particular fault type is related to the method overloading property of object-oriented programming.

All mutation operators were induced and tested by the test suite and the suite was successful in finding the faults in the program Thus, we can say that the new and the improved mutation operators were effective in creating mutants that were killed i.e. found by the test suite and are a good representation of faults that can be present in a program for that fault type. There is still a need to address other fault types listed in table 1. There is need for new mutation operators that represent specific faults. The future work includes creating new mutation operators for the existing fault types and also creating new fault types that are missing from the existing collection.

Table 1: Object-Oriented Fault Types vs. Existing Mutation Operators

Fault types associated with Classes	**Related Mutation Operators**
Incorrect initialization or missing re-initialization	JDC (Java-supported default constructor create)
Fault types associated with Attributes	**Related Mutation Operators**
Incorrect values of attributes	ISK (super keyword deletion)
Not initialized	JID (Member variable initialization) deletion)
Incorrect visibility, access modifier misuse	AMC (Access modifier change)
Fault types associated with Methods	**Related Mutation Operators**
Incorrect method used	EAM (Accessor method change), EMM (Modifier method change)
Parameter mismatch	OAO (Argument order change)
Fault types Associated with Inheritance	**Related Mutation Operators**
Deletion of resource in subclass violates abstraction of superclass	IHD (Hiding variable deletion), IHI (Hiding variable insertion)
Incompatibility between subclass resources and existing superclass resources.	IOP (Overridden method calling position change)
Incorrect initialization (super not used for initialization)	IPC (Explicit call of a parent's constructor deletion)
Inheritance allows super class features with the same names, providing the opportunity for misnaming. Changes to either superclass will affect the subclass.	IOR (Overridden method rename)
Original resource in super class not overridden in subclass	IOD (Overriding method deletion)
super keyword misuse	ISK (super keyword deletion)
Fault types associated with Polymorphism	**Related Mutation Operators**
Failure to check the compatibility of the actual parameters	PPD (Parameter variable declaration with child class type)
Failures associated with instantiation	PNC (new method call with child class type), PMD (new method call with child class type)
Fault types associated with Method	**Related Mutation Operators**
Misusing a dynamically bound method	OMD (Overloading method deletion)
Message not received by correct method or message received by incorrect method	OMD (Overloading method deletion), OAO (Argument order change), OAN (Argument number change)
Method incorrectly performed	OMR (Overloading method contents change)
Fault types associated with Inheritance manifested by polymorphism	**Related Mutation Operators**
State visibility anomaly	SVA (State visibility anomaly)
State definition inconsistency due to state variable hiding	SDIH (State definition inconsistency hiding)
State definition anomaly	SDA (possible post-condition violation)
State defined incorrectly	SDI (State defined incorrectly)
Inconsistent type use	ITU (Inconsistent type use)
Incomplete construction	IC (Incomplete construction)
Indirect inconsistent state definition	IISD (Indirect inconsistent state definition)
Anomalous construction behavior	ACB (Anomalous construction behavior)
Fault types associated with programming language specific features	**Related Mutation Operators**
this keyword misuse	JTD
Static modifier misuse	JSC

References

1. Agrawal H, DeMillo RA, Hathaway B, William H, Wynne H, Krauser EW, et al, Design of Mutant Operators for the C Programming Language. SERCTR41-P. 1989.
2. Alexander RT, Offutt J, Bieman JM. Syntactic fault patterns in Object-oriented programs. Proceedings of the Eighth IEEE International Conference on Engineering of Complex Computer Systems. 2002:193-202.
3. Binder RV. Testing Object-oriented Software: A Survey. Journal of Software Testing, Verification and Reliability. 1996;6:125-252.
4. Binder RV. Testing Object-oriented Systems: models, patterns, and tools. Addison Wesley. 1999.
5. DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: help for the practicing programmer. Computer. 1978;11(4):34-41.
6. Derezińska A. Advanced mutation operators applicable in C# programs. Software Engineering Techniques: Design for Quality. 2006:283-288.
7. Firesmith DG. Testing Object-oriented Software. Proceedings of the Eleventh International Conference on Technology of Object-oriented Languages and Systems. 1993:407-426.
8. Gupta P, Gustafson DA. Object-oriented Testing beyond Statement Coverage. Proceedings of International Conference on Computer Applications in Industry and Engineering, 2010:150-155.
9. Hayes HJ. Testing of Object-oriented Programming Systems (OOPS): A Fault-Based Approach. Proceedings of Object-oriented Methodologies and Systems. 1994:205-220.
10. Jia Y, Harman M. MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. Proceedings of Third Testing: Academic and Industrial Conference Practice and Research Techniques. 2008:94-98.
11. Kim S, Clark JA, McDermid JA. Class mutation: Mutation testing for Object-oriented programs. Proceedings of Net Object days Conference on Object-oriented Software Systems. 2000.
12. Lin JC, Huang YL, Liu CH. Testability Analysis for Polymorphism. Proceedings of the Second IASTED International Multi-Conference on Automation, Control, and Information Technology. 2005:98-103.
13. Ma YS, Kwon YR, Offutt J. Inter-class mutation operators for Java. Proceedings of the 13th International Symposium on Software Reliability Engineering. 2002:352-363.
14. Ma YS, Harrold MJ, Kwon YR. Evaluation of Mutation Testing for Object-oriented programs. Proceedings of the 28th international conference on Software engineering, 2006:869-872.
15. Offutt JA, Lee A, Rothermal G, Untch RH, Zapf C. An experimental determination of sufficient mutant operators. ACM Transactions of Software Engineering Methodology. 1996;5(2):99-118.
16. Offutt JA, Untch RH. Mutation 2000: Uniting the orthogonal. In Mutation testing for the new century. 2001:34-44.
17. Offutt JA, Alexander R, Wu Y, Xiao Q, Hutchinson C. A fault model for subtype Inheritance and Polymorphism. Proceedings of the 12th International Symposium on Software Reliability Engineering. 2001:84-93.
18. Purchase AJ, Winder RL. Debugging Tools for Object-oriented Programming. Journal of Object-oriented Programming, 1993;4(3):10-27.
19. Walkinshaw N, Roper M, Wood M. Collecting and Categorizing Faults in Object-oriented Code. Sheffield: UKTest. 2005.
20. Woodward MR. Mutation Testing - an Evolving Technique. Proceedings of IEE Colloquium on Software Testing for Critical Systems. 1990.
21. Chevalley P. Applying mutation analysis for Object-oriented programs using a reflective approach. Proceedings of the 8th Asia-Pacific Software Engineering Conference (APSEC). 2001.