Advanced mutation operators applicable in C# programs

Anna Derezińska

Institute of Computer Science, Warsaw University of Technology, Nowowiejska 15/19
00-665 Warsaw, Poland
A.Derezinska@ii.pw.edu.pl

Abstract. This paper is devoted to advanced mutation operators for C# source code. They deal with object-oriented (OO mutations) and other complex features of the code. They require structural information about a code, unlike the standard mutations. Applicability of OO operators in C# is compared with those for other OO languages. Operators for specific features of C# language are also proposed. The detailed specification of operators can be provided in terms of pre- and post-conditions of a program transformation. Based on the operators' specification, the generation of mutated C# programs can be automated.

1 Introduction

Mutation testing is a fault-based testing technique used for evaluating tests and for measuring the effectiveness of test cases [11]. Mutations are simple changes inserted into a source code. They are defined in terms of mutation operators in order to make the automated testing process. Standard (traditional) mutation operators can be easily specified for many languages, e.g. an operator replacing an arithmetic operator "+" with "-". Testing of new features in object-oriented languages require more complex operators. The changes, introduced by these operators, should be consistent, for instance, with the inheritance hierarchy of classes. These operators take into account information that is non-local to the placement of the change in the source code.

This paper is devoted to advanced mutation operators specialized for C# code. They can be more dependent on the programming language than the standard mutation operators. The known (from Java [3,7,10] and C++ [4]) object-oriented operators were revised and adopted for C#. Some of the operators have altered definition or different scope of application due to different constructs used in the C#. New operators for specific, not only object-oriented, features of C# were also proposed.

Mutation operators were usually defined informally and illustrated by code examples [3,7,10]. It is not sufficient for the precise definition of advanced mutation operators. To make a definition unambiguous an operator can be specified as a program transformation with pre- and post-conditions. This approach is presented in the paper. Precise specification of operators allows effectively generating mutated programs (so-called *mutants*) that could be successively compiled. The specification and the quality of selected operators were verified in experiments on functional and unit tests [5,6].

Please use the following format when citing this chapter:

Derezińska, A., 2006, in IFIP International Federation for Information Processing, Volume 227, Software Engineering Techniques: Design for Quality, ed. K. Sacha, (Boston: Springer), pp. 283–288.

2 Object-oriented mutation testing

In object-oriented programs standard mutation can be used for intra-method level testing. Object oriented languages provide also new constructions that are not tackled directly by standard mutation operators. The research on the OO mutation was done mostly on Java programs [1,3,7-10]. Mutation of object-oriented features of a UML class specification and C++ code was studied in [4]. To my best knowledge the only research on OO mutation in C# was performed by Baudry et al. [2]. They referred to standard mutation operators, invocation of an exception and only two OO operators. The OO operators were not studied in detail but announced in their Mutator tool. Other C# mutation tools (Nester) support so far only the standard mutation operators.

An important issue is determining the quality of mutation operators in order to choose the best ones to be applied [8]. A good operator should satisfy the following conditions: (1) reflect typical errors of program developers, (2) generate proper and non-equivalent mutants, (3) be effective in qualification of tests. An *equivalent mutant* gives for any input exactly the same output as the non-mutated program. The judgment about the equivalence is very effort-consuming. Some mutation operators can generate mutants that are killed very easily by any test. Such operators are not useful in qualification of tests, although they can mimic typical errors of developers. Although OO operators generate fewer mutants than standard operators [9], we would like to limit the number of mutants and choose the most appropriate operators for C#.

3 Advanced mutation operators for C#

The comprehensive set of mutation operators for C# language is presented in tab. 1. The relations for previously defined operators for Java [3,7,10] and/or for C++ [4] are indicated in the column "Ref". The differences between the applicability of the corresponding operators in different languages were examined [5]. Also eight new operators concerning the specific features of C# were defined. Different groups of operators are discussed below. For the brevity reasons, a description and a specification is given only for two exemplary operators, OPD and IOK.

An informal description of advanced operators is not sufficient for their precise specification. Therefore, any operator could be specified using pre- and post-conditions of the transformation of program *P* to a mutant P_{Oi}' (i-th mutant after applying operator *O* on *P*). The pre- and post-conditions are specified using logical predicates with quantifiers (exists ∃, for all ∀) and operators (and, not, or, xor, ⇔, ⇒). In post-conditions, the elements marked with the apostrophe (eg. *x*') relate to elements changed in the mutant P_{Oi}'. Different features of the elements are defined by Boolean values using the dot notation. For example:

x.class	True if *x* is a class
z.override	True if *z* has the modifier *override*
x.z.method	True if *z* is a method declared in *x* (or inherited by *x*)

The following expression denotes that *s* is a syntactically and semantically correct part of instruction *p* (is used in *p*): $s \otimes p$, or equivalently $"s" \otimes p$ for complex *s*. The full notation of the specification and specifications of operators are given in [5].

Table 1. Advanced mutation operators for C#

Operators		Inv	Spec	Appl	Ref
AMC	Access modifier change	-			[10,7,4]
IHD	Hiding variable deletion		-		[10,7]
IHI	Hiding variable insertion		-		[10,7]
IOD	Overriding method deletion		-		[10,7]
IOP	Overridden method calling position change		-		[10,7]
IOR	Overridden method rename	-	-	-	[10]
ISK	*Base* keyword deletion				[10]
IPC	Explicit call of a parent's constructor deletion		-		[10]
PNC	*New* method call with child class type		-		[10,7]
PMD	Member variable declaration with parent class type				[10,7]
PPD	Parameter variable declaration with child class type				[10,7]
PRV	Reference assignment with other compatible type				[10,4]
OMR	Overloading method contents change				[10]
OMD	Overloading method deletion			-	[10,7]
OAO	Argument order change			-	[10,7]
OAN	Argument number change			-	[10,7]
JTD	*This* keyword deletion				[10]
JSC	*Static* modifier change			-	[10,7]
JID	Member variable initialization deletion				[10]
JDC	C#-supported default constructor create				[10]
EOA	Reference assignment and content assignment replacement			-	[10,3]
EOC	Reference comparison and content comparison replacement			-	[10,3]
EAM	Accessor method change			-	[10,3,4]
EMM	Modifier method change			-	[10,3,4]
MNC	Method name change				[3,4]
MBC	Member changed				[4]
MCO	Member call from another object				[4]
MCI	Member call from another inherited class, MCR in [2]				[4,2]
RFI	Referencing fault insertion				[2]
EHR	Exception handler removal				[7]
EHC	Exception handling change	-			[7]
DMC	Delegated method change				
DMO	Delegated method order change				
DEH	Method delegated for event handling change				
PRM	Property replacement with member field				
IOK	*Override* keyword substitution				
OPD	Overriding property deletion				
OID	Overriding indexer deletion				
NDC	Namespace declaration change				

Inv - invalid operators (listed for compatibility reasons),
Spec - differences in specification to Java or C++,
Appl - differences in meaning or application scope to Java or C++

Several object-oriented inter-class mutation operators can be applied in the similar way in different languages. These operators refer mainly to usage of classes related by inheritance, e. g. PMD, PPD, PRV. Also the operators dealing with incorrect calling of methods are language-independent, e. g. MCO, MNC, MBC, MCI.

Some mutation operators are not appropriate for C# programs. This was stated by a code analysis and experiments [5,6]. These operators (AMC, IOR, EHC) are listed for compatibility reasons and indicated in the column “Inv" of the Table 1.

Other operators for C# have to be specified in a different way than the corresponding operators for C++ or Java (the column "Spec" of the table 1). The specification has to take into account new features of C#. For example, extended usage of keywords (*new* - in operators IHD, IHI), keywords newly introduced in C# (*override* - in operators IOD, IOP).

Regardless of an operator specification its application can be different in the considered languages (the column "Appl" in table 1). They can have a different meaning, or the scope of the application can be broader or narrower than that from Java or C++.

The JSC operator for C# deletes the *static* modifier for any member of a class. The reverse operation (adding *static* modifier as in the JSC operator for Java) could be omitted, because it provides non-compiled code in most cases.

The EOA operator replaces assignment of an object reference pointing to an object with the clone (duplicate) of this object. It intends to check a possible mismatch of objects and object references. In C# the overloaded *Clone()* method creates an object duplicate and is defined for many types. The EOA operator can be applied for a class which has its *Clone()* method.

The EOC operator replaces one kind of comparison with another one (== with *Equals()* or v.v.). In C# the default *Object.Equals* method calls *Object.ReferenceEquals* which results in a reference comparison instead of a value comparison. For many types *Equals* method is overloaded to implement value comparison. The user can overload *Equals* method and the operator == for own declared types.

Operators EAM and EMM dealing with accessor and modifier methods (*get* and *set*) have a minor significance in C# because a new element - *property* can be used.

New mutation operators for the specific features of C# were also defined. They are dealing with delegates, properties, indexers, override modifier and namespaces. *Properties* are values that can be stored or retrieved of a class using an accessor (*get*) and a modifier (*set*). Properties can be overriding in the similar way as methods do. The OPD operator deletes a whole definition of a property from the derived class, e. g.:

OPD: Original code

```
public class Figure
{         public virtual double  Area
          {         get
                    {         return 0;}
          }
}
public class Square : Figure
{         public override double Area
          {         get
                    { return Math.Ar(base. 2); }
          }
}
```

Mutated code

```
public class Figure
{      public virtual double  Area
       {         get
                 {         return 0;}
       }
}

       public class Square : Figure
       {         _______          }
```

The operator forces the usage of the appropriate property from the base class. It can be applicable only if the class does not inherit from an abstract type, otherwise a compilation error would be detected because the class does not implement inherited abstract member. A specification of the OPD operator is given below:

OPD Pre: $\exists_x$ x.class and $\exists_y$ y.class and y.x.public_inherited and and not x.abstract and $\exists_z$ (y.z.property and z.override)

OPD Post: not y'.z'.property

Indexers are used to index a class in the same way as an array. The OID operator deletes a whole definition of an indexer from the derived class. This operator can be defined in the similar way as the OPD operator.

In properties the *set* modifier uses an implicit parameter called *value*, whose type is of the property. By convention names of properties begin with capital letter, while names of fields with a small one. By mistake a property can be called instead of a field or vice versa. The PRM operator replaces those two names.

Omission of the keyword *override* in a method declaration is a common mistake of C# program developers, who have habits from C++ or Java. In C#, special keywords (*override, new*) denote override or hide of a method from the base class. The IOK operator substitutes an *override* occurrence with the *new* keyword, or vice versa (*new* with *override).* This substitution cannot be revealed by a compiler.

IOK: Original code

```
public class  Figure
{         public virtual void Draw()
              { ..... }
}
public class Square : Figure
{         public override void Draw()
              { .....          }
}
```

Mutated code

```
public class  Figure
{    public virtual void Draw()
              { ..... }
}
public class Square :  Figure
{         public new void Draw()
              { ..... }
}
```

This mutation will be detected if polymorphism is used. In the above example an object of *Square* can be referenced as a *Figure*. After mutation a method *Draw* called for this object will invoke a method from class Figure instead of class *Square*.

IOK Pre: $\exists_x$ x.class and $\exists_y$ y.class and y.x.public_inherited and $\exists_z$ (x.z.method and (z.override or z.new))

IOK Post: (z.override $\Rightarrow$ z'.new) and (z.new $\Rightarrow$ z'.override)

In C# *delegates* are the object-oriented equivalents of function pointers. However, unlike function pointers, delegates are type-safe and secure. Delegates can be used in callback and event-handling scenarios. The DMC operator changes a delegated method into another method visible in this context and taking the same types of parameters. The operator simulates a fault, when a developer used by mistake a different method as a callback. The DMO operator changes the order of assignment of delegated methods. The DEH operator changes a method delegated for the event handling. Simulated fault can be for example caused by misleading of elements during construction of a GUI. The operators on delegates are extensively studied in [6].

The *namespace* statement is used in C# to define a new namespace, which encapsulates the classes. The NDC operator changes a namespace declaration. It is used only if exists an appropriate declaration of the class in both namespaces.

4 Final Remarks

The object-oriented mutation operators adopted for C# programs and other advanced operators dealing with new programming features were studied. Defining a mutation operator as a program transformation with pre- and post-conditions allows to give a precise specification of the operator. It is especially important for complex operators dealing with structural features of a program. Based on provided specifications of operators a tool for mutation of C# programs is currently under development.

The application of selected mutation operators for C# was evaluated in experiments. They allowed verifying the specifications, comparing usefulness of operators and suitability for the test selection. The sets of functional tests and unit tests were used [5,6]. The preliminary results showed that object-oriented operators IHD, IHI, IOD, IOP, IOK, OPD and OMD generated proper, non-equivalent mutants and were selective in assessment of the quality of functional tests. Mutants generated by the PRM operator were non-equivalent but killed by all functional tests. Among the operators dealing with exception handling and delegates two operators EHR and DMC were the most promising ones. The evaluation of mutation operators for C# and comparison with other testing criteria needs still further experiments.

Acknowledgment This work was supported by the Polish State Committee for Scientific Research under the project 4 T 11 C 04925.

References

1. Alexander, R. T., Bieman, J. M., Ghosh, J. M., Bixia, J.: Mutation of Java objects, Proc of 13th Int. Symp. on Software Reliability Eng., (2002) 341-351
2. Baudry, B., Fleurey, F., Jezequel, J-M., Traon, Y. Le.: From genetic to bacteriological algorithms for mutation-based testing, Sof. Testing Verif. and Reliab., vol 15, no 2, (2005)
3. Chevalley, P.: Applying mutation analysis for object-oriented programs using a reflective approach, Proc of 8-th Asia-Pacific Softw. Engin. Conf., ASPEC (2001) 267-270
4. Derezińska, A.: Object-oriented mutation to assess the quality of tests, Proc. of 29th Euromicro Conf., Belek, Turkey, 1-6 Sept. 2003, IEEE Comp. Soc. (2003) 417-420
5. Derezińska, A.: Specification of mutation operators specialized for C# code, ICS Res. Raport 2/05 WUT (2005)
6. Derezińska, A.: Quality assessment of mutation operators dedicated for C# programs, accepted for Inter. Conf. on Quality Software, QSIC06, Beijing, China, Oct. (2006)
7. Kim, S., Clark, J., McDermid J. A.: Class Mutation: mutation testing for object-oriented programs, Proc of Conf. on Object-Oriented Soft. Systems, Erfurt, Germany, Oct. (2000)
8. Kim, S., Clark, J., McDermid J. A.: Investigating the effectiveness of OO testing strategies with the mutation method, J. of Soft. Testing, Verif, and Rel., 11(4) (2001) 207-225
9. Ma, Y-S., Offutt, J., Kwon, Y-R.: MuJava: an automated class mutation system, Softw. Testing, Verif. and Reliab., vol 15, no 2, June (2005)
10. Ma, Y-S., Kwon, Y-R., Offutt, J.: Inter-class mutation operators for Java, Proc. of Inter. Symp. on Software Reliability Engin., ISSRE'02, IEEE Computer Soc., (2002)
11. Voas, J.M., McGraw, G.: Software fault injection, Inoculating programs against errors, John Wiley & sons Inc. (1998)