Object-oriented Mutation Applied in Common Intermediate Language Programs Originated from C#

Anna Derezińska, Karol Kowalski
Institute of Computer Science, Warsaw University of Technology
Warsaw, Poland
e-mail: A.Derezinska@ii.pw.edu.pl

***Abstract*—Application of object-oriented mutation operators in C# programs using a parser-based tool can be precise but requires compilation of mutants. Mutations can be introduced faster directly to the Common Intermediate Language of .NET. It can be simple for traditional mutation operators but more complicated for the object-oriented ones. We propose the reconstruction of complex object-oriented faults on the intermediate language level. The approach was tested in the ILMutator tool implementing few object-oriented mutation operators in the intermediate code derived from compiled C# programs. Exemplary mutation and performance results are given and compared to results of the parser-based mutation tool CREAM.**

Keywords-mutation testing of C#, object oriented mutation operators, Common Intermediate Language of .NET

I. INTRODUCTION

Mutation testing inserts faults in a program under test in order to assess or generate test cases, or evaluate the reliability of the program. When the testing of programs written in an object-oriented language is considered, we cannot ignore faults related to object-oriented programming constructs.

A transformation rule that generates a mutant from the original program is called a *mutation operator*. So-called standard (or traditional) mutation operators introduce small, basic changes that are possible in typical expressions or assignment statements of any general purpose language [1]. Mutation operators dealing with specific programming features, including object-oriented ones, were also developed [3-8].

Traditional mutation operators are not sufficient for verification of tests to reveal object-oriented flaws. Experiments performed with unit tests distributed with various open source programs indicated at the insufficient ability of these tests to detect faults of object-oriented origin in C# programs [9].

Mutation testing should contribute to improving a test suite. A mutant is said to be *killed* if the result of running the original program is different from the mutants' result for at least one test case of the test suite T. A mutant that cannot be killed by any test suite is counted as an *equivalent* one. The adequacy level of the test suite T, so-called *mutation score*, is calculated as a ratio of the number of mutants killed over the total number of non-equivalent mutants.

Mutation testing is known to be very laborious and has high demands on computing resources. Object-oriented mutation can be successfully introduced into a source code of a high-level language, like on C# in the CREAM system [10]. In its next version some improvements in the performance were achieved [9], but a parser-based approach requires mutant recompilation or code interpretation [11].

This paper presents development of object oriented mutations on the intermediate code derived from compiled C# programs. This approach can be considered as an execution cost reduction technique [12], similar to Java Bytecode manipulation [13]. An efficient tool introducing this kind of mutations should make program changes possibly omitting the steps of code compilation or its disassembly/assembly. Introduced changes should not damage a correctly compiled code.

An open question is how the idea can be realized in the .NET environment. Introduction of traditional mutation operators on the intermediate language level is straightforward, as it is similar to the mutation on the high language level. However, the problem addressed in this paper refers to object oriented operators, which are substantially more complex and in general might not be directly reflected on the intermediate level. We have shown how the changes that correspond to selected object oriented mutation operators specified on the high-language (C#) are made in the Common Intermediate Language (CLI) of .NET.

The approach was used in the ILMutator (Intermediate Language Mutator) tool [14] Its first version supported six selected types of object oriented program changes. Modification of a program is performed through manipulation in its metadata and intermediate code, what enables omitting recompilation. The Mono.Cecil library [15] was used for browsing and altering metadata and an intermediate code.

The tool was used to perform mutation and testing of several widely used open source libraries and their unit test suites. It allowed to determine the quality of the tests, which were supplied with these tested programs, the usefulness of the implemented mutation operators and the tool performance.

The paper is organized as follows. In the next section we discuss briefly the background of object-oriented mutation testing. In Sec. 3 we present object oriented mutations at CIL of .NET. Mapping of several mutation operators representing object oriented faults of the C# language to intermediate

A.Derezińska, K. Kowalski, Object-Oriented Mutation Applied in Common Intermediate Language Programs Originated from C#, *Proc. of IEEE 4th Inter. Conf. Software Testing Verification and Validation Workshops (ICSTW)*, IEEE Comp. Soc., 2011, pp. 342 - 350
The original publication is available at http://ieeexplore.ieee.org, DOI 10.1109/ICSTW.2011.54

language was proposed. Basic information about the tool implementing the approach and experimental results is given in Sec. 4 and Sec. 5, respectively. Finally, Sec. 6 concludes the paper.

II. Related Work

The mutation testing approach was developed primarily for structural languages, like Fortran, Ada, C, developing standard mutation operators that can be adopted for any general purpose language [1]. Further, similar standard mutation operators were also applied for object-oriented languages, like Java, C++, C#.

In object-oriented programs standard mutation was used for intra-method level testing. Object oriented languages also provide new structures, like class declarations and references, information hiding, inheritance, polymorphism, method over-loading. They are not considered directly by standard mutation operators, therefore the object-oriented mutation operators were defined.

The research on the object-oriented mutation was done mostly on Java programs [2-5, 13, 16, 17].

Kim et al. [4] proposed 15 mutation operators for Java programs regarding object-oriented concepts and other programming mechanisms (e.g. exception handling). It was extended by Chevalley [2]. Ma et al. proposed the most comprehensive set of 24 class mutation operators for Java described in [5]. These operators were implemented in the tools supporting mutation of Java programs MuJava [13]. Usefulness of the object oriented operators and their orthogonality were studied in experiments [16 - 19].

Mutation of object-oriented features of C++ code and a UML class specification was studied in [6]. Five groups of object oriented operators were proposed and evaluated in experiments. The operators were introduced to programs on the source code level.

Object oriented operators of Java and C++ were examined according to their applicability to the C# language. The set of adapted object oriented operators was extended with other operators dealing with object oriented features (e.g. handling of exceptions) or specific for C# language (e.g. delegates, properties). In result, the set of about 40 advanced (not traditional) mutation operators of C# was proposed and partially evaluated in some experiments [8, 20].

Mutation operators are usually defined informally and illustrated by code examples. To make a definition concise and unambiguous the operators were specified as a transformation with pre- and post-conditions [7, 8]. It is important especially for complex object oriented operators dealing with the structural features of a program.

A comprehensive overview of mutation testing applied to different programming and specification languages can be found in the survey of the development of mutation testing [12].

Simple changes to Java source code, without parser involvement, were implemented in Jester environment [21]. The ideas of Jester system were transformed to Python and C# languages. The Nester tool [22] supports the standard mutations of C# language. The improved version of Nester makes only one compilation run for all mutants. Afterwards, it is decided during test execution which mutant should run. Five traditional mutation operators (sufficient according to Offutt) has been incorporated into PexMutator of C# code applied also in CIL code [23].

Several testing systems for the C++ language use standard mutation testing also in commercial products, as Insure++ from Parasoft [24].

Standard mutations introduced into Java Byte Code are supported in different tools, like Jumble [25], MuGamma [26]. Selected traditional and some object oriented mutations in Java were also implemented in MuJava [13], MuClipse [17] - an Eclipse plug-in based on MuJava, Judy [27] and Javalanche [28] tools. In MuJava, mutants are generated either as a parameterized program (so-called meta-mutant) that is further recompiled and executed as a mutant according to a given parameter, or a mutation is introduced directly in the Byte Code. Test cases considered for Java programs were commonly unit tests suitable for JUnit environment [21, 25, 27, 29], or similar but specialized like in MuJava [13].

The Nmutator tool announced in 2002 was supposed to introduce object oriented mutations into C# programs but it was probably not completed. Research in [30] mentioned object oriented features of C# but concentrated on algorithms for optimization of test cases selection. They referred to standard mutation operators (LOR, NOR, ROR), perturbation on values of constant and variables, invocation of an exception and two object oriented operators (MCR, RFI). Operator MCR replaces a method call by another method with the same signature. Operator RFI forces the reference to an object to be stuck at null after its creation. The object oriented operators were not studied in detail.

The first tool supporting selected object-oriented mutations for C# was the CREAM system [10, 31]. It was a parser-based mutation tool cooperating with the NUnit environment [32]. Further development of the CREAM v2 system, including code parsing improvements, preventing generation of invalid and partially of equivalent mutants, cooperation with the distributed Tester environment and reduction of disc space requirements by storing mutants updates in a SVN repository, is presented in [9]. In comparison to this approach reduction of time requirements could be obtained by introducing mutation at the lower program level, eliminating necessity of many mutant compilations but also a time-consuming interpretation of a high-level program.

In [33] examples of introducing simple mutations into intermediate .NET code are discussed. The principles of a mutation tool based on this approach can be summarized in the following way:

- disassembly of an assembly file into a managed code expressed in a textual format,
- searching for a mutation location according to pattern matching,
- introduction of mutation into found locations,
- assembly of the text file with intermediate code including a mutant.

The author persuades of an easy way of mutant implementation, but the solution has two main limitations. The disassembly and assembly steps require a time overhead for each mutant (can be compared to the compilation of C# code). Secondly, introduction of mutation via simple pattern matching can be successful but only for a subset of simple traditional mutation operators. Therefore we present idea of introduction of selected object oriented mutations directly in the intermediate code derived from compiled C# programs and their implementation in the ILMutator prototype [14]. The idea is similar to the Bytecode translation of Java programs realized for structural mutants in MuJava [13]. In order to introduce advance mutation operators possible forms of intermediate code generated by the compiler were examined and all structures corresponding to primary C# constructions distinguished. Introduction of this kind of operators requires an intermediate code analysis, but it can be automated for various operators, as discussed below.

III. Object-oriented Mutation Injected on Common Intermedite Language Level

In this section we present how object-oriented mutation operators can be introduced directly into Common Intermediate Language. Examples of several operators are provided.

A. Common Intermediate Language

The C# language is one of the languages the applications of which can be run on .NET Framework using a common runtime environment [34]. The basic components of the framework are *Common Language Runtime* (CLR) and *Framework Class Library* (FCL). CLR is a runtime environment that supports the object-oriented programming paradigm, including concepts of types, objects and their behavior. All .NET applications are run in CLR - an intermediate layer between them and the operating system. FCL includes library classes supporting standard functionality for applications, e.g. GUI interfaces, web facilities. Compilers of different programming languages, like C++, C#, Visual Basic, J#, IL Assembler, can be used, building a code aimed at the common runtime environment. A so-called assembly is created, which is an intermediate form of a .NET application that can be run in the CLR environment. An assembly consists of managed modules that include two parts: metadata and managed code. Metadata describes a structure of the application, its elements and relations. Managed code includes body of application methods written in an intermediate form called *Microsoft Intermediate Language* (MSIL) or *Common Intermediate Language* (CIL) [35].

The intermediate language is a machine-level language. In comparison to other low-level languages, it includes instructions for creating, initializing and manipulating object types. It also supports array manipulation and exception handling. The language has different mechanisms allowing to exploit all capabilities of the CLR layer. However, programs translated from a high-level source language (e.g. C#) use only a subset of these mechanisms [36]. CIL is a stack-based language. All arguments are taken from a stack and return values are put on it. An instruction consists of one or two bytes operation code (*opcode*) and an optional parameter.

B. Mutation Operators

A mutation operator defined for a given programming language is a transformation of language instructions. It substitutes given language structures with a defined set of others. The source and target code should be syntactically correct. For the traditional mutation operators the substituted sets are reduced to a single instruction or even a single logical or arithmetical operators and are not disseminated in a program. Object-oriented or other advanced operators in a high-level language can refer to complex structures that are not only locally interpreted but are distributed over a whole program, e.g. class hierarchy. However, the most of changes are bounded to a single instruction at this language level.

While reflecting an object oriented mutation operator on the intermediate language level, we have to identify appropriate language structures and introduce changes into several instructions of this lower level. Below, exemplary object oriented mutation operators of C# and their corresponding operators in the Intermediate Language are discussed. We also define correctness conditions that should be met in order to create valid mutants.

Operator PNC (*new method call with child class type*) swaps a calling of operator *new* of a non-parametric constructor of a given class, with a constructor of an inherited class. A given class can be used as a base class in different assemblies distributed on different physical places. Hence, inherited classes are searched only in the assembly in which a given base class is defined. The PNC operator applied in the C# and CIL code is illustrated by an example (Fig. 1).

After identification of a constructor calling in the CIL code, the appropriate conditions are checked. The assembly defines a type that inherits for the given type, as *classB* inherits from *classA* in the example. The considered type should have the definition of a non-parametric constructor. A type name can be substituted in the appropriate constructor call. In this case the change in the intermediate language is also made only in one instruction, as at the C# level.

Operator OMR (*overloading method contents change*) substitutes a body of an overloaded method having some parameters (let call it method A) with a calling of an overloaded method with a less number of parameters (method B). The necessary condition of the substitution is existence of at least one variation of method A parameters that corresponds to parameters of calling method B. To avoid a recursive calling of methods, there should be no calling of method A in the body of method B. An example of the operator is shown in Fig. 2.

If all correctness conditions of the operator are satisfied, calling of method *B* is inserted into the body of *A* before its first instruction. On the intermediate level it consists of several instructions, namely an instruction putting parameters into a stack, an instruction for calling a method will less number of parameters, and a *return* instruction. The rest of instructions of method *A* will be deleted. The

Before mutation

```
//C#
public class ClassA
        {         }
public class ClassB:ClassA
        {         }
public void initiate()
        {
            ClassA a;
            a = new ClassA();
        }

//CIL
 .method public hidebysig instance
void  initiate() cil managed
{
 // Code size 8 (0x8)
      .maxstack  1
      .locals init ([0] class Operators.ClassA a)
      IL_0000: nop
      IL_0001: newobj instance void
      Operators.ClassA::.ctor()
      IL_0006: stloc.0
      IL_0007: ret
} // end of method
      //Program::initiate
```

After mutation

```
 public class ClassA
    {        }
 public class ClassB:ClassA
    {        }
 public void initiate()
            {
                    ClassA a;
                    a = new ClassB();
             }

.method public hidebysig instance
void  initiate() cil managed
 {
  // Code size 8 (0x8)
  .maxstack  1
  .locals init ([0] class Operators.ClassA a)
   IL_0000: nop
   IL_0001: newobj instance void
   Operators.ClassB::.ctor()
   IL_0006: stloc.0
   IL_0007: ret
   } // end of method
   //Program::initiate
```

Figure 1. PNC example - C# and its corresponding intermediate code.

Before mutation

```
//C#
public class ClassA
     {     void count(int a)
                {         }
           void count(int a, int b)
                {         }
     }

//CIL
.method private hidebysig instance
void  count(int32 a, int32 b) cil managed
{
// Code size 2 (0x2)
        .maxstack  8
        IL_0000: nop
        IL_0001: ret
}//end of method ClassA::count }
```

After mutation

```
public class ClassA
{       void count(int a)
              {         }
 void count(int a, int b)
           {    count (a);     }
}

.method private hidebysig instance
void  count(int32 a, int32 b) cil managed
{
 // Code size  10 (0xa)
   .maxstack  8
   IL_0000: nop
   IL_0001: ldarg.0
   IL_0002: ldarg.1
   IL_0003: call instance void
   Operators.ClassA::count(int32)
   IL_0008: nop
   IL_0009:  ret
 } //end of method ClassA::count
```

Figure 2. OMR example - C# and its corresponding intermediate code.

sequence of inserted instructions can be seen on the CIL level from Il_0000 to Il_0003.

Each constructor of a class is transformed to a special *.ctor()* method on the intermediate level. The method consists of three sections. In the first section initial values are assigned to all fields that were initialized in the class definition. In the second section an appropriate constructor of the base class or another constructor of the same class is called. Instructions realizing operations defined directly in the constructor in the C# code are placed in the third section. This constructor structure was used in mutation operators dealing with constructors, such as JDC and JID.

Operator JDC (*C#-supported default constructor create*) deletes a definition of a non-parametric constructor. Therefore the C# compiler creates a default constructor. This operator is used when this non-parametric constructor is the only one constructor of the class. On the intermediate language level it is also checked if exactly one constructor exists. In case of a unique constructor, all instructions of its third section are deleted. It simulates deletion of instructions defined in a constructor body in the C# language. The constructor remains with its two initial sections and is the same as if it were created by the compiler.

Operator JID (*member variable initialization deletion*) deletes a member variable initialization that was placed in the variable definition. The operator will be applied only for primitive types because deletion of an initialization of a field of a reference type could usually cause an erroneous call to a non-initialized field. In the intermediate language this initialization is deleted from all constructors available in this class. The initialization is placed in the first section of any constructor and consists of few instructions. If the initialization refers to a static field, it should be deleted from the static constructor of the class.

For example, to substitute a variable initialization "*private int x = 5*;" with "*private int x*;" in a C# code, the following instructions will be deleted in each constructor:

```
IL_0000: ldarg.0
IL_0001: ldc.i4.5
IL_0002: stfld int32 Operators.ClassA::x
```

In C# all non-static instance constructors call another instance constructor. As a result the following three possibilities can be distinguished:

1. A class inherits from its base class and a constructor of the base class is called using *:base(args)* expression. The constructor with the given list of parameters has to be declared in the base class.
2. Another constructor of the same class is called using *:this(args)*. In this case, the appropriate constructor with given parameters is declared in the same class.
3. Non of the above structures is used while calling a constructor. It is assumed that a non-parametric constructor of the base class is called. It is equivalent to the usage of *:base()* phrase.

Operator IPC (*explicit call of a parent's constructor deletion*) deletes calling of a parametric constructor of the base class used in an inheriting class. In C#, this constructor call is substituted by a default calling of a non-parametric constructor of the base class. The operator is applied only if the base class has its non-parametric constructor. This deleting of the constructor can be realized in the intermediate language as modification of the appropriate constructor call. For example, in the second section of a constructor the following instructions:

```
IL_0000: ldarg.0
IL_0001: ldarg.1
IL_0002: call instance void
Operators.Class.A::.ctor(int32)
```

will be substituted by the sequence:

```
IL_0000: ldarg.0
IL_0001: call instance void Operators.Class.A::.ctor()
```

Operator EOC (*reference comparison and content comparison replacement*) swaps occurrences of method *Equals* used for variable comparison with usage of operator "==", and vice versa. In C#, for EOC operator we only consider comparison on arguments of reference type. Value types are omitted, because their *Equals* methods are implemented as usage of operator "==" and there is no difference in the .NET platform. We also omit *System.String* type as both cases refer to the same static private method for comparing strings.

While introducing EOC in the intermediate language, we have to differentiate two cases. In the first case, operator "==" is overloaded neither in a considered class nor in any of its base classes. Appropriate *Equals* instruction available in the intermediate level will be substituted with *ceq* (compare equal) instruction. A reverse transformation is done in the similar way, but an occurrence of *ceq* instruction and its context defined by current parameters on the stack should be carefully examined.

In the second case, operator "==" is overloaded in the considered class or any of its base classes. Instead of instruction *ceq* an appropriate version of the overloaded method *op_Equality* is considered. The overloaded operator is searched in the class hierarchy according to the principles given in the C# language specification [37].

IV. ILMUTOTOR SYSTEM

The ILMutator (Intermediate Language Mutator) system was designed to support mutation of programs in .NET environment. Its first version [14] introduced object-oriented mutations in the intermediate code derived from compiled C# programs.

ILMutator reads a correct assembly compiled from a C# program and prepared for running in *Common Language Runtime*. The mutation operators are selected and optionally a maximal number of generated mutants can be set. The assembly is searched for a location were a mutation can be introduced. The assembly is modified and the mutant is stored in a disc. A separate assembly is created for each location where an operator is introduced (i.e. it is not a meta-mutant approach [13]). On demand, a user can view the

original intermediate code and the mutated code observing highlighted differences.

The tool supports execution of tests on the original and mutated assemblies. A mutant is counted as killed if its test result is different than the result of the original program.

The ILMutator system consists of the following components:

- functional modules (assembly management, mutation operators processing, test management),
- visualization modules (viewing assembly intermediate code, presenting test results),
- helpers modules (DiffEngine - to compare original and mutated code, PEVerify, NUnit, Mono.Cecil).

Portability of .NET applications in different operating systems is supported by the Mono project. One component of the project – the Mono.Cecil library [15] is used for creation and exploitation a code written in the intermediate language consistent with the format ECMA CIL [35]. It supports reading and modifying metadata and an intermediate code stored in a managed portable, executable file. A roundtrip operation on an assembly can be performed. Capabilities of the library assists ILMutator during the intermediate code analysis for identification of mutation operator's areas, its conditions, and changing of appropriate instructions.

An application on the intermediate level does not have to be compiled, but its correctness can be verified. The PEVerify tool distributed within .NET Framework SDK can be used for verification an assembly consisting of managed modules. It validates metadata (MDValidator) and intermediate managed code (ILVerifier). The tool is used in ILMutator to verify an intermediate code after a mutation was introduced.

In the prototype tool [14] six object-oriented mutation operators (EOC, IPC, JDC, JID, OMR and PNC) were implemented. Its next release was extended with some traditional operators, as well as operators dealing with exceptions and delegtes:

- EHR (Exception handler removal),
- EHC (Exception handling change),
- DMC (Delegated method change),
- DMO (Delegated method order change).

V. EVALUATION

Assemblies from the Castle project [38] and the NUnit library [32] were analyzed using the ILMutator tool. The assemblies are public distributed with their unit tests. The basic complexity measures, such as program size, number of code lines, classes and unit tests associated with the assembly are given in Tab. 1.

A. Experimental results

In experiments six object-oriented mutation operators were applied. The number of mutants depends on a programming style and a program domain influencing the number of different structures used in a program (Tab. 2). A column of PNC operator is omitted as no mutant was created for any program. In these programs no local variable was initialized with a non-parametric constructor of a type to which its base constructor could be applied. In all programs some comparisons were performed and EOC operator generated nonempty sets of mutants. Operator IPC generated more mutants for assemblies from the Castle Project, as there were more types defined and the inheritance mechanisms were more frequently used than in the NUnit assemblies. Operator JDC was rarely used because the most of types have more than one constructor. Operator JID was applied especially often in the NUnit.uikit assembly. In NUnit many graphical components with default values of their properties are defined. OMR operator created mutants for all assemblies, but particularly many for the framework of NUnit. In this program many assertions are defined and used for comparison of expected and actual results of unit tests. Therefore many overloaded methods can be mutated with OMR. We can observe that using different operators the investigation of a test suite is adjusted to the object-oriented mechanisms really used in the program under test.

TABLE I. MUTATED ASSEMBLIES

	Program	Size [kB]	LOC	Classes	Unit tests
1	Castle.Dynamic Proxy	76	5036	71	82
2	Castle.Core	60	6119	50	171
3	Castle.Micro Kernel	112	11007	86	88
4	Castle.Wiondsor	64	4240	34	92
5	Nunit.framework	40	4415	37	397
6	NUnit.mock	20	579	6	42
7	NUnit.util	88	6405	34	211
8	NUnit.uikit	352	7556	30	32

TABLE II. NUMBER OF MUTANTS IN PROGRAMS PER OPERATORS

No	EOC	IPC	JDC	JID	OMR	Σ
1	20	4	2	1	6	33
2	37	13	2	4	29	85
3	76	27	5	3	28	139
4	15	16	2	3	12	48
5	4	7	0	3	231	245
6	17	2	0	0	1	20
7	12	3	1	13	23	52
8	10	3	5	46	20	84
Σ	191	75	17	73	350	706

The original program and all created mutants were run with the unit tests associated with the considered programs. Mutants were killed when their test results differed from the test results of the original program (i.e. a testing assertion passed in the original program and failed in a mutant or vice versa) or when an exception was detected. The last case encountered in two assemblies where mutants of OMR operator caused *StackOverflowException* due to a recursive method call.

The numbers of killed mutants per each operator are given in Tab. 3. The last column shows a mutation score indicator calculated for a program and a given test suite. The

TABLE III. KILLED MUTANTS

	EOC	IPC	JDC	JID	OMR	Σ	[%]
1	6	1	1	0	5	13	41
2	7	3	0	0	10	20	40
3	20	3	4	0	17	44	47
4	15	2	2	0	1	20	34
5	4	3	0	2	68	77	46
6	2	2	0	0	1	5	35
7	3	0	0	1	22	26	45
8	5	2	1	1	0	9	38
Σ	62	16	8	4	124	214	
[%]	32.5	21.3	47.1	5.5	35.4	30.3	

summarized mutation results are not very high. It confirms, similarly as during previous experiments using the CREAM tool, that tests aimed at checking basic program functionality do not detect all object-oriented flaws.

Mutants generated with ILMutator were compared with results of the CREAM tool - a parser-based mutation environment of C# programs [9]. The comparison referred to only three object-oriented operators EOC, IPC and JID that were common for both tools. Numbers of generated and , killed mutants are given in Tab. 4. It was checked that, as presumed, sets of mutants created by ILMutator were proper subsets of mutants generated by CREAM v2.

IPC operator had the same interpretation in both tools and generated the same mutants. Therefore also the number of killed mutants were the same, and calculated mutation scores would be identical.

For operators EOC and JID less mutants were generated by ILMutator. This follows directly for the more restrictive assumptions about operators. The EOC operator was applied only for reference types. Mutation comparison of primitive types, as well as types *System.String* and *System.Object* were omitted. In the case of JID operator initialization of primitive types were mutated, whereas initialization of referenced types remained unchanged. These conditions prevented creation of an invalid code and possibly many equivalent mutants. Although some valid mutants were also not generated. It should be noted that identification of a mutated location is more difficult on the intermediate level and we would like to generate less mutants corresponding to a desired high-level structure but for sure valid and possibly not equivalent.

In case of operators EOC and JID, the number of generated mutants and the number of killed mutants differ; the mutation scores are lower for CREAM. However, especially for EOC operator, a part of mutants that were generated by CREAM and not killed but not generated by ILMutator were equivalent. After omitting these equivalent mutants, the mutation scores were similar (difference was no bigger than 3%).

In two last columns of Tab. 4. times of introduction of all mutations are shown. In case of the CREAM tool, it includes parsing of the code, its analysis and storing a modified program, but also compilation time of a generated mutant with the external compiler. Based on given data we can calculate that generation time for a single mutant was on average about 13 second, in detail for operator EOC 13 s, IPC 13.45 s and JID 12.68 s accordingly. A time delay measured for ILMutator consists of the assembly analysis, changing the intermediate code and storing it to a disc. The averaged time per one mutant was about 0.25 second (EOC 0.27, IPC 0.3, JID 0.22). It showed, that for examined assemblies and given operators the time performance was about 50 times better. For other programs and operators the results can be different, but as expected, the time improvement was very significant.

However, the overall time of mutation testing is influenced not only by a mutant generation time but also by many factors, such as a time of a test execution depending on a program complexity, a sequential or distributed execution of many mutants, a usage of repository for storing mutants

TABLE IV. COMPARISON OF MUTANTS' NUMBER AND MUTANT GENERATION TIME FOR COMMON OBJECT ORIENTED OPERATORS

Program		Operator	Mutants generated | killed | killed/generated[%]						Mutation time [s]	
			CREAM			*ILMutator*			*CREAM*	*ILMutator*
1	Castle.Dynamic Proxy	EOC	42	9	21	20	6	30	568	6.5
		IPC	4	1	25	4	1	25	66	2.0
		JID	39	33	84	1	1	100	543	0.4
2	Castle.Core	EOC	74	9	12	37	7	19	917	7.5
		IPC	13	3	23	13	3	23	131	3.0
		JID	22	7	32	4	4	100	294	0.6
7	NUnit.util	EOC	107	15	14	12	3	25	1416	4.7
		IPC	3	0	0	3	0	0	72	1.0
		JID	34	28	82	13	13	100	368	3.0

(local or remote, storing as a whole or in an incremental repository) [9], automation of the whole testing process management. As an example, average testing time for a mutant (including test evaluation by a NUnit compatible tool) was about 0.5 s for Castle.DynamicProxy and about 0.24 s for Castle.Core [39]. It shows, that a mutant generation time for this sort of programs was significant longer than a test execution time in case of the parser-based CREAM, and the times were of the same magnitude in case of ILMutator.

In this paper we deal with the object-oriented operators, as realization of traditional operators in CIL code is not such a difficult task. Other tools dealing with operators for C#, like Nester [22] and Pexmutator [23], do not consider any object-oriented operators. Therefore the evaluation results are not comparable.

There are tools that support object-oriented operators for Java language, like [13,28]. The exemplary OBJECT ORIENTED operators, for which CIL implementation was presented, have their corresponding operators in MuJava. However, they use different source and target languages. Thus, the problems of mapping programming structures in .NET environment discussed here are irrelevant.

B. Threads to Validity

While interpreting experiment results several threats to validity should be taken into account.

Threats to external validity are conditions that limit the ability to generalize the results of experiments. The subjects chosen for the analysis were "real", practically used programs. Also the examined test suites were developed in advance independently of the concerned mutation process. On the other hand, the limited number of mutation operators was used in experiments. A threat to the mono-operational bias was lowered by conducting experiments not only on one, but on several, different programs.

The basic limitation of the approach is the ability to imitate a complex fault from the high programming level at the lower level. As we have shown it is possible for selected advanced operators. The number of generated mutants was limited in comparison to parser-based mutation in order to avoid possibility of creation of equivalent mutants.

However, we cannot guarantee that this approach can be applied for any kind of object-oriented or other specialized operators. Each operator have to be carefully examined analyzing all possible situations in the corresponding intermediate code. It is also sensitive to the current version of the compiler that translates C# code to its intermediate form. Apart from discussed six operators also other can be implemented in this way (it is made for few in the next tool release). In general, even complex object oriented operators of C# can be implemented in CIL, but some operators will not cover all possible situations and generate less mutants. An open question is how this will influence mutation score for some operators.

Another fact is that it is not worthwhile to use all operators due to their tendency to create equivalent mutants. On the other hand, many new structures introduced in C# 2.0 and 3.0 language versions are not distinguishable on the intermediate level and their faults are inappropriate to create mutation operators like that. This problem was discussed in [40].

In order to cope with threats to the statistical conclusion validity the selected programs were not very small. However, due to a rare usage of some programming structures, a limited number of mutants was created during experiments. The calculated mutation scores can be treated approximately. Therefore the comparison of mutation scores obtained by both tools can be generalized only to some extend. The presented evaluation results illustrate realization possibilities of the approach and performance improvements. We do not focus here on statistical evaluation of the object-orented operators, hence we do not related the statistics to more complete experiments of C# [41] or Java [19].

VI. CONCLUSIONS

Introducing mutations on the intermediate language level turned out, according to expectations, more efficient than to the high-level source program. Possibility of defining object-oriented operators for several faults reflecting the object-oriented faults at C# level was presented. A mutated program does not have to be recompiled and thanks to direct manipulation on the intermediate code via the Mono.Cecil library no disassembly or assembly was necessary. In the comparison to a parser-based mutation it has more limitations concerning identification of mutation locations and correctness conditions.

The improved version of ILMutator is enhanced with greater number of mutation operators, other ways of generating and storing mutants, and diverse methods of testing, e.g. cooperation not only with NUnit. It supports visualization of mutation changes not only on the intermediate but also the C# code level. It could improve identification of equivalent mutants, if necessary. Further investigation of object-oriented operators using both C# related tools considers analysis of dependencies between operators, reduction of mutants number via selection of mutants subsets for a given operator etc.

REFERENCES

[1] J.M. Voas, G. McGraw, Software Fault Injection, Inoculating ProgramsAgainst Errors, John Wiley & Sons Inc., 1998.

[2] P. Chevalley, “Applying mutation analysis for object-oriented programs using a reactive approach,” in Proc. of 8-th Asia-Pacific Soft. Eng. Conf., ASPEC 2001, pp. 267-270.

[3] P. Chevalley, P. Themvenod-Fosse, “A mutation analysis tool for Java programs,” J. on Software Tools for Technology Transfer (STTT), Vol 5 Issue 1, Nov. 2003, pp. 90-103.

[4] S. Kim, J. Clark, J. A. McDermid, “Class Mutation: mutation testing for object-oriented programs,” in Proc. of Net.ObjectDays Conf. on Object-Oriented Software Systems, Erfurt, Germany, Oct. 2000.

[5] Y-S. Ma, Y-R. Kwon, A.J. Offutt, “Inter-class mutation operators for Java,” in Proc. of 13-th Inter. Symphosium on Softwre Reliability Engineering, ISSRE'02, IEEE Computer Soc. 2002.

[6] A. Derezińska, “Object-Oriented Mutation to Assess the Quality of Tests,” in Proc. of the 29th Euromicro Conf., IEEE Comp. Society, 2003, pp.417-420.

[7] A. Derezińska, Specification of mutation operators specialized for C# code, Inst. of Computer Science Research Raport 2/05, Warsaw University of Technology, 2005.

[8] A. Derezińska, “Advanced mutation operators applicable in C# programs,” in K. Sacha (ed.) IFIP Vol. 227, Software Engineering Techniques: Design for Quality, Springer, Boston, 2006, pp. 283-288.

[9] A. Derezińska, A. Szustek, “Object-Oriented Testing Capabilities and Performance Evaluation of the C# Mutation System,” Pre. of Proc. 4th Central and Eastern Europen Conf. on Softwre Engeenieering Technique, Krakow, Poland, 2009, pp. 270-283.

[10] A. Derezińska, A. Szustek, “CREAM - a System for Object-oriented Mutation of C# Programs,” in: S. Szczepański, M. Kosowski, and Z. Felendzer (eds.) Annals Gdansk University of Technology Faculty of ETI, no 5, Information Technology, vol.13, Gdańsk, 2007, pp. 389-406.

[11] B. Choi, A. P. Mathur, “High-performance mutation testing,” Journal of Systems and Software, vol. 20, no. 2, Feb. 1993, pp.135-152.

[12] Y. Jia, M. Harman, “An analysis and survey of the development of mutation testing,” Crest Centre, King's College London, Technical Report TR-09-06, 2009, http://www.dcs.kcl.ac.uk/pg/jiayue/repository/

[13] Y-S. Ma, A.J. Offutt, Y-R. Kwon, “MuJava: an automated class mutation system,” Soft. Testing, Verification & Reliability, vol 15, no 2, June 2005, pp.97-133.

[14] K. Kowalski, Implementing object mutations into intermediate code for C# programs, Bach. Thesis, Inst. of Comp. Science, Warsaw Univ. of Techno., 2008 (in polish).

[15] Mono.Cecil, http://www.mono-project.com/Cecil

[16] H-J. Lee, Y-S Ma, Y-R. Kwon, “Empirical evaluation of orthogonality of class mutation operators”’ in 11th Asia-Pacific Softwar. Engineering Conf., IEEE Comp. Soc. 2004.

[17] B. H. Smith, L. Williams, “An Empirical Evaluation of the MuJava Mutation Operators,” Mutation'07 at TAIC.Part'07, 3th Inter. Workshop on Mutation Analysis, Cumberland Lodge, Windsor UK, Sep. 2007, pp. 193-202.

[18] Y.-S. Ma, M. J. Harrold, Y.-R. Kwon, “Evaluation of Mutation Testing for Object-Oriented Programs,” in Proc. of the 28th Inter. Conf. on Software Engeeneering (ICSE 06), Shanghai, China, 20-28 May, 2006, pp. 869-872.

[19] M.Y-S Ma, Y-R Kwon, S-W Kim, “Statistical Investigation on Class Nutation Operators,” ETRI Journal, vol 31, No 2, April 2009, pp. 140-150.

[20] A. Derezińska, “Quality Assessment of Mutation Operators Dedicated for C# Programs,” in: 6th Inter. Conf. on Quality Software, QSIC'06, Beijing, China, Oct. 2006, IEEE Computer Soc. Press, California, 2006, pp. 227-234.

[21] I. Moore, Jester a JUnit Test Tester. eXtreme Programming and Flexible Process in Software Engineering - XP2000, 2000.

[22] Nester, http://nester.sourceforge.net/

[23] Pexmutator, http://www.pexase.codeplex.com

[24] Parasoft Insure++, http://www.parasoft.com/jsp/products/home.jsp?product=Insure

[25] Irvine, S. A., Pavlinic at. all, “Jumble Java Byte Code to Measure the Effectiveness of Unit Tests,” Mutation'07 at TAIC.Part'07, 3th Inter. Workshop on Mutation Analysis, Cumberland Lodge, Windsor UK, Sep. 2007, pp. 169-175.

[26] S-W. Kim, M. J. Harrold, Y-R. Kwon, “MuGamma: Mutation Analysis of Deployed Software to Increase Confidence and Assist Evolution,” in 2nd Workshop on Mutation Analysis, Mutation 2006, Releigh, North Carolina, Nov. 2006.

[27] L. Madeyski, “On the effects of pair programming on thoroughness and fault-finding effectiveness of unit tests,” in: J. Munch, P. J. Abrahamsson (eds.) Profes 2007. LNCS, vol. 4589, Springer, Heidelberg, 2007, pp. 207-221.

[28] B.J.M. Grun, D. Schuler, A. Zeller, The Impact of Equivalent Mutants. 4th Inter. Workshop on Mutation Analysis, Mutation'09, Denver, Colorado, 1-4 Apr. 2009, pp. 192-199.

[29] H. Do, G. Rothermel, A Controlled Experiment Assessing Test Case Prioritization Techniques via Mutation Faults. in Proc. of the 21st IEEE Int. Conf. on Software Maintanance, IEEE Comp. Soc., 2005, pp. 411-420.

[30] B. Baudry, F. Fleurey, J-M. Jezequel, Y. Le Traon, “From genetic to bacteriological algorithms for mutation-based testing,” Software Testing Verification. and Reliability, vol 15, no 2, June, 2005, pp.73-96.

[31] A. Derezińska, A. Szustek, “Tool-supported mutation approach for verification of C# programs,” in W. Zamojski, et al. (eds.) Inter. Conf. on Dependability of Computer System, DepCos, IEEE Comp. Soc., 2008, pp. 261-268.

[32] NUnit, http//www.nunit.org

[33] J. McCaffrey, “Create a Simple Mutation Testing System with the .NET Framework,” MSDN Magazine, The Microsoft Journal for Developers, vol. 21, no 5, Apr. 2006.

[34] D. Chappell, Understanding .NET, Second Edition, Addison Wesley Professional, 2006.

[35] Standard ECMA-335, Common Language Infrastruc-ture (CLI), 4th edition (June 2006), http://www.ecma-international.org/publications/standards/Ecma-335.htm

[36] J. Richter, CLR via C#, 2nd Ed., Microsoft Press 2006.

[37] Standard ECMA-334, C# Language Specification, 4th edition, June 2006, http://www.ecma-international.org/publications/standards/Ecma-334.htm

[38] Castle Project, http://www.castleproject.org

[39] A. Derezińska, K. Sarba, “Distributed environment integrating tools for software testing,” in: K. Elleithy (Ed.) Advanced Techniques in Computing Sciences and Software Eng., Springer Dordrecht, Netherlands, 2010, pp.545-550.

[40] A. Derezińska, “Analysis of Emerging Features of C# Language Towards Mutation Testing,” in J. Mazurkiewicz, et al. (eds.), Models and methodology of system dependability, Monographs of system dependability, vol. 1, Publish. Wrocław University of Technology, 2010, pp. 47-59.

[41] A. Derezińska, “Classification of Operators of C# Language,” in L. Borzemski et al. (Eds.) Information Systems Architecture and Technology, New Developments in Web-Age Information Systems, Publish. Wrocław Univ. of Technology 2010, pp.261-271.