Object-oriented mutation applied in Common Intermediate Language programs originated from C#

Anna Derezińska, Karol Kowalski

Institute of Computer Science
Warsaw University of Technology

www.ii.pw.edu.pl/~adr/
Outline

- Object-oriented mutations
- Common Intermediate Language
- O-O mutations on CIL level
- ILMutator system
- Experiments
- Conclusions
Object-oriented mutations

- OO – misusing of class and object interrelations
- Locally interpreted or distributed over a whole program, e.g. class hierarchy
- Single instruction at high-level language
- Several instructions at low-level language, e.g. Common Intermediate Language
- Advanced operators – more language-related than standard (traditional) mutation operators
Advanced operators for C#

C# 1.1 Specified 40 mutation operators including:
- analogous to Java adopted for C#
  - with different specifications
  - with different application scope
- for specific features of C#:
  delegates, properties, indexers, override modifier

C# 2.0, 3.0,.. many new features not suitable for mutation: sealed modifier, generics, partial classes and methods, extension, anonymous methods, .. (Many not applicable in the CIL)
Mutation tools for C#

- Nester – simple mutation by use of regular expressions
- PexMutator – standard mutation operators
- CREAM – parser based,
  18 object-oriented, 8 standard mutation operators (v3)
- ILMutator – mutation operators in the Intermediate Language of .NET originated from C# code
Common Intermediate Language

- Common Language Runtime (CLR) – runtime environment of Microsoft .NET Framework
- Assembly = metadata + managed code
- Managed code = Common Intermediate Language (CIL)

- Machine level language exploiting all capabilities of CLR
- Programs translated from C# use only subset of these capabilities
O-O mutations on CIL level

PNC – new method call with child class type
OMR operator (Overloading method contents change)

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

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

Pre:
- Avoiding recursive call of methods
- At least one consistent combination of parameters
.method private hidebysig instance
void count(int32 a, int32 b) cil managed
{
    .maxstack 8
    IL_0000: nop
    IL_0001: ret
}

                .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
///C#
public class ClassB
{
    private int a;
    private int b = 1;

    public ClassB()
    {
        a = 2;
    }

    ....
    .ctor() ....
    {
        // initialization of fields defined in ClassB
e.g. a=0; b=1;

        // constructor of the base class or another constructor of this class is called
        // constructor body
e.g. a = 2;
    }
}
Constructors changed by operators

**JDC** – *C*-supported default constructor create
Pre: A non-parametric constructor is the only class constructor
This constructor is deleted
CIL – 3rd section of the constructor is deleted
(= constructor without its body)

**JID** – *member variable initialization deletion*

```csharp
private int a = 5;  // private int a;
```
Initializations deleted from the 1st section of all constructors
Restriction: only primitive types
O-O mutations on CIL level

JDC – C#-supported default constructor create

//C# before mutation
public class ClassA
{
    private int a;
    public ClassA()
    {
        a = 5;
    }
}

//C# after mutation
public class ClassA
{
    private int a;
}
O-O mutations on CIL level

JID – member variable initialization deletion

//C# before mutation
public class ClassA
{
    private int a = 5;
    public ClassA()
    {
        public ClassA(int a)
        {
            this.a = a;
        }
    }
}

//C# after mutation
public class ClassA
{
    private int a;
    public ClassA()
    {
        public ClassA(int a)
        {
            this.a = a;
        }
    }
}
### IPC operator (Explicit call of parent's constructor deletion)

<table>
<thead>
<tr>
<th>Original C# code:</th>
<th>Mutated C# code:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public class Vehicle</code></td>
<td></td>
</tr>
<tr>
<td><code>{ private int x;</code></td>
<td></td>
</tr>
<tr>
<td><code> public Vehicle() {...}</code></td>
<td></td>
</tr>
<tr>
<td><code>  public Vehicle(int x)</code></td>
<td></td>
</tr>
<tr>
<td><code>  { this.x = x; }</code></td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
</tr>
<tr>
<td><code>public class Car:Vehicle</code></td>
<td></td>
</tr>
<tr>
<td><code>{ public Car(int y) :base(y)</code></td>
<td></td>
</tr>
<tr>
<td><code>{  }</code></td>
<td></td>
</tr>
<tr>
<td><code>{...}</code></td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Original C# code:</th>
<th>Mutated C# code:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public class Vehicle</code></td>
<td></td>
</tr>
<tr>
<td><code>{ private int x;</code></td>
<td></td>
</tr>
<tr>
<td><code> public Vehicle() {...}</code></td>
<td></td>
</tr>
<tr>
<td><code>  public Vehicle(int x)</code></td>
<td></td>
</tr>
<tr>
<td><code>  { this.x = x; }</code></td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
</tr>
<tr>
<td><code>public class Car:Vehicle</code></td>
<td></td>
</tr>
<tr>
<td><code>{ public Car(int y) </code></td>
<td></td>
</tr>
<tr>
<td><code>{  }</code></td>
<td></td>
</tr>
<tr>
<td><code>{...}</code></td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
</tr>
</tbody>
</table>

**Pre: Base class defines its non-parametric constructor**
### IPC operator

(Explicit call of parent's constructor deletion)

<table>
<thead>
<tr>
<th>Original CIL code:</th>
<th>Mutated CIL code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance void .ctor(int32 b)</td>
<td>instance void .ctor(int32 b)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>.maxstack 8</td>
<td>.maxstack 8</td>
</tr>
<tr>
<td>IL_0000: ldarg.0</td>
<td>IL_0000: ldarg.0</td>
</tr>
<tr>
<td>IL_0001: ldarg.1</td>
<td>IL_0001: call instance void Operators.Car::ctor(int32)</td>
</tr>
<tr>
<td>IL_0002: call instance void Operators.Car::ctor(int32)</td>
<td>}</td>
</tr>
<tr>
<td>..........</td>
<td>..........</td>
</tr>
</tbody>
</table>
| } | }
ILMutator system

- Intermediate Language Mutator supports mutation of programs in .NET environment
- Introduces standard and object-oriented mutations in the intermediate code derived from compiled C# programs using Mono.Cecil library
- User can view the original intermediate code and the mutated code with highlighted differences
ILMutator system

- Execution of tests on the original and mutated assemblies (NUnit)
- Verification of mutated assemblies with PEVerify tool (delivered with .NET Framework)
- Implements 4 standard and selected object-oriented operators
ILMutator system - architecture

- Operator management
  - Mono
  - Cecil
- Assembly management
- Test management
- Test result visualisation
- Assembly visualisation
  - DiffEngine
  - PEVerify
  - NUnit
ILMutator system – during work

![ILMutator system screenshot](image-url)
ILMutator system – test runner
Experiments – mutation operators

**EOC** – reference comparison and content comparison replacement

**IPC** – explicit call of a parent’s constructor deletion

**JDC** – C# supported default constructor create

**JID** – member variable initialization deletion

**OMR** – overloading method contents change

**PNC** – new method call with child class type

0 mutants
### Experiments - mutated assemblies

<table>
<thead>
<tr>
<th>Program</th>
<th>Size [kB]</th>
<th>LOC</th>
<th>Classes</th>
<th>Unit tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Castle.Dynamic Proxy</td>
<td>76</td>
<td>5036</td>
<td>71</td>
<td>82</td>
</tr>
<tr>
<td>2 Castle.Core</td>
<td>60</td>
<td>6119</td>
<td>50</td>
<td>171</td>
</tr>
<tr>
<td>3 Castle.Micro Kernel</td>
<td>112</td>
<td>11007</td>
<td>86</td>
<td>88</td>
</tr>
<tr>
<td>4 Castle.Windowsor</td>
<td>64</td>
<td>4240</td>
<td>34</td>
<td>92</td>
</tr>
<tr>
<td>5 NUnit.framework</td>
<td>40</td>
<td>4415</td>
<td>37</td>
<td>397</td>
</tr>
<tr>
<td>6 NUnit.mock</td>
<td>20</td>
<td>579</td>
<td>6</td>
<td>42</td>
</tr>
<tr>
<td>7 NUnit.util</td>
<td>88</td>
<td>6405</td>
<td>34</td>
<td>211</td>
</tr>
<tr>
<td>8 NUnit.uikit</td>
<td>352</td>
<td>7556</td>
<td>30</td>
<td>32</td>
</tr>
</tbody>
</table>
Results – number of mutants

The graph shows the number of mutants for different libraries and frameworks across various categories (EOC, IPC, JDC, JID, OMR). The Y-axis represents the number of mutants ranging from 0 to 250. The categories are represented by different colors:
- Castle.DynamicProxy.dll (blue)
- Castle.Core.dll (red)
- Castle.MicroKernel.dll (blue)
- Castle.Windsor.dll (purple)
- Nunit.framework.dll (square pattern)
- Nunit.mocks.dll (filled square)
- Nunit.util.dll (blue)
- Nunit.uiokit.dll (dashed line)

Each category has bars indicating the number of mutants for each library or framework.
Results – mutation score

![Mutation Score Graph]

- Castle.DynamicProxy.dll
- Castle.Core.dll
- Castle.Windsor.dll
- Castle.MicroKernel.dll
- NUnit.framework.dll
- NUnit.mocks.dll
- NUnit.util.dll
- NUnit.uikit.dll

Bar graph showing mutation scores for different components.
CREAM system

- Parser based CREAtor of Mutants
- Applies standard and object-oriented operators
- Uses compilation and reflection mechanisms
- Tests mutants with unit test frameworks
Comparison with CREAM 2.0

Generated and killed mutants

- **Mutants' number**
  - EOC
  - IPC
  - JID

- **CREAM generated**
- **ILMutator generated**
- **CREAM killed**
- **ILMutator killed**
Comparison with CREAM 2.0

Average mutant's generation time
(including compilation time for CREAM)

<table>
<thead>
<tr>
<th></th>
<th>EOC</th>
<th>IPC</th>
<th>JID</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREAM 2.0</td>
<td>13</td>
<td>13.45</td>
<td>12.68</td>
</tr>
<tr>
<td>ILMutator</td>
<td>0.27</td>
<td>0.3</td>
<td>0.22</td>
</tr>
</tbody>
</table>

seconds
Conclusions

- Introducing mutations on the intermediate language level – more efficient, faster
- Mutated program doesn’t have to be compiled
- Identification of mutation locations - more effort to implement
- Lack of compilation - necessity of correctness checking
Future work

- More mutation operators
- Other ways of generating and storing mutants (e.g. metamutant)
- Other methods of testing (not only unit tests)
- New versions of libraries (Mono.Cecil 0.9) or other libraries (Microsoft.CCI) for mutation injection
- Better visualization of mutated code (CIL<->C#)
- Identification of equivalent mutants
Q&A
CREAM – main window
CREAM – original and mutated code