dr Anna Derezińska, Piotr Trzpil
Institute of Computer Science
Faculty of Electronics and Information Technology
Warsaw University of Technology
e-mail A.Derezinska@ii.pw.edu.pl

Mutation testing of ASP.NET MVC

Abstract
Mutation testing deals with assessing and improving quality of a test suite for a computer program. The range and effectiveness of the method depends on the types of modifications injected by mutation operators. We have checked whether mutation testing technique can be used to evaluate test cases for ASP. NET MVC-based web applications. Several new specific mutation operators were created and discussed. The operator judgment was experimentally verified with the mutation tool implementing the operators in the Common Intermediate Language (CIL) of .NET. The results show that mutation testing can be successfully applied to an application running on a web server, but execution times of functional tests can be long.

Keywords: mutation testing, ASP .NET MVC, C#, Common Intermediate Language

Introduction

Mutation testing is a process that can be used to measure quality of a test suite for a computer program [4]. It is based on injecting deliberate mistakes into the application code and testing the modified program to gain information about insufficient and missing tests. Algorithms used to create modifications (mutation operators) can be devoted to general features of a programming language such as logical expressions, or object-oriented characteris-

A . Derezińska, P. Trzpil, Mutation testing of ASP.NET MVC, J. Swacha (Ed.) Advances in Software Development, Scientific Papers of the Polish Information Processing Society Scientific Council, Warsaw 2013, pp. 127-136

tics. However, specific application technology, such as web processing also requires comprehensive testing, which could be verified with the mutation approach. The ASP.NET MVC programming environment was chosen for evaluation. This framework is a set of libraries for creation of easily-tested web applications using the Model-View-Controller design pattern [5,11].

We proposed several specialized mutation operators that can be applied in the ASP.NET MVC applications at the Common Intermediate Language (CIL) code originated from the C# source code. The operators were implemented in the mutation tool and experimentally evaluated. In experiments two common methods of application testing were taken into account: unit tests and functional tests run in a web browser.

1. Related work

Mutation testing was applied for different general purpose languages as well as specific domain languages [4]. Mutation operators related to .NET platform were developed at two code levels, with changes provided into C# source code or into lower level of the Common Intermediate Language (CIL).

General purpose structural mutation operators are implemented in the Nester tool [9]. The simple C# code modification rules are defined in regular expressions or XML document and can result in invalid mutants. The tool is not further developed. PexMutator [10] cooperates with the Pex extension of the Microsoft Visual Studio. It injects several structural changes into Intermediate Language. The mutated code is verified with tests automatically generated by Pex. CREAM (CREAtor of Mutants) was the first mutation testing tool dealing with object-oriented mutation operators for C# programs [1,2]. Faults are injected into the C# code in the form of a syntax tree which is an output of the parser analysis. The current - third version supports 8 standard and 18 object-oriented mutation operators of C#. Mutations of Intermediate Language of .NET for programs originated from C# are introduced by the ILMutator prototype [3]. It implements 10 object-oriented and C# specific mutation operators.

Mutation testing was considered for web applications based on the ASP.NET Web Forms [6]. Though, applications using this former library have less test facilities and do not support the MVC pattern that is fundamental for

mutation operators aimed at ASP.NET MVC. Advantages and disadvantages of integration and unit testing of ASP.NET MVC are discussed in [12].

2. Mutation operators for ASP.NET MVC framework

The ASP.NET MVC framework is a set of libraries supporting building of highly testable Internet applications based on the MVC (Model-View-Controller) architectural pattern [5, 11]. It combines programming paradigms common to Ruby on Rails, such as conventions over configuration, model binding and code simplicity, with the ASP.NET web technology of Microsoft (running on .NET framework).

The MVC architectural pattern separates an application into three main components: the model, the view, and the controller. In the framework, URL requests are mapped to controller classes and their methods. The controller handles and responds to user input and interactions. The controller performs operations on the model, and forwards a response e.g. a view to the user. Action methods (also called 'actions') are controller methods that can handle HTTP requests. They are recognized by their return type – deriving from *ActionResult*. The platform manages and calls specific actions to handle incoming requests.

Views are components providing generic data for presentation of web pages. In the framework, views are files returned by controller actions. The files consist of HTML code, combined with the source code of an imperative language of .NET - usually C#.

Model objects implement the logic for the business data domain. They often cooperate with the data base that stores the model data.

Separation of components and loose coupling of controllers with the execution platform encourage application testability. In unit tests, we can create controller objects, call their methods and verify results.

Mutation operators devoted to selected features of a programming technology should take into account various criteria, such as:

- a place of a change can be easily identified in the code,
- a code modification can be straightforwardly realized,
- a modified code is not detected by all tests,

- a mutation mimics a mistake that can be commonly made by a software developer.

We propose six new mutation operators for ASP.NET MVC that can be implemented at the CIL level. Selected mutation operators are illustrated by examples in the C# code corresponding to actual CIL code on which the mutation operators operate. In other cases code examples are omitted due to brevity reasons. Full examples are available in the thesis [13]. The following sections present mutation operators grouped by area of application.

2.1. Modifications of Model Binding

Values of client requests can be automatically adjusted to action parameters. A request is passed to a method if its name is identical to the name of the action parameter.

CAPN - Change Action Parameter Name is a mutation operator that changes the name of an action parameter. The name is substituted by a dummy name such as “mutatedParameterName#”, where # stands for an order number (Listing 1). In consequence, a request value for the action parameter will not be found during a mutant execution, unless a default value was defined. The result of this mutation depends on the parameter type. If the parameter is of reference type, it will be set to *null* and will probably cause a fault of the method. In case of a value parameter, an exception will be raised immediately.

This mutation can be easily introduced in the intermediate language. It is more complicated when applied in the C# code due to usage of optional parameters. In C# the whole project has to be searched for occurrence of the method calls (expected in unit tests) in order to ensure a compliable code.

```
// Before mutation - C# code
public ActionResult Edit(string name)             { ... }

// After mutation - C# code
public ActionResult Edit(string mutatedParameterName1)   { ... }
```

Listing 1. Example of CAPN operator - Change Action Parameter Name

2.2. Modifications of Action Attributes

There are two kinds of C# attributes that are placed before action methods: method selectors and filters. A programmer can use attributes delivered by the platform or create their own attributes.

Method selectors are used for identification of an action which will be executed after a request delivery. One of such attributes is *ActionNameAttribute* that changes a default action name, which is the name of a method, into a given name.

Filters make actions to be constrained with additional restrictions. Filter attributes can be placed before a controller class, thus influencing all actions of the controller. Among other filters of the framework, we can use *AuthorizeAttribute* for an action that has to be authorized, or *HandleErrorAttribute* stating what should be done when an exception was raised.

Attributes have influence on application execution only if it is executed on a server. Therefore the most obvious tests that verify usage of attributes are functional tests run in a web browser. Using unit tests a presence and a state of an attribute can be verified.

SWAN - Swap Action Names could be a mutation operator that swaps names of two actions through interchange of *ActionName* attributes. In result, in all cases when one action should be executed another action is raised. In order to have a consistent code, both actions should have the same number of parameters of the same types. Moreover, action names can be checked by a compiler, e.g. while calling *RedirectToAction* method, and the mistake can be easily detected.

RAAT - Remove Authorize Attribute - is a mutation operator that removes *Authorize* attribute placed before an action or a controller. Therefore the action or all actions of the controller can be called by an anonymous client.

This mutation checks an important feature of an application concerning its security. In many programs, it is easy to be applied both in C# and CIL code. However, *Authorize* attribute can be extended by inheritance with additional functionality or other authorization policy. In such cases the removal of the attribute should be waived.

2.3. Modifications of Action Results

An action of a controller returns a value describing a server answer to a client request. There are different types of such answers inherited from the *ActionResult* class, for example: *ViewResult* - a view is generated, *RedirectResult* - redirection of a client to another address, *JsonResult* - a return value is in JavaScript Object Notation, *FileResult* - a file is returned. Methods of controller support creating of these answers.

An application changes its behavior if a value returned by an action is modified. The mutation is limited for the cases when the return value inherits from the *ActionResult* class, which is a typical solution.

RVRA - Replace View with RedirectToAction - is a mutation operator that changes an object returned by a controller action; *RedirectToActionResult* is returned instead of *ViewResult* (Listing 2). The mutation can be detected by tests that check a type of an object returned by an action.

```
// Before mutation - C# code
public ActionResult ViewOrRedirect(object obj)
{   return base.View(obj);    }

// After mutation - C# code
public ActionResult ViewOrRedirect(object obj)
{   return base.RedirectToAction("Index");    }
```

Listing 2. Example of RVRA operator - Replace View with RedirectToAction

CRAT - Change RedirectToAction Target - is a mutation operator that changes a target action being a redirection method call parameter. The mutation can be implemented by substitution of a string identifying a target action.

In the selected solution the name is substituted by a dummy action "*MutatedIrrelevantActionName*". Usage of a dummy action is easy to be implemented and it is irrelevant whether the action redirected to exists or not. The action will not be found and will cause an error when run on a web server. However in unit tests this will not be the case and a user must check the *ActionResult* object for valid action name.

2.4. Modifications of Route Mapping

URL routing is used for mapping incoming URL requests to the appropriate controllers and their actions. The routing engine parses variables defined in the URL and the framework passes the parameter values to the controllers.

CMRA - Change MapRoute Address Pattern - is a mutation operator that changes an URL address. The string defining the URL pattern is substituted by a dummy one, e.g. "*MutatedString*". Therefore the route will be not corresponding to any incoming request. One of other existing routes will be used and as a result the appropriate controller might not be found.

The basic rule of the mutation is easily implemented. However, there are many overloaded forms of the *MapRoute* method. Extension of the mutation operator to all of them requires investigation of many possible parameter combinations. The CMRA operator is reasonable for bigger projects with many routes applied. In a small project a routing mistake can be easily detected by a developer.

3. Experimental evaluation of ASP.NET MVC mutation operators

Mutation experiments on the above discussed mutation operators were performed with the VisualMutator tool [13]. This tool was developed as a Visual Studio extension and provides an expansible framework for mutation testing at the CIL level. Tight coupling with the Visual Studio development framework makes the mutation testing process efficient, as the program under test is compiled only once and mutants can be generated fast.

Two subjects based on the ASP.NET MVC platform were evaluated in experiments (Tab. 1). Their open source code is available on the codeplex.com service. The first subject is NerdDinner [8] - an open source project that helps Internet people plan get-togethers. It utilizes the authorization system based on the Open ID standard and local accounts. The application also uses Bing search engine, geolocation and RSS feeds. NerDinner is distributed with a set of unit tests. The second subject of experiments is MVC Music Store [7] - a store which sells music albums online. This application was tested with functional test cases that run in a web browser implemented as control instructions of the WebDriver library.

The basic metrics of the applications and their test cases are summarized in Tab. 1. The metrics were measured with the NDepend tool. In Music Store, big samples of exemplary data included in the program were omitted.

Table 1. Subjects of experiments

	NerdDinner		Music Store	
	Applic.	Test cases	Applic.	Test cases
LOC without comments	730	461	195	61
Type number	71	20	27	2
Method number	399	156	172	17

Results of the test ability to detect faults injected by the mutation operators are shown in Tab. 2. In case of NerdDinner mutants of only two operators were killed by unit tests. Operators RVRA and CRAT modify results returned by actions, which is usually covered by unit tests. Equally important is verification of route mapping (CMRA) that is not covered by the tests designed for the application. Other mutants are not easily killed by unit tests unless the reflection mechanism was applied.

Tests of Music Store run were more effective in killing mutants. They required less code (Tab.1) but were run in the web browser and took more time. An average test time of a mutant was equal 1.9 s for NerdDinner with unit tests run with NUnit, whereas 26.2 s for Music Store mutants run in the ASP.NET Development Server and functional tests executed with the assistance of VS MsTest.

Table 2. Mutation testing results

Mutation operators	NerdDinner		Music Store	
	mutant number	killed	mutant number	killed
CAPN	25	0	14	7
RVRA	37	22	18	6
CRAT	11	5	10	3
SWAN	15	0	8	6
RAAT	13	0	4	1
CMRA	3	0	1	1
Sum	104	27	55	24

Conclusions

We have shown how the mutation testing approach can be applied for the ASP.NET MVC-based web services.

Efficiency of a unit test suite in the respect of the considered mechanism verification was not very high (mutation score about 26%). However, it is difficult to cover by unit tests all mechanisms utilized by an application run on a server. Better mutation results (44%) with mutants killed of all fault types gave test cases run in a web browser but their execution times were significantly longer.

In many cases, the functionality of presented operators can be approximated by standard and object mutation operators for C# language. However it can be assumed that part of possible programmer error space will not be covered in that case, due to differences of ASP.NET MVC-based application and standard desktop application. The specific operators for the platform should operate on higher level of abstraction, making use of concepts of the platform and the language. This puts them to good use along standard and object operators. Nevertheless, usability of each operator should be analyzed to avoid duplicating functionality of classic operators.

Some faults, as e.g. injected by RAAT operator, can be easily detected by test cases run in a web browser. On the other hand simple unit tests do not kill such mutants, which might be treated as equivalent in their context. These mutants can be killed by unit tests with the usage of meta-programming techniques, such as reflection, that allow investigating and modifying a program during its run. An open question remains whether an application has to be run on a web server or should we accept usage of meta-programming in unit tests. In the first case, the long execution time might make the entire process impossible to use efficiently with large number of tests or mutants. In the latter case, interesting consequences of such a decision emerge. The mutant equivalence is then relative, depending on the testing approach. If we allow the usage of meta-programming techniques, no mutant with changed code can be considered equivalent, as the modification can always be detected by static analysis and not program behavior.

The VisualMutator tool is currently extended with selected standard and object-oriented mutation operators for C# language. It is also planned to be used in evaluation of automatically developed mutation-based test cases.

Bibliography

1. CREAM - Creator of Mutants, http://galera.ii.pw.edu.pl/~adr/CREAM/ [access: 2013]
2. Derezińska A., Szustek A.: *Object-oriented testing capabilities and performance evaluation of the C# mutation system,* Szmuc T., Szpyrka M., Zendulka J. (eds.), LNCS, vol. 7054, Springer, 2012, pp. 229-242
3. Derezińska A. Kowalski K.: *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
4. Jia Y., Harman M.: *An analysis and survey of the development of mutation testing,* IEEE Transactions of Software Engineering, Vol. 37, No. 5 Sep/Oct 2011, pp. 649-678
5. Madeyski L., Stochmialek M.: *Architectural design of modern web applications,* Foundations of Computing and Decision Sciences, Vol. 30, No 1, 2005, pp. 49-60
6. Mansour N., Houri M.: *Testing web applications,* Information and Software Technology, vol. 48, issue 1, Jan 2006, pp. 31-42
7. MVC Music Store, http://www.asp.net/mvc/tutorials/mvc-music-store
8. NerdDinner, http://nerddinner.com [access: 2013]
9. Nester, http://nester.sourceforge.net [access: 2013]
10. PexMutator, http://pexase.codeplex.com/wikipage?title=PexMutator [access: 2013]
11. Sanderson S.: *Pro ASP.NET MVC 2,* Apress, 2010
12. Smirnow A.: *Automated testing of ASP.NET MVC applications*, Methods & Tools, Vol. 20, No 1, 2012, pp. 13-18
13. Trzpil P.: *Mutation testing in ASP.NET MVC,* Bach. Thesis, Institute of Computer Science, Warsaw University of Technology, 2012