Migration of Unit Tests of C# Programs

Anna Derezińska[0000-0001-8792-203X] and Sofia Krutko

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

Abstract. Maintenance of a project with unit tests could require moving from one test platform to another. It can be established by an automated test re-generation or by transformation of the test code. The latter is recommended in case of a high quality test set. Preservation of the intrinsic knowledge introduced by the test developers could also be of high importance. NUnit and MSTest of Visual Studio are among the most popular unit test frameworks for C# programs. The ability of test transformation from NUnit to MSTest has been investigated. Transformation rules were implemented in a prototype tool integrated with Visual Studio, but only a subset of possible constructions have their straightforward equivalents. Experiments confirmed the potential benefits of the approach, but also limitations of the target tests.

Keywords: Unit Test Migration, Test Maintenance, C#, NUnit, MS Test, Legacy Code.

1 Introduction

Creation of unit tests is an important task in program development [1,2]. Unit tests are regarded as a specification notion, e.g., in test-driven development [3], agile approaches, and other methodologies. Maintenance of long-living software requires keeping up with changes of a production code and modifications of an environment, which also refer to unit tests associated with the code.

Tests of C# programs could be prepared using many platforms. NUnit [4], developed similarly to JUnit for Java, has been rewritten as a specific .NET solution since its 3rd version. This free tool has still been widely used and counted as the best unit testing tool in 2020 according to the Software Testing Help service [5]. Testing in the .NET environment [6] has also been supported by MS Visual Studio Testing Framework [7], which includes MSTest to prepare and run unit-like tests. Tests using both platforms are often applied in real-word programs, while NUnit has a longer history and offers more capabilities. The testing support of MS VS has been steadily extended and MSTest benefits from the tight integration with the development environment.

As no definite unit test leader for C# programs exists, there could be different reasons for migrating tests from one platform to another, such as: client requests, code reuse in another project, organizational changes, compatibility requirements for code integration or outsourcing, reducing of the number of platforms to be supported in an enterprise, etc. In general, there are two basic approaches to cope with this problem:

1. Automatic generation of test cases for the target test platform; the tests are based on the given application code.
2. Transformation of test cases from one test platform to another.

Both approaches have their pros and contra. Automatic tests could meet different criteria and show the high ability to detect faults [8]. On the other hand, generated tests could not take into account specific domain or logical constrains. If a project is accompanied with a set of high-quality tests or if manually tests contain a valuable specification and expert experience, there could be worthwhile to transform the existing tests into the tests of a target platform.

In this paper, we deal with unit test migration from NUnit to MSTest. The main contributions of the work are: (i) preparation of the transformation rules taking into account all NUnit attributes and possible assertion structures, (ii) implementation of a prototype tool integrated with MS VS, (iii) experimental evaluation of the approach.

The paper is organized as follows. The next Section describes the background and related work. Basic issues of the test migration are reported in Sec. 3. Tool support and experiments are discussed in Sec. 4. Finally, Sec. 5 concludes the paper.

2 Background and Related Work

Preparation and maintenance of unit tests is a labor-intensive activity. Therefore, a lot of research has been done on their automation. Test generation has achieved promising advances [8]. However, maintaining generated tests can take more time, and using test generation tools could result in creating more tests than manually [9].

Software maintenance refers not only to a production code but also to various kinds of its tests. Test cases can be treated in different ways. In disposable testing, tests are not maintained but substituted by automatically generated tests from the code [10]. Comparison of the latter and former tests is recommended to inspect the tests, reveal some changes, and throw away unnecessary tests. Broken tests that do not compile can be treated by a test repair procedure. Analyzing of a data-flow graph has been applied to rebuild tests that preserve the intent of the original tests [11].

However, testing activities cannot only be restricted to automation [12]. The most important thing in testing is thinking, analyzing, and creating good effective tests. Hence, having such tests we want to keep them going in the future software revisions.

Not all kinds of tests are equally worthwhile to be maintained. Automatically generated tests are often successful in fulfilling different coverage conditions or detecting given classes of faults, but could be not very realistic and hardly to read and comprehend [9]. Initial high-quality tests from test-driven development could be treated as a program specification, but sometimes TDD could result in poorly maintainable tests.

Maintenance of tests could be indispensable in legacy systems. Although some authors treat legacy code as a code without tests that need to be supplemented [13], in general, legacy systems could have long live cycles, a degraded structure, lack of documentation, but implement an important business logic. The experiments showed that the lack of unit tests created at the beginning provides to a hardly testable code. Efficiency of manual tests could be complemented with automatically generated tests, giving the best results for the hybrid solutions. Therefore, it could be profitable to retain existing tests, but the decision should be based on a cost analysis [14]. Addressing 10-20 year live cycles, as many government and military programs live, there should be decided whether to replace or maintain different types of test platforms.

Most of the work discussed above has been performed for Java programs. A set of guidelines for preparing readable and maintainable unit tests and working with legacy code in C# can be found in [15]. Building of integration tests for .NET platform has been reported in [16]. This approach is also based on the code analysis provided by Roslyn [17], but the ideas have not been realized yet. However, this direction could be combined with the test migration support to enhance the test creation.

Necessity of test migration can be motivated by different factors. Tests are run many times, need to be looked at and modified. Therefore, except of the efficiency in detecting errors, also readability or performance are the important obligations [2]. We could be interested in the preservation of the good features of a test set.

The problem of test migration between mobile applications has been addressed in [18]. GUI-based test cases written using the Espresso testing framework are analyzed and categorized according to different GUI elements. The tests of an Android application are adopted to another one that shares a common functionality.

To the best of our knowledge, migration of C# unit tests and automation of such a process has not been reported so far.

3 Migration of NUnit Tests into MS Test

Translation of unit tests requires modification of attributes, test code, and assertions. Directives (e.g., *using*) will also be converted accordingly.

3.1 Attributes

Attributes, a kind of metadata, are used in unit tests for specification of test cases and description of the environment behavior during test execution. NUnit uses more than 40 attributes to manage assemblies, classes, and test methods. Attributes associated with their parameters are used to control threads, execution time, value of test parameters, and other test features.

Translation of tests with all NUnit attributes has been analyzed. Only part of these attributes have their equivalents in MSTest. Functionality of another part of the attributes could be fully or partially realized using other MSTest mechanisms. The remaining attributes have no support in the structures of MSTest.

Detailed translation rules for all attributes of NUnit have been proposed. These rules could be clustered into translation schemata that refer to the attribute processing and modification of the corresponding test code. Six translation schemata have been identified (**Table 1**). The attributes of a schema are listed in the last column.

Table 1. Attribute translation schemata.

Schema	Description	Attributes
TE-NM	Attribute is Equivalently Translated and test code Not Modified	Category, Description –for method, Ignore, LevelOfParallelism, MaxTime, NonParallelizable, Paralleizable – for assembly, Property, Range, Sequential, SetUp, TearDown, Test, TestCaseSource, TestFixture, TestOf, Values
TE-M	Attribute is Equivalently Translated and test code Modified	Author, OneTimeSetUp, OneTimeTearDown, TestCase, TestFixtureSetup, TestFixtureTeardown
TP-NM	Attribute is Partially Translated and test code Not Modified	Combinatorial, Explicit, Pairwise
R-NM	Attribute is Removed and test code Not Modified	Apartment, Description – for class or assembly, NonTestAssembly, Paralleizable – for method or class, Repeat, Retry
R-M	Attribute is Removed and test code Modified	DefaultFloatingPointTolerance, SetCulture, SetUICulture
C-C	Attribute and test code are converted into Comments	Culture, Theory (Datapoint, DatapointSource), Order, Platform, Random, RequiresThread, SetUpFixture, SingleThreaded, TestFixtureSource, Theory, ValueSource

An attribute can be substituted by another one, removed, or converted into a comment. If an attribute is translated, its name is changed accordingly, and attribute parameters are modified, or additional attributes are added if necessary. In some cases, the equivalent attribute can encompass a broader or a narrower domain, or additional constraints have to be met. There could also be some related attributes. In these cases a sub-dependent attribute can be handled accordingly to the main attribute. They are shown in parentheses in the list of attributes in (Table 1).

An attribute could also be removed from the target test, usually when there is no equivalent attribute and the removal has no influence on the test result.

Sometimes, the functionality of the attribute cannot be reflected, and the attribute has an impact on the test result. In those cases, the attribute and the test will be not deleted but converted into a comment. Such commented tests will be omitted in the target test suite or could be manually adopted towards the desired notation.

Apart from attributes, the test code could be changed. The main modifications refer to the addition of a piece of code to the method or to a class under concern.

3.2 Assertions

There are two approaches to specifying assertions in NUnit: Classical Model and Constraint Model.

In the classical model, separate methods are used to verify different constrains concerning a unit under test and its behavior. The methods are collected in the following classes: *Assert*, *StringAssert*, *CollectionAssert*, *FileAssert*, and *DirectoryAssert*. Many commonly used methods have their equivalents in MSTest (**Table 2**). These methods will be translated and their parameters adjusted accordingly, e.g., swapped parameters in the methods from the StringAssert class. The remaining methods are not straightforwardly translatable and would be converted into comments.

Table 2. Methods from the NUnit classical model that have equivalents in MSTest.

Class	Methods
Assert	True, False, Null, NotNull, AreEqual, AreNotEqual, AreSame, AreNotSame, Pass, Fail, Ignore, Inconclusive
StringAssert	StartsWith, EndsWith
CollectionAssert	AllItemsAreInstancesOfType, AllItemsAreNotNull, AllItemsAreUnique, AreEqual, AreEquivalent, AreNotEquivalent, Contains, DoesNotContain, IsSubsetOf, IsNotSubsetOf, IsSupersetOf, IsNotSupersetOf

Constraint model is an alternative approach to express conditions to be verified. All assertions are specified with one method *Assert.That*. The first parameter of the method takes some data to be verified, and the second parameter represents the assertion logic. For example, the following assertion checks whether the given string variable includes the requested text.

```
Assert.That(someString, Is.EqualTo("Hello"));
```

The constraint model is recommended to be used in NUnit. The *Assert.That* method can be overloaded in a hundred ways, therefore, many assertions can be specified. However, MSTest supports only a classical model of assertions. Therefore, only a subset of assertions of the constraint model will be translated into their equivalents (**Table 3**). The remaining assertions will be commented.

Table 3. A subset of constraint model assertions (NUnit) and their equivalents (MSTest).

NUnit	MSTest
Assert.That([…], Is.EqualTo([…]))	Assert.AreEqual([…],[…])
Assert.That([…], Is.Not.EqualTo([…]))	Assert.AreNotEqual([…],[…])
Assert.That([…], Is.Null)	Assert.IsNulll([…])
Assert.That([…], Is.Not.Null)	Assert.IsNotNulll([…])
Assert.That([…], Is.True)	Assert.IsTue([…])
Assert.That([…], Is.False)	Assert.IsFalse([…])

A simple example illustrates the translation of attributes of TE-NM and TE-M schemata.

NUnit test method (before translation):

```
[TestCase(12, 3, ExpectedResult = 4)]
[Category("TestCaseCategory")]
public int DivideTest(int n, ind d)
{   return n/d; }
```

MSTest test method (after translation) with an assertion that has been added:

```
[DatRow(12, 3, 4)]
[TestCategory("TestCaseCategory")]
[TestMethod]
public void DivideTest(int n, ind d, int expectedResult)
{   Assert.AreEqual(n / d, expectedResult); }
```

4 Experimental Evaluation of Test Migration

We describe now experiments on the automated migration of unit tests of C# programs from the NUnit (version 3) to MSTest platform (version 2).

4.1 Test Translator – Automated Support for Test Migration

Experiments on the test migration from NUnit to MSTest have been supported by a prototype Test Translator [19]. The tool is integrated with the MS Visual Studio, as its extension. The translation engine uses the .NET Compiler Platform (Roslyn) [17] to handle the code in the form of an Abstract Syntax Tree.

Processing of the original unit tests consists of two phases: analysis and transformation. During the analysis phase, the code is scanned and expected modifications are recognized. At first, modifications are specified that refer to the whole set of unit tests. They could correspond to attributes of the whole assembly, *using* directives, etc. Then, all test classes are analyzed and their transformations collected, e.g., modifications of class attributes, modification of class declaration. Finally, the test methods are examined and their transformations identified, which could modify the method attributes, parameters, and code.

During the transformation phase, all previously specified modifications are performed and the target tests are generated.

4.2 Subject Programs

The experiments have been conducted on three programs of different origin and goals. The first one, *Benchmark*, was a simple program with a comprehensive test set that was developed to cover all attributes of NUnit and a variety of assertion structures. The second subject was an open source code, a part of *OpenRA* [20] game engine with tests prepared and published by their developers [21]. The third example, *IntroToNUnit* [22], was a loan handling program. It was aimed at learning testing with NUnit, in particular, using assertions in the constraint model [23]. The basic complexity metrics of the programs are given in **Table 4**, where:

- *LSC, Lines of Source Code* – the number of source code lines including comments and blank lines.
- *LEC, Lines of Executable Code* – the number of operations in the executable code.
- *NC, Number of Classes* – the number of classes in a project or assembly calculated as the object number of the *ClassDeclarationSyntax* type.

Table 4. Characteristics of the subject programs.

Programs	LSC	LEC	NC
1) Benchmark	15	1	1
2) OpenRA (classes of Game and Common modules)	4802	1354	35
3) IntroToNUnit	270	60	8

4.3 Results of Test Migration

While using TestTranslator, the sets of tests accompanied with the subject programs have been translated from NUnit into MSTest. The programs were tested with both variants of the test sets and the corresponding code coverage checked. Software metrics of the tests are given in **Table 5**. The consecutive rows correspond to subjects 1)-3) and their tests before (NUnit) and after (MSTest) translation. Apart from the metrics specified in the previous section, the following values have been provided:

- *NM, Number of Methods* **-** the number of method declarations in a project or assembly calculated as the object number of the *MethodDeclarationSyntax* type.
- *NA, Number of Assertions* – the number of assertions used in a project or assembly, calculated as the object number of the *InvocationExpressionSyntax* type included in the classes with an assertion token.
- *NTC, Number of Test Cases* – the sum of the number of nonparametric test methods plus the number of input parameter sets for the parametric test methods.
- *CA, Code Coverage* – the percentage of code lines covered by the unit tests during a program execution.

Table 5. Comparison of the software metrics of the tests before and after translation.

Test platform	LSC	LEC	NC	NM	NA	NTC	CA
1) NUnit	629	189	18	63	101	90	60.0%
1) MSTest	649	127	15	48	57	51	60.0%
2) NUnit	1330	672	19	63	164	55	54.2%

2) MSTest	1396	601	19	63	103	55	47.1%
3) NUnit	431	118	7	32	39	34	55.8%
3) MSTest	431	61	7	28	12	23	32.6%

The metrics change is summarized in **Table 6**. It shows the relation of a metric after translation to its value before translation, in percent. A result of 100% denotes no change in the values, more than 100% stands for the increase, and less than 100% for the decrease of the corresponding metrics.

Table 6. Change of test characteristics after test translation [in %].

Program	LSC	LEC	NC	NM	NA	NTC	CA
1) Benchmark	103.2	67.2	83.3	76.2	56.4	56.7	100
2) OpenRA	105	89.4	100	100	62.8	100	87
3) IntroToNUnit	100	51.7	100	87.5	30.8	35.3	58.4

4.4 Discussion of Results

The subject programs have different characteristics and various goals, hence their test results are diverse.

The Benchmark program (1) was intended to use all NUnit elements that could be modified during the test translation. It included attributes from all translation schemata as well as assertions written in the classic but also in the constraint model. In result, 3 classes, 15 methods, and 44 assertions were not converted into the corresponding target units. Hence, the number of code lines executed in the tests (LEC) was lowered. The number of source code lines (LSC) was not lowered, as all not translated units remained as comments. The remaining test parts were transformed according to the given rules, and the outcome of all translated tests was the same as the outcome of the original tests. The number of test cases lowered, but as some tests referred to the same code units, the coverage measures obtained for both test sets were identical.

Results of the Benchmark test translation have been used for verification of the approach and the translation rules in particular. Though, it was intentionally an artificial program and its results could not be generalized to usual application programs.

The modules and tests of OpenRA (2) have been prepared for purposes of the application development, independently of the experiment under concern. It can be observed that all test classes and test methods have been translated and the number of test cases remained unchanged (100% change of NC, NM, and NTC).

Unit tests of OpenRA consisted mainly of test methods with a single attribute - *TestCase*. This attribute could be used for passing argument values to a parametrized test method. However, in this program, the test methods were not parametrized. The *TestCase* attribute served as an indicator of a test method and was associated with an argument with a test name. Therefore, after translation, this attribute was converted into two attributes: *TestMethod* - to point at the test method and *TestProperty* which stores a test name. This is also a main cause of the rise in the number of code lines (LSC) after the test translation.

In OpenRA tests, a variety of assertions from the classical and constraint models have been used. Most of them were translated, but about 60 had no equivalents in MSTest (drop in NA) and were commented. Therefore, these tests could have required additional effort to get the complete original tests.

A special case concerned six assertions of the constraint model that changed test outcomes and caused the tests to fail. There were assertions structured as `Assert.That([..], Is.EqualTo([..]))` that were used for comparing collections. They were directly transformed into assertions of the form `Assert.AreEgual.` This assertion checked whether its arguments are equal, while the tests needed comparison of the collection content, and therefore, application of the `CollectionAssert.AreEqual` assertions. This modification was not anticipated before, and would have been handled automatically only after a successful identification of the cases during the test code analysis.

Unit tests of the third program, IntroToNUnit, were intentionally designed to present various kinds of parameterized NUnit tests and were mainly based on the constraint model of assertions. They used many mechanisms that are not implemented in MSTest. Therefore, the number of executable code lines lowered significantly. The number of target assertions also decreased considerably.

5 Conclusions

Automating of a test migration can assist the test maintenance, especially for long-living systems. This problem has been considered in the context of C# programs with original NUnit tests and target MSTest. A set of translations was developed, taking into account attributes and assertions that could encounter in a testing code.

Application of the tool support gave promising results, while all test classes, test methods, and test cases of a part of the production application OpenRA were successfully transformed. About 87% of the statements covered by the original tests remained also covered by the target ones. However, the lack of many assertion mechanisms in the target notation of unit tests caused missing almost 40% of the assertions.

The translation results highly depend on the kind of tests used in an original program. In the case of tests based on the assertion constraint model, the straightforward translation not always could be applicable. Consequently, in an application of such kind, only about 30% of assertions and 35% of test cases were transformed and 58% of the code lines stayed covered by tests.

Although the number of subject programs was too small to generalize the outcomes, we could observe the benefits of the migration automation and the drawbacks due to non-transformed or partially transformed test cases. The approach could be further enhanced with a more comprehended analysis of the test code. It would allow translating more kinds of assertions. Furthermore, the approach could be aimed at different platforms, such as xUnit.net; and/or combined with the automated test generation, which could supplement missing test cases according to the given criteria.

References

1. Daka, E., Fraser, G.: A survey on unit testing practices and problems. In: IEEE 25th International Symposium on Software Reliability Engineering, pp. 201-211. IEEE Comp. Soc. (2014). doi: 10.1109/ISSRE.2014.11
2. Fields, J.: Working effectively with unit tests. Leanpub (2014)
3. Beck, K.: Test Driven Development: by Example. Addison-Wesley Professional (2002).
4. NUnit, https://nunit.org/, last accessed 2020/12/28.
5. 20 most popular Unit testing tools in 2020, https://www.softwaretestinghelp.com/unit-testing-tools/, last accessed 2020/12/28.
6. Ritchie, S.: Pro .NET best practices. Apress (2011)
7. MS Test V2 framework, https://github.com/microsoft/testfx, last accessed 2020/12/28.
8. Ramler, R., Klammer, C., Buchgeher, G.: Applying automated test case generation in industry: a retrospective. In: International Conference on Software Testing, Verification and Validation Workshops, pp. 364-369, IEEE (2018). doi: 10.1109/ICSTW.2018.00074
9. Shamshiri, S., Rojas, J. M., Galeotti, J. P., Walinshaw, N., Fraser, G.: How do automatically generated unit tests influence software maintenance? In: 11th International Conference on Software Testing, Verification and Validation, pp.250-261. IEEE Comp. Soc. (2018). doi: 10.1109/ICST.2018.00033
10. Shamshiri, S., Campos, J., Fraser, G., McMinn, P.: Disposable testing: avoiding maintenance of generated unit tests by throwing them away. In: 39th International Conference on Software Engineering, pp.207-209, IEEE/ACM (2017). doi: 10.1109/ICSE-C.2017.100
11. Li, X., d'Amorim, M., Orso, A.: Intent-preserving test repair. In: 12th IEEE Conference on Software Testing, Validation and Verification (ICST), Xi'an, China, pp. 217-227, IEEE Comp. Soc. (2019). doi: 10.1109/ICST.2019.00030.
12. Roman, A.: Thinking-driven testing. Springer Inter. Pub., Cham (2018). doi: 10.1007/978-3-319-73195-7
13. Ramler, R., Winkler, D., Schmidt, M.: Random test case generation and manual unit testing: substitute or complement in retrofitting tests for legacy code? In:38th Euromicro Conference on Software Engineering and Advanced Applications, Cesme, Izmir, pp. 286-293, IEEE Comp. Soc. (2012). doi: 10.1109/SEAA.2012.42
14. Kent, J., Dewey, M.: Legacy test systems — Replace or maintain. 2016 AUTOTESTCON, Anaheim, CA, 2016, pp. 1-5, IEEE (2016). doi: 10.1109/AUTEST.2016.7589645
15. Osherove, R.: The art of unit testing with examples in C#. 2nd ed. Manning Publications Co., Shelter Island, NY (2014)
16. Saadatmand, M.: Towards automating integration testing of .NET applications using Roslyn. In: International Conference on Software Quality, Reliability and Security (Companion Volume), pp. 573-574, IEEE Comp. Soc. (2017). doi: 10.1109/QRS-C.2017.99
17. Roslyn .NET compiler, https://github.com/dotnet/roslyn, last accessed 2020/12/28.
18. Behrang, F., Orso, A.: Test migration between mobile apps with similar functionality. In: 34th International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 2019, pp. 54-65, IEEE/ACM (2019). doi: 10.1109/ASE.2019.00016
19. Krutko, S.: Translation of NUnit tests to MSTest in C# programs. Bachelor Thesis, Warsaw University of Technology (2020) (in polish)
20. OpenRA – real-time game engine, http://www.openra.net/, last accessed 2021/01/03.
21. OpenRA code, https://github.com/OpenRA/OpenRA, last accessed 2020/03/28.
22. Introduction to .NET Testing with NUnit3, https://www.pluralsight.com/courses/nunit-3-dotnet-testing-introduction, last accessed 2021/01/03.
23. IntroToNUnit, https://github.com/irmoralesb/IntroToNUnit, last accessed 2020/04/22.