Coverage Guided Multiple Base Choice Testing

Tugkan Tuglular
Dept. of Computer Engineering
Izmir Institute of Technology
Izmir, Turkey
tugkantuglular@iyte.edu.tr

Onur Leblebici
Univera, Inc.
Izmir, Turkey
onur.leblebici@univera.com.tr

***Abstract*—A coverage guided input domain testing approach is presented with a feedback loop-controlled testing workflow and a tool is developed to support this workflow. Multiple base choices coverage criterion (MBCC) is chosen for systematic unit test generation in the proposed approach and branch coverage information is utilized as feedback to improve selection of bases, which results in improved branch coverage. The proposed workflow is supported with the tool designed and developed for coverage guided MBCC-based unit testing.**

Keywords— unit testing, input domain testing, multiple base choices coverage criterion, branch coverage criterion, feedback guided testing

I. Introduction

Developing unit tests is usually one of the responsibilities of software developers, who do not necessarily have required knowledge to design test cases and convert them to concrete unit tests. However, developers easily learn how to run unit tests and check their coverage using integrated development environment (IDE).

In this process, there are two incidents that the developers need to deal with. First, when they see a failed test, they start debugging with fault localization. Although name of the failed unit test helps a lot in such a case, naming unit tests is usually not done as good as they should be. Some companies, such as Google, have their own unit test naming standard, which is enforced through peer reviews, but this is not the case for most of small and medium software companies. Second, when they see a lower coverage than the company's expected standard, they start looking for ways to improve coverage.

For the problems depicted here, there is a need for a clear workflow supported with some automation. The proposed approach aims to solve these problems through the following:

- systematic unit test generation in such a way that rework for improvement is easy and effective
- naming unit tests so that fault localization is fast
- improving unit test coverage via a feedback-based gray-box technique

The proposed method is a gray-box approach as a mixture of input domain testing and path testing with branch coverage. Input domain testing is a black-box technique, where input parameters to a software under test (SUT) are determined along with their equivalence classes using domain knowledge. In the proposed approach, SUT is a method of an object-oriented class and unit tests are generated at the granularity of methods. Multiple base choices coverage (MBCC) [1] criterion is chosen for systematic unit test generation in the proposed approach. Once the input domain parameters with their values representing equivalence classes are determined, the developer decides on the base choices with respect to MBCC. Then, unit test inputs are automatically generated along with test names. Afterwards, the developer fills expected outputs of unit test. Although this step can be improved by model-based oracles, it requires modeling knowledge, experience, and tool support, which are not available in most of small and medium software companies.

Branch coverage is a testing criterion for path testing, which is a white-box technique. Current IDEs along with the unit test frameworks provide branch coverage values for each method under test (MUT). Although branch coverage values can be obtained easily, tool support for utilizing these values to improve coverage is limited. Current IDEs only highlight the blocks that are not covered. The proposed approach enables developers to connect not covered blocks with input domain parameters and to improve coverage by reworking on the multiple base choices.

The novelty of the proposed approach lies in the tool supported workflow that helps unit test developers to achieve high branch coverage through domain testing. This gray-box approach has a feedback loop, where lower value for branch coverage triggers developer for better choices in MBCC, which results in better branch coverage values. This loop continues until the expected branch coverage is reached. The proposed approach allows developers or companies to set their desired or expected branch coverage percentage. Setting coverage level is a cost/benefit trade-off. In the examples, we set expected branch coverage level to 95%. The coverage guided MBCC-based unit testing approach provides efficient and systematic unit testing process for developers lacking test case design knowledge and experience. For such developers, the proposed approach helps faster achievement of desired level of coverage criterion. The end result for companies employing those type of developers is software with higher quality and efficient use of developer time. Moreover, the tool enables developers to save design knowledge of unit tests with its archival feature.

The paper is organized as follows. After the fundamentals and related work sections, the proposed approach is explained in Section IV. Section V presents the developed tool for the proposed approach with a running example along with the results obtained. In Section VI experiences with the proposed approach and tool are discussed. Section VII concludes the paper and lists possible future work.

II. Fundamentals

Input domain testing is a black box testing approach, where the code for SUT is unknown and only domain information is used to develop test cases. It requires partitioning input space into equivalence classes for each test input parameter or variable and then selecting values from each equivalence class to form test cases [2]. If there are more than one input parameter, then the question "how should we

978-1-7281-8915-4/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS-C51114.2020.00020

consider combinations among equivalence classes or partitions?" arises. The terms "equivalence class" and "partition" are used interchangeably.

If cartesian product of all partitions from all test input parameters are considered, this approach is called strong equivalence class testing [2] or all combinations coverage criterion [1]. If only one partition from each test input parameter used in test case development, this approach is called weak equivalence class testing [2] or each choice coverage criterion [1]. When all combinations coverage criterion is the top element and each choice coverage criterion is the bottom element in subsumption relations among choice-based input space partitioning criteria given in Fig.1, multiple base coverage and base coverage criteria exist in this order [1].

In base choice coverage criterion, a base choice partition is chosen for each test input parameter, and tests are prepared by holding all but one base choice constant and using each non-base choice in each other test input parameter [1]. In MBCC, multiple base choice partitions are chosen for each test input parameter, and tests are formed by holding all but one base choice constant for each base test and using each non-base choice in each other characteristic [1]. As shown in Fig.1, a coverage criterion subsumes the ones below.

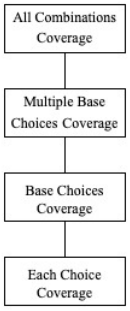

Fig. 1. Subsumption relations among choice-based input space partitioning criteria [1]

The IEEE defines unit testing as "the testing of individual software or hardware units or groups of related units" [3]. Since unit testing can result in significant gains in software quality [4], many unit testing frameworks have been developed and they are integrated with IDEs. Although structural (white-box) test data generation methods for unit tests [5] have been around so long, unit testing frameworks seem to be reluctant to add them to their functionality.

Path testing is a white-box technique. Branch coverage, as a coverage criterion for path testing, is achieved when every branch from a node is executed at least once by a test suite [6]. Branch coverage is more effective than statement coverage but less effective than condition coverage, where condition coverage requires 2^n combinations of a predicate with n conditions to be exercised [6].

III. Related Work

Test case values should be determined in order to execute test cases. Domain testing enables test practitioners to divide the domain into partitions and select values from those partitions [6]. One approach in partitioning is to use equivalence classes for program inputs [7]. In contrast to this black-box approach, White and Cohen [8] proposed a white-box testing criteria where test values are determined using program execution paths. This control-flow testing criteria is improved with data-flow testing criteria [9], where dataflow relationships in a program guide test data selection.

Feedback based testing approaches in the literature are adaptive random testing (ADT) [10], adaptive combinatorial testing [11], adaptive concolic testing [12], and search-based testing [13]. All these techniques can be applied to unit test generation. ADT was introduced to improve the fault-detection effectiveness of random testing by distribute test cases more evenly within the input space [10]. Adaptive combinatorial testing is a feedback-driven combinatorial testing approach aimed at working around masking effects that are observed in combinatorial testing [11]. The main idea of adaptive concolic testing is to improve the coverage obtained by feedback-directed random test generation methods, by utilizing concolic execution on the generated test drivers and utilize non-linear solvers to generate new test inputs for programs with numeric computations [12]. In search-based testing, metaheuristic search techniques have been applied to automate test data generation for structural and functional testing [13]. The proposed approach differs from structural, random, combinatorial, and search-based unit test data generation approaches since it provides feedback to input domain testing with coverage information.

Research on automatic test generation for unit testing has been heavily on Java and JUnit. One of the first in this field, "The JML and JUnit Way" uses a formal specification language's runtime assertion checker to decide whether methods are working correctly, thus automating the writing of unit test oracles [14]. "Jartege" is another unit test generation tool that generates random tests for Java programs specified in JML using this specification as a test oracle in the JML-JUnit way [15]. "Eclat" utilizes a technique for automatically producing an oracle, i.e. a set of assertions, for a test input from the operational model that is inferred from programs correct executions [16]. Stock et al. proposed a technique for automatic generation of a test suite from a given UML class diagram of the system [17]. Sharma investigated automatic generation of test suites from decision tables [18]. All this research is on the generation of test oracles through models, which require extensive modeling and formalism knowledge that does not exist on the developer profile this paper aims for.

Different than most of the tools developed for the related research, the supporting tool works with C# and MSTest unit testing framework.

IV. Proposed Approach

The proposed approach defines a workflow that helps unit test developers to achieve aimed branch coverage through MBCC-based testing. High level algorithm of the proposed workflow is presented using both Alg.1 and Alg.2. Alg.1 shows steps in MBCC-based testing. To include coverage guidance, the proposed workflow is extended as in Alg.2. The proposed workflow is supported with a tool, which is explained in the next section.

The first step of Alg.1 is determination of the test input parameters for the MUT. MUT parameters and return values of the called methods within MUT are considered as test input parameters. The second step is determination of equivalence classes and their representative values for test input parameters using the domain knowledge. In the next step, at least one base for MBCC is decided. In the following steps, test inputs and test names for unit test cases are automatically

generated and then duplicate test cases are automatically removed. At this point, the test suite is ready for test outputs. Developer is expected to fill in test outputs for each test case. Once the test suite is ready, unit test driver method is automatically generated with test data but without specific assert statements. After the developer has completed unit test driver method, she runs it and checks for failed tests. Since workflow for failed tests is not in the scope of this paper, that step is omited here.

Alg. 1. MBCC-Based Unit Testing

```
1. read test input parameters
2. for each parameter do
3.     read partitions and representative values for each parameter
4. end for
5. read bases
6. generate_test_inputs_for_test_cases()
7. generate_test_names_for_test_cases()
8. remove_duplicate_test_cases()
9. read test outputs for test cases
10. generate_unit_test_driver_method()
11. complete unit test method
12. run unit test driver method
```

The proposed workflow extends workflow outlined in Alg.1 as seen in Alg.2. After completion of the steps in Alg.1, branch coverage is checked. If it passes the desired level, Alg.2 ends there. However, if it is below the desired level, then the feedback loop is entered, and we let the coverage information guide the improvement process. In the proposed approach, coverage information is used only for guidance, especially for two purposes:

1. Use coverage percentage to check if desired level is reached because branch coverage is one commonly used metric in industry.
2. Use uncovered blocks to obtain clues on the missed equivalence classes and bases.

Alg. 2. Coverage Guided MBCC-Based Unit Testing

```
1. run Alg. 1
2. read desired branch coverage
3. read branch coverage
4. while desired branch coverage < branch coverage do
5.     read additional partition(s) with value(s) and/or base(s)
6.     generate_test_inputs_for_additional_test_cases()
7.     generate_test_names_for_additional_test_cases()
8.     remove_duplicate_test_cases()
9.     read test outputs for additional test cases
10.    generate_unit_test_driver_method()
11.    complete unit test method
12.    run unit test driver method
13.    read branch coverage
14. end while
```

If the coverage percentage is below the desired level, then we consider each uncovered block providing a hint to improve equivalence classes and bases. Once the developer decides on the additional partition(s) with value(s) and/or base(s), additional unit test cases are generated without duplicates and their expected outputs are entered. Then the data rows for unit test driver code is automatically generated from scratch. The developer replaces existing data rows of unit test driver method with the new data rows and updates assert statement(s) if necessary. After that, the developer runs it and checks for coverage percentage. The loop continues until the desired coverage level is reached.

V. Proposed Tool

The proposed workflow is supported with the tool designed and developed for coverage guided MBCC-based unit testing. The tool is named as Multi Base Choices Coverage Tool and is shown in Fig.2.

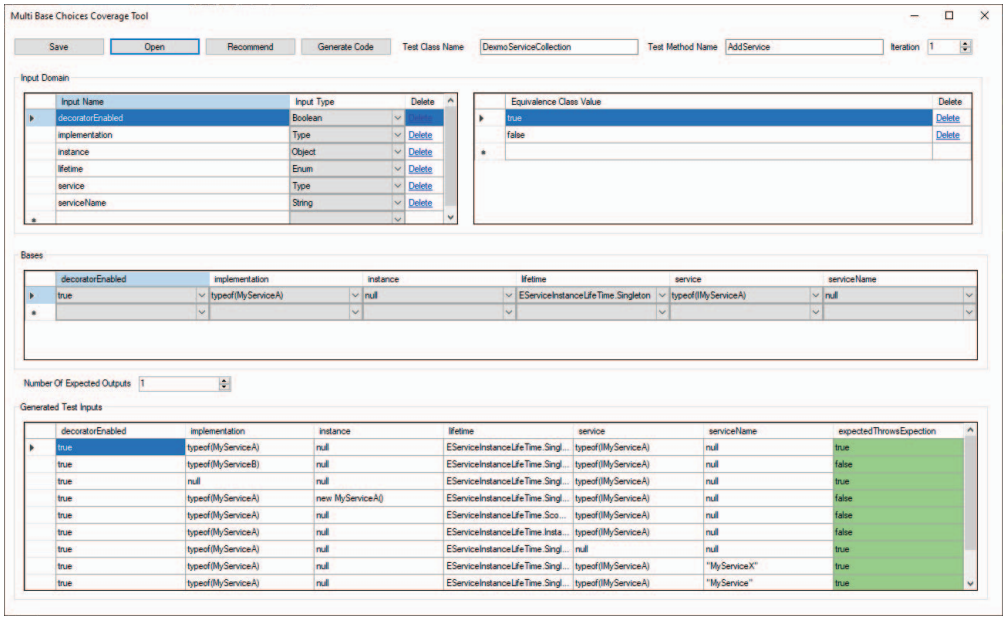

Fig. 2. Main screen of Multi Base Choices Coverage Tool

The workflow explained above also outlines how to use the developed Multi Base Choices Coverage Tool. To demonstrate the tool, Add Service method of Dexmo Service Collection class written in C# is used as a running example. AddService method is responsible for adding new items (service definitions) into the DexmoServiceCollection instance. It has 76 lines of code with the following method signature:

public void addService(DexmoServiceDescription description);

The main screen of Multi Base Choices Coverage Tool is composed of four panels. The top panel contains buttons along with MUT and its class as well as iteration number. The *Input Domain* panel enables the developer to enter test input parameters with its type and equivalence classes with representative values. The chosen base(s) are entered using the *Bases* panel. The bottom panel shows on-the-fly generated test inputs for test cases.

In the *Bases* panel, each row represents a base in MBCC. The selected equivalence classes and their representative values for test input parameters form a base. Test cases are automatically generated by keeping base value for the first parameter constant and alternating values for other parameters. Then same operation is performed by keeping base value for the second parameter constant and alternating other values. This operation is repeated for all test input parameters in the base. Automatically generated test cases may contain same test case more than once. When it is recognized the second test case is eliminated and not shown in the bottom panel.

Although not shown in the bottom *Generated Test Inputs* panel, test case names are also automatically generated in the following format:

Test Case Number [,InputName:Value] [,ExpectedOutput:Value,]**

Our experience has shown that developers have hard time to remember and understand a test case if it fails. Therefore, we choose such a test case naming format. Since the developers are familiar with the domain and they follow the principle of keeping representational gap (between the concepts used in real life and identifiers/names used for variables, types/classes and methods in code) low, they are able to see immediately what is wrong with the failed test case. Moreover, this principle helps us in analyzing uncovered blocks and therefore improving equivalence classes and base choices.

Once the developer is ready with base choices and that means test inputs and test names are automatically ready, she can enter expected output(s) for each test case using the main screen of the Multi Base Choices Coverage Tool. *Generate Code* button takes the developer into a dialogue screens where she finds generated unit test driver method template with data rows representing unit test cases as given in Fig.3. As industry best practice, unit test data should be presented in rows above the test driver method. However, its representation changes with respect to unit testing framework utilized. In our case, the representation in Fig.3 reflects MSTest unit testing framework. The proposed workflow expects the developer to copy and paste the generated unit test driver method to the IDE, which is in our case Microsoft Visual Studio™, and complete it with necessary assert statement(s), such as shown in Fig.4.

After executing unit test driver method, the developer checks coverage percentage as shown in Fig.5 and compares it with the desired percentage. If it is lower than the desired, then improvement loop in Alg.2 should be run. Before explaining that, it should be noted that to archive unit test designs, the developer can use *Save* button. She can save a test design with or without coverage information, which is noted in the upcoming dialogue screen.

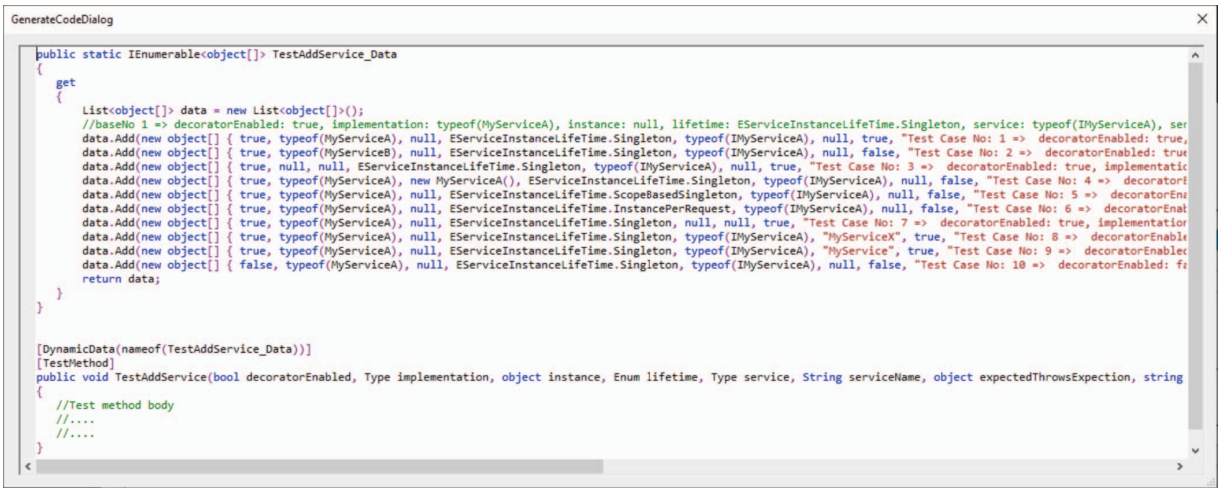

Fig. 3. Automatically generated unit test driver method template in C# for MSTest unit testing framework

```
[DynamicData(nameof(TestAddService_Data))]
[TestMethod]
0 references | 0 changes | 0 authors, 0 changes
public void TestAddService(bool decoratorEnabled, Type implementation, object instance, EServiceInstanceLifeTime lifetime,
                           Type service, String serviceName, bool expectedThrowsException, string testCaseName)
{
    DexmoServiceCollection collection = new DexmoServiceCollection();
    collection.AddService(typeof(IMyService1), new MyService1(), "MyServiceX");
    var description = new DexmoServiceDescription
    {
        DecoratorEnabled = decoratorEnabled,
        Implementation = implementation,
        Instance = instance,
        Lifetime = lifetime,
        Service = service,
        ServiceInterceptors = null,
        ServiceName = serviceName
    };
    try { collection.addService(description); }
    catch (Exception)
    {
        if (!expectedThrowsException) { throw; }
        else { return; }
    }
    if (string.IsNullOrEmpty(description.ServiceName))
    {
        Assert.IsTrue(collection.IsServiceRegistered(description.Service));
    }
    else
    {
        Assert.IsTrue(collection.IsServiceRegistered(description.Service, description.ServiceName));
    }
}
```

Fig. 4. Developer completed unit test driver method

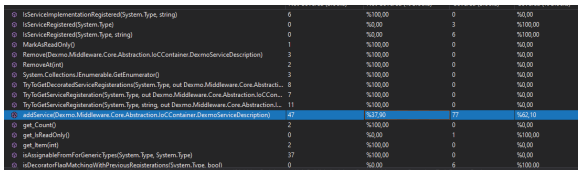

Fig. 5. Coverage information in Microsoft Visual Studio™

To improve MBCC-based unit test design, we propose an automated recommendation mechanism. This mechanism is implemented in the Multi Base Choices Coverage Tool. It gives recommendation on which test input parameter(s) may be evaluated in that code block so that the developer can work on the missing equivalence class(es) and base(s) by analyzing uncovered blocks. The developer should choose one of the uncovered blocks indicated by the IDE, in our case Microsoft Visual Studio ™ and copy-paste it to dialogue screen appears after clicking *Recommend* button. When OK button is pressed as shown in Fig.6, if there is a recommendation it is shown in the following dialogue screen such as the one given Fig.7.

Fig. 6. Request recommendation for an uncovered block

The Generalized Levenshtein Distance Algorithm [18] is utilized to match identifiers in the uncovered block with the test input parameters. The Levenshtein distance between two words is the minimum number of single-character edits (i.e. insertions, deletions, or substitutions) required to change one word into the other [19]. The developer can work on the recommended test input parameter either by adding new equivalence classes to it or by adding it as a base.

Once the developer enters additional partition(s) with value(s) and/or base(s), Multi Base Choices Coverage Tool automatically generates additional unit test cases without duplicates and asks the developer to enter expected outputs for these additional test cases. Then the data rows for unit test driver code is automatically generated from scratch. The developer replaces existing data rows of unit test driver method with the new data rows and updates assert statement(s) if necessary. After that, the developer runs it and checks for coverage percentage. The loop continues until the desired coverage level is reached.

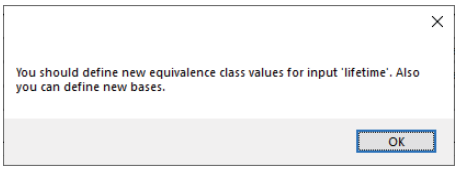

Fig. 7. Base recommendation for an uncovered block

At each iteration, the developer improves coverage using MBCC-based test case design. For the running example, it took us four iterations to reach company set level of coverage, which is 95%. Screenshots of the Multi Base Choice Coverage Tool for the remaining iterations are given in the Appendix.

The improvements obtained throughout these four iterations are presented in Table I and explained below:

- After iteration 1, 62,10% coverage is obtained.
- After iteration 2, where we added "EServiceInstance LifeTime.PerMatchingScopeBasedSingleton" item into "lifetime" input's equivalence class, coverage is improved to 68,55%.
- At iteration 3, we added "typeof(IMyService1)" item into "service" input's equivalence class resulting in 74,19% coverage.
- With the fourth iteration, where the base "decorator: false, implementation: typeof(MyService1), instance: null, lifetime: EServiceInstanceLifeTime.Singleton, service: typeof(IMyService1), serviceName:null" defined, coverage reached to 95,97% passing the desired level.

TABLE I. IMPROVEMENT THROUGH COVERAGE GUIDED MBCC-BASED TESTING FOR DEXMOSERVICECOLLECTION.ADDSERVICE METHOD

Iteration #	# of Bases	# of Test Cases	Test Execution Time	Total Blocks	Covered Blocks	Block Coverage
1	1	10	18 ms	124	77	62,10%
2	1	11	18 ms	124	85	68,55%
3	1	13	19 ms	124	92	74,19%
4	2	26	20 ms	124	119	95,97%

For each iteration, test execution times are 18, 18, 19, and 20 milliseconds in average, respectively. They are obtained after 10 trials on a Windows 10 Pro v. 1909 - 64bit machine with Intel Core i7-9750H @2.6GHz CPU and 16 GB RAM.

For the methods with one base, developers easily reach the desired level of coverage using our tool. With more bases, the test design process gets harder. In the following section, it is discussed why the proposed approach and tool is necessary for efficient unit testing.

VI. DISCUSSION

The proposed workflow supported with the Multi Base Choices Coverage Tool is for software developers who have limited knowledge and experience on formal approaches, modeling, and test case design but know the software domain and unit testing.

Our experience shows that the proposed approach and tool is useful for the methods under test that has more than one base. For a MUT having a base, developers can easily produce inline test data. However, we still insist to use our tool because of its archival feature. If that method changes in the future, even the same developer has hard time remembering equivalence classes and selected values for those classes. If a MUT has more than one base, then developer is confused in selection order of bases. With our tool, trial and error is fast and cheap.

While using the proposed approach and tool, it is better to take just one uncovered block and work on it in one iteration since it is hard to know which uncovered blocks will be covered with added equivalence class(es) and base(s). We haven't observed any infeasible test cases during the use of approach and tool. However, if it happens, the developer leaves the expected output for that test case empty in the generated test inputs panel and the tool recognizes it as an infeasible test case and do not include in the automatically generated unit test driver method template code.

Developers feel the control of test design with our tool for several reasons. First of all, adding partitions and values are very easy as well as selecting them. Second, duplicate test cases are automatically eliminated. Third, while entering test case outputs, developers have another chance to evaluate their selection of partitions and values and if necessary, they can easily change them. Each change is automatically reflected on the test cases in the tool, there is no need to press any buttons. Fourth, since test cases are systematically generated human error is eliminated in this step. Fifth, test driver with inline test data is automatically generated within a second. Sixth, giving a hint about an uncovered block for missing equivalence class is a feature that developers enjoy. Finally, the archival feature helps developers to store the knowledge of test design for future use.

Assuming the developer has the domain knowledge of the MUT, time required for determining test input parameters, equivalence classes and their respective values as well as bases takes 3 to 5 minutes. Entering test case expected outputs, which is the second manual task that the developer should complete, requires more time and that time changes with respect to the number of test cases. Our developers needed 15 to 20 minutes to fill in the expected outputs for a MUT like in the running example. Once the test driver is automatically generated, filling in the assert statements takes 8 to 10 minutes for a MUT like in the running example. If new bases were required, then comparatively low additional time would be necessary.

For the running example, we asked one developer to prepare unit tests with our tool and another one without our tool. While the one using our tool finished within half an hour, the other one needed half a day. When we asked the one, who didn't use our tool, what troubled him mostly, the answer was that he got lost among test cases. For high number of bases, our approach with the tool is a necessity. One drawback of our approach is possible redundancy in test cases. However, with the automation brought by our tool, this drawback is minimized.

VII. CONCLUSION

A coverage guided MBCC-based unit testing approach is presented with a feedback loop-controlled testing workflow. The proposed workflow is supported with a tool. Multiple base choices coverage criterion is chosen for systematic unit test generation in the proposed approach and branch coverage information is utilized as feedback to improve selection of bases, which results in improved branch coverage.

The Multi Base Choices Coverage Tool automatically generates unit test case names and test case inputs once the developer enters test input parameters, equivalence classes with representative values, and bases. After the developer fills in expected outputs for unit test cases, the tool automatically generates unit test driver method template, where only assert statements are missing. After completion of assert statements and execution of unit test method, branch coverage percentage is checked using Microsoft Visual Studio™ IDE. Then following the proposed feedback loop, branch coverage is advanced by improving MBCC.

As future work, we plan to convert the tool into an add-on for Microsoft Visual Studio™. The developed tool generates unit test driver method template with respect to MSTest unit testing framework. However, the tool can be enhanced to support other unit testing frameworks. Moreover, we plan to have a Java and JUnit version of it. In this version of the tool, test case outputs are filled by the developer. This step can be improved by automated oracles, which is left as future work as well. Moreover, we plan to improve the recommendation mechanism using Gensim (https://github.com/RaRe-Technologies/gensim), which is an advanced NLP library for similarity retrieval.

REFERENCES

[1] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[2] A. P. Mathur, *Foundations of software testing, 2/e*. Pearson Education India, 2013.

[3] N. Juristo, A. M. Moreno, and W. Strigel, "Guest editors' introduction: Software testing practices in industry," *IEEE software*, vol. 23, no. 4, pp. 19–21, 2006.

[4] E. Dustin, *Effective Software Testing: 50 Ways to Improve Your Software Testing*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[5] B. Korel, "Automated test data generation for programs with procedures," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 3, pp. 209–215, 1996.

[6] R. Binder, *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[7] G. J. Myers, "The art of software testing. 1979," *A Wiley-Interscience Publication*.

[8] L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE transactions on software engineering*, no. 3, pp. 247–257, 1980.

[9] M. J. Harrold and G. Rothermel, "Performing data flow testing on classes," *ACM SIGSOFT Software Engineering Notes*, vol. 19, no. 5, pp. 154–163, 1994.

[10] T. Y. Chen, H. Leung, and I. Mak, "Adaptive random testing," presented at the Annual Asian Computing Science Conference, 2004, pp. 320–329.

[11] E. Dumlu, C. Yilmaz, M. B. Cohen, and A. Porter, "Feedback driven adaptive combinatorial testing," presented at the Proceedings of the 2011 International Symposium on Software Testing and Analysis, 2011, pp. 243–253.

[12] P. Garg, F. Ivančić, G. Balakrishnan, N. Maeda, and A. Gupta, "Feedback-directed unit test generation for C/C++ using concolic execution," presented at the 2013 35th International Conference on Software Engineering (ICSE), 2013, pp. 132–141.

[13] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[14] Y. Cheon and G. T. Leavens, "A simple and practical approach to unit testing: The JML and JUnit way," presented at the European Conference on Object-Oriented Programming, 2002, pp. 231–255.

[15] C. Oriat, "Jartege: a tool for random generation of unit tests for java classes," in *Quality of Software Architectures and Software Quality*, Springer, 2005, pp. 242–256.

[16] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," presented at the European Conference on Object-Oriented Programming, 2005, pp. 504–527.

[17] M. Stock, A. Brucker, and J. Doser, "Automatic Generation of JUnit Test-Harnesses," *Semester Thesis, Swiss Federal Institute of Technology, Zurich, Switzerland*, 2007.

[18] M. Sharma, "Automatic generation of test suites from decision table-theory and implementation," presented at the 2010 Fifth International Conference on Software Engineering Advances, 2010, pp. 459–464.

[19] L. Yujian and L. Bo, "A normalized Levenshtein distance metric," *IEEE transactions on pattern analysis and machine intelligence*, vol. 29, no. 6, pp. 1091–1095, 2007.

[20] N. Babr, "The Levenshtein Distance Algorithm," Oct. 02, 2018. https://dzone.com/articles/the-levenshtein-algorithm-1 (accessed Apr. 26, 2020).

Appendix

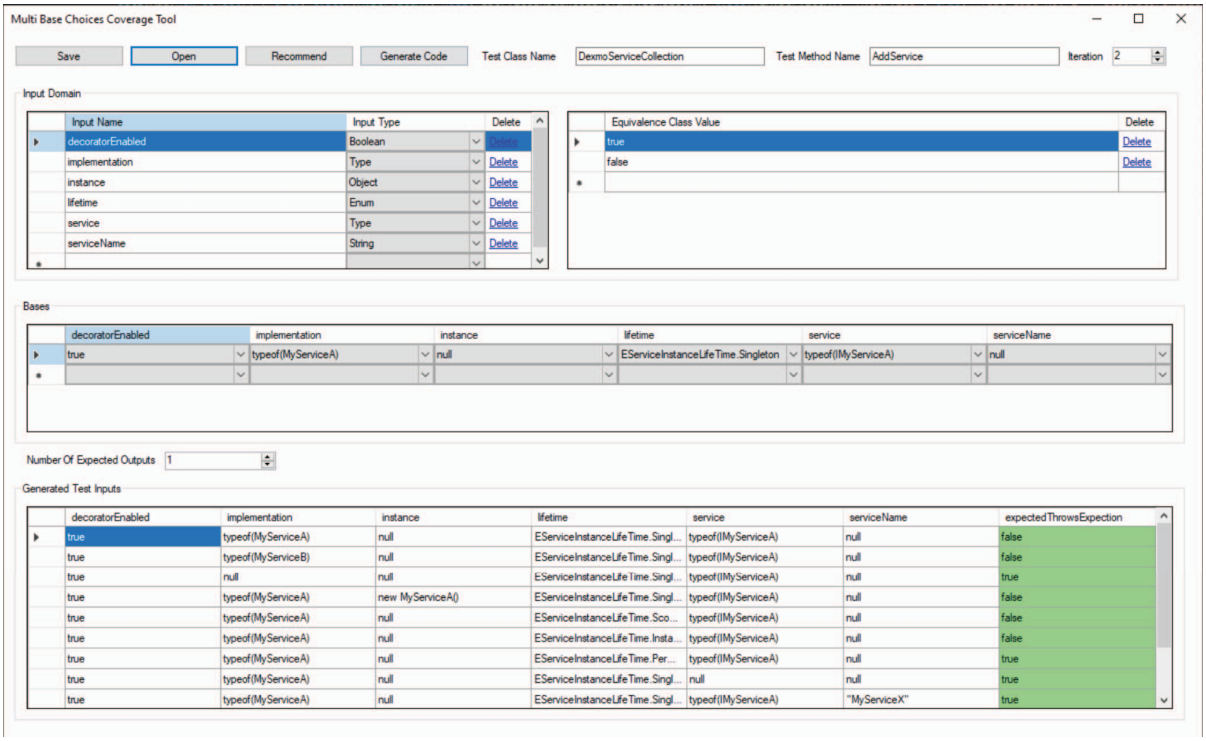

Fig. 8. Main screen of Multi Base Choices Coverage Tool for iteration 2

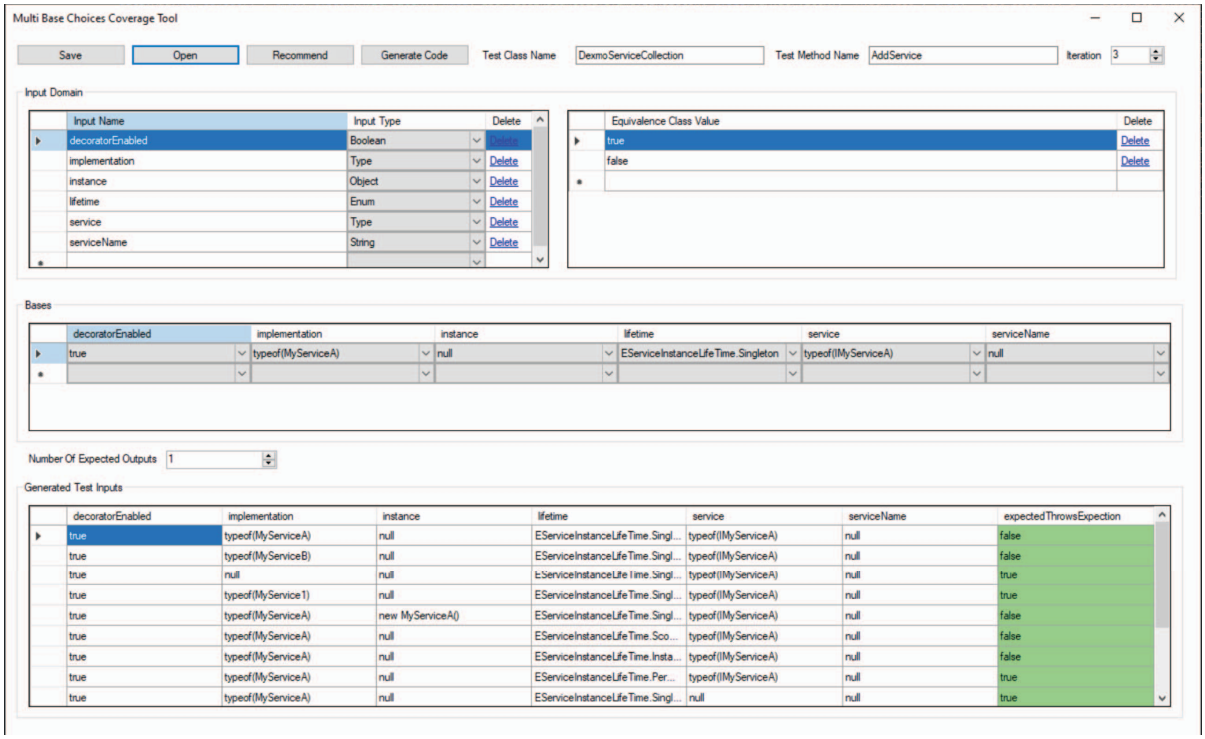

Fig. 9. Main screen of Multi Base Choices Coverage Tool for iteration 3

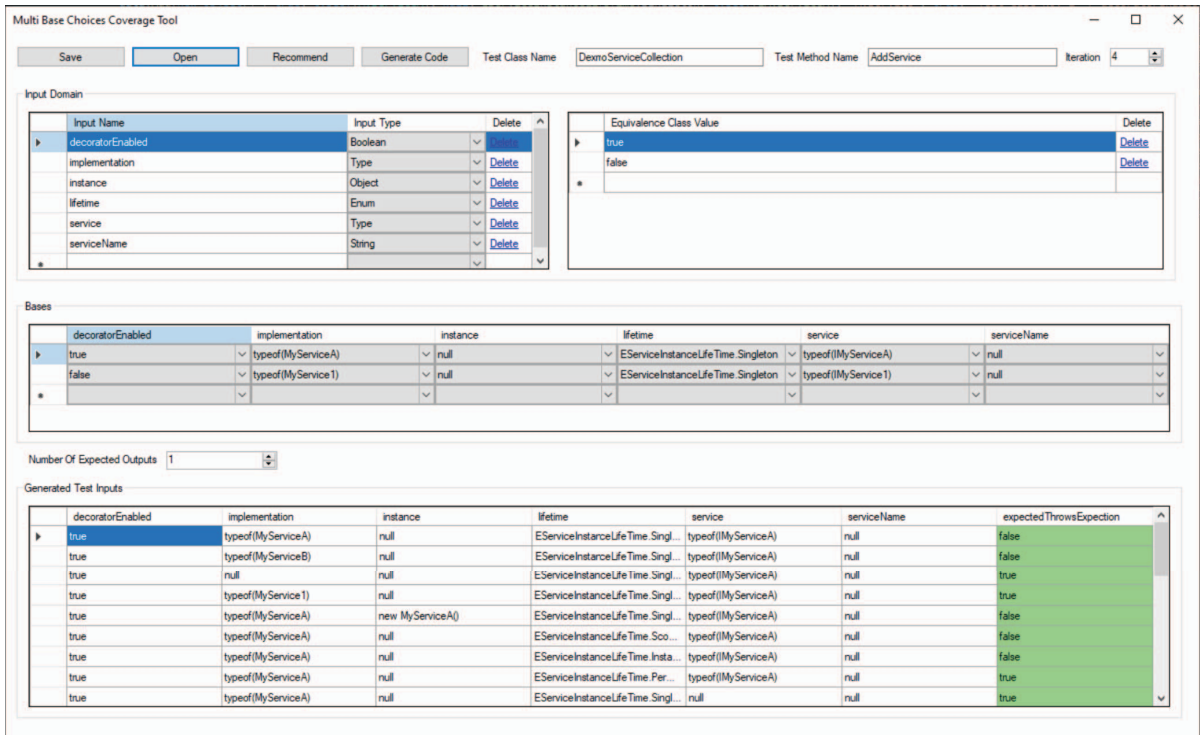

Fig. 10. Main screen of Multi Base Choices Coverage Tool for iteration 4