Code Coverage Aware Test Generation Using Constraint Solver

Krystof Sykora¹, Bestoun S. Ahmed², and Miroslav Bures¹

 ¹ Department of Computer Science, Faculty of Electrical Eng, Czech Technical University, Prague, Czech Republic {sykorkry,buresm3}@fel.cvut.cz
² Department of Mathematics and Computer Science, Karlstad University, Sweden bestoun@kau.se

Abstract. Code coverage has been used in the software testing context mostly as a metric to assess a generated test suite's quality. Recently, code coverage analysis is used as a white-box testing technique for test optimization. Most of the research activities focus on using code coverage for test prioritization and selection within automated testing strategies. Less effort has been paid in the literature to use code coverage for test generation. This paper introduces a new Code Coverage-based Test Case Generation (CCTG) concept that changes the current practices by utilizing the code coverage analysis in the test generation process. CCTG uses the code coverage data to calculate the input parameters' impact for a constraint solver to automate the generation of effective test suites. We applied this approach to a few real-world case studies. The results showed that the new test generation approach could generate effective test cases and detect new faults.

Keywords: Software Testing · Code Coverage · Automated Test Generation · Test Case Augmentation · Constrained Interaction Testing.

1 Introduction

In software engineering, regression testing has become a common practice to be used during test development. In this practice, testers begin by rerunning existing test suites to validate new software-under-test (SUT) functionality. However, this approach faces many problems as existing tests decrease their ability to detect SUT faults. This paper shows how Test suite augmentation techniques can be used to solve this problem³.

To improve the test generation, tools like code coverage analysis and constraint solving are commonly utilized. We use these techniques in the Code Coverage-based Test Case Generation (CCTG) method. This approach is akin to Test suite augmentation techniques [10], which is commonly used in regression testing. Augmentation is employed to adjust existing test cases by analyzing

³ This paper is accepted for publication in the 2nd International Workshop on Automated and verifiable Software sYstem DEvelopment

changes in the SUT. Practitioners of Test suite augmentation techniques believe the use of preexisting test cases can improve that test suites [9]. To promote these test cases, the SUT is analyzed using code coverage and other criteria to prioritize how changes to the test suites should be conducted. In our method, we focus on the second part of the augmentation approach – code coverage. By analyzing the code coverage of preexisting generated test cases, we can determine test parameters' impact. This information is then used for the test generation.

CCTG can be used for the test suite augmentation method by utilizing the code coverage data and test models to refine the test generation process. To generate the test cases, CCTG first creates the test data sets used for the input interaction. A test model consists of the SUT parameters classes (i.e., possible parameter values). The results are sets of specific SUT parameter values, which will be referred to as test cases. The parameter classes are then used to generate random test cases to be executed for code coverage monitoring. This process generates coverage data related to each generated test case. If a specific test parameter change affects the coverage consistently or significantly, the parameter's weight also increases. The weight indicates the extent to which each parameter is used and permuted when generating a new set of test cases. The process as a whole contributes towards the test suite augmentation. Here, the code coverage data used for the augmentation may also minimize the number of test cases and generate more effective test cases in terms of fault-finding.

As in most modern software systems, the test suite augmentation also suffers from input interaction constraint problems [1]. Here, we deployed a constraint solver within our approach during the test generation process to resolve the input values' constraints. The solver makes sure no test cases are generated with meaningless interactions with the SUT. The resulting test cases should resemble the general workflow of test cases [4] as the code coverage analysis motivates variety in decision paths, and the constraint solver ensures that the combinations remain reasonable.

2 Background

Classically, code coverage has been used as an analytical approach with the test suite execution. The approach is also used with the test suite generation strategies to maximizing the code-base covered by generating more test cases [8]. We have recently used more advanced techniques function as gray-box methods for test case analysis and test generation [2]. We considered the code's internal structure while augmenting a generated test suite for Combinatorial Interaction Testing (CIT). This approach improved the test generation process by studying the program's code to determine individual parameters' impacts. We used this impact factor to select the input parameters and values for the test cases. In this regard, the CCTG strategy proposed in this paper is very similar. However, examining the internal code structure can prove costly, in terms of a manual analysis of the input parameters and values. So, the CCTG simplifies this process by automatically evaluating the parameter impact, requiring only a test model of

parameters and their possible values. It can also explore various hidden criteria for test generation (such as SUT configuration and state).

A common problem for test generation is that the test models don't consider specific constraints that would make the test cases useless or nonsensical. To avoid this, test generation methods [7,11] rely on constraint solvers to eliminate such undesirable combinations. Such constraint or rather Boolean satisfiability problem-solvers [3] (SAT solvers) are also employed in the CCTG method. The solver's additional benefit is that the CCTG user may incorporate the SAT constraints into the test model to prevent the generation of unwanted test cases and correct the focus of the test cases. If new functionality is added to the SUT, it can be specified that some parameters (representing the choice to use the new functionality) must be used. This allows the tester to influence the test generation by only adjusting the test model without changing the code coverage analysis results.

While the CCTG finds relations to the previously mentioned techniques, it is primarily a test suite augmentation method [10]. Our approach is based on an innovative idea of augmenting test cases using the coverage criteria. The CCTG method, however, uses randomly generated test cases to establish the coverage data. This data is then used to generate what is essentially the first set of actual test cases.

3 The proposed method

In this section, we examine each step of the CCTG method. The following subsections illustrate these steps in detail.

3.1 Determining parameter weight

The initial step in the CCTG methods is the code coverage analysis. This process is used to calculate the impact (weight) of SUT parameters that will be used for test generation.

SUT Test model For each SUT, a model (shown in figure ??(a)) for interaction consisting of the input parameters and constraints among them. The parameters are represented as the P[n] array, where each index P[1], P[2],... P[n] represents one of the test model parameters. Each parameter has a value. The value V is the possibility of a parameter. This can be represented in two ways, either the V is an array of the possible values for parameters (booleans or enums) from the SUTs perspective, or the value can be represented as a number with a range. In the case of range representation, the parameter depth [d] is added to represent the number of values chosen from the range for further test generation. The other parameter property in the model is its weight, which is graded as a float ranging from 0 to 1.

The second part of the model is a set of SAT constraints to ensure that the conflicting decision is not selected. This model is used to generate the initial test

4



Fig. 1: SUT modeling and parameter analysis

cases for code coverage analysis and generate the resulting test set for the actual testing. For regression testing, the first set of random test cases can be replaced with a set of regression tests from earlier stages of the SUT development.

Test case analysis and generation In contrast to other strategies, maximizing of code coverage is not a direct goal for CCTG. The strategy relies on using the coverage data in a more specific manner more targeted on condition or coverage principles. We follow the standard definition of code coverage for lines of code per individual test case. The $gcov^4$ tool and stored for later analysis for each test case. The test suite used for coverage determination is designed in a very specific way. As the initial step of generation, we select a test depth level. This level represents how many values will be selected for each parameter. The same number of values as overall test depth is selected for each parameter P and randomly ordered in an array RV. The first test case TC uses each parameter's first value in all randomly ordered lists. For every test case after that, one parameter changes the value to the next in the randomised list (P[1],RV[1])to P[1].RV[2]) as shown in ??(b). We proceed this way until all the arrays are exhausted. The rationale behind this approach is its effects on code coverage change that determines the parameter impact. We always select two test cases where all but one parameter have the same values to measure this. This will be explained in the Parameter weight calculation section and the figure 2

Parameter weight calculation The initial step in this phase is to determine each parameter's effect on code coverage (i.e., the parameter weight). Using the initial test cases, the code coverage is gathered automatically. To determine the information about a specific parameter, we must take a look at a set of test cases, where the examined parameter changes, while the remaining parameters stay the same, as shown in figure 2.

Initially, a parameter is selected by the algorithm for analysis. Then, similar pair test cases are selected, except for the value of the parameter under investi-

⁴ https determines the code coverage://linux.die.net/man/1/gcov

Code Coverage A	Aware Test	Generation	Using	Constraint Solver
-----------------	------------	------------	-------	-------------------

TC[1] TC[2] TC[3]	$ = \begin{matrix} \{[P(1], RV[1], P[2], RV[1], \dots, P[k], RV[1], \dots, P[n], RV[1]], CC\} \\ = \\ \{[P(1], RV[2], P[2], RV[1], \dots, P[k], RV[1], \dots, P[n], RV[1]], CC\} \\ = \\ \{[P(1], RV[2], P[2], RV[2], \dots, P[k], RV[1], \dots, P[n], RV[1]], CC\} \end{matrix}$
	•
	•
TC[k] TC[k+1]	$ = \{ [P[1], RV[2], P[2], RV[2],, P[k], RV[1],, P[n], RV[1]], CC \} \\ = \{ [P[1], RV[2], P[2], RV[2],, P[k], RV[2],, P[n], RV[1]], CC \} $
	•
TC[n] TC[n+1] TC[n+2]	$ = \{ [P[1], RV[2], P[2], RV[2],, P[k], RV[2],, P[n], RV[1] \}, CC \} \\ = \{ [P[1], RV[2], P[2], RV[2],, P[k], RV[2],, P[n], RV[2] \}, CC \} \\ = \{ [P[1], RV[3], P[2], RV[2],, P[k], RV[2],, P[n], RV[2] \}, CC \} $

Fig. 2: Test case selection

gation. The difference in the code-coverage results of the test cases is recorded. These steps are repeated for all remaining pairs of test cases matching the criteria of all parameters being the same, except for the changing parameter under analysis. From the derived differences in coverage, an average is calculated. This average represents an absolute value of the parameters weight. These steps are illustrated in detail in the Algorithm 1.

Algorithm 1: Steps in the code coverage analysis to determine parameter's impact

1 CodeCoverageAnalysis $(T_C L, P)$				
	Input : Test case list $T_C L$, Parameter-under-test P			
	Output: Measure of the impact of parameter P on code coverage $avrg(DL)$			
2	TL = TestCaseList;			
3	DL = list of float values;			
4	while $T_C L$ not empty do			
5	$TL \operatorname{add}(T_C L \operatorname{pop} \operatorname{first item});$			
6	for each T_C in $T_C L$ do			
7	if each $i/PI : PV[i]$ in $TC == PV[i]$ in $TL[0]$ then			
8	TL add(pop TC from $T_C L$);			
9	DL add $(max(TL), min(TL))$:			
10				
10				
11	return $avrg(DL);$			

Maximum code coverage for the generated tests is not the aim here. Instead, by determining the impact on individual parameters' coverage, parameters are selected for permutation in test cases if they influence code coverage more.

3.2 Test case generation

Test cases are generated using the data of parameter weight. All unary parameters have only two possible permutations (0,1) as they can only be or not be included upon program execution. The number of binary parameter permutation

5

is equal to 1 + the number of pre-selected possible values. The unary parameter is a set with two elements (one being the parameters exclusion) and the binary a set with 1 + (number of values) elements. All test cases created are solved for constraints.

The code-coverage based method is relying on the selected parameters to permute based on their weight. All weights are places on an axis, spanning the range from 0 to 100. Thanks to normalizing the values, their sum is 100 exactly and, as such, fills up the entire axis. For example, take 3 parameters with respective weights 15, 60, and 25. The parameters would assume the following ranges on the axis. P1 (0,15), P2 (15,75) and P3(75,100).

The first constructed test case has all parameters with their default values. A random real number is generated in the range of (0,100) to construct the next test case. A parameter whose range corresponds (on the axis) to the number generated is selected for the permutation. The selected parameter's value is permuted to the next value in the sum that represents it. The test case is then saved if it is not identical to a previously created one. The test case generation algorithm is included in Algorithm 2.

Algorithm 2: Code-coverage-based test-case generation				
1 CVBasedTCGeneration (PL, n)				
Input : ParameterList PL , Number of test cases to be generated n				
Output: TestCaseList T_CL				
2 $T_C L = \text{list of TestCases };$				
3 CT_C = TestCase where each P_V in $P_V L$ is 0;				
4 while $n > 0$ do				
5 $r = random float value;$				
6 t = 0;				
7 foreach P in PL do				
8 $t = t + PW$ in P;				
9 if $r \leq t$ then				
10 permute (P) ;				
11 add(PL);				
12 $n-;$				
13 $\begin{bmatrix} -\pi \\ return \\ T_C L; \end{bmatrix}$				

We have used the Z3 solver with this test generation to resolve the constraints. We have also developed an interface for the tool so that the relevant constraints for test cases exclusion can be imported from the test model.

4 Experimental evaluation

To evaluate the CCTG method, we have conducted three case studies. The experiments were designed according to the Mutation Testing [6] approach. For each case study, several mutants were created using a fault seeding framework. To test the CCTG method's effectiveness, the generated test cases were used for both the original version of the SUT and the mutated (faulty) version. The

outputs of the two SUTs were then compared. When the outputs differ, the fault is considered to be detected. The mull [5] LLVM-based tool was used.

For the case studies, the Unix utilities Flex, grep, gzip were selected. The Software-artifact Infrastructure Repository⁵ obtained from the Gnu site. These utilities were chosen, as the test cases generated represent their command-line arguments. Therefore, each test case is essentially a parametrized call of these utilities that produces a standard output test.

The test case structure reflects the archetype of the SUTs from the case study, standard C utilities, requiring only a set of arguments. For the case study, CCTG test cases were therefore represented by a set of command-line arguments only. The type of arguments or parameters used can be divided into a few basic categories. There is a unary argument – which always represents a Boolean value. As the parameters are for typical command-line programs, these usually specify some functionality (e.g., –printToCommandLine) that is enabled or a specific instruction (e.g., –help). The second type is binary arguments, consisting of parameter and value (e.g., –input /inputs/file1).

For each SUT, a set of all possible parameters is gathered. This is a simple list of all unary parameters accompanied by possible values for the binary ones. In many cases, the parameter values are subjective to the testing environments. Input files are part of prepared testing inputs for a specific program. Values with range such as integers are also limited to discrete and finite selection. The range's selecting values are done either based on other testing information, chosen randomly, or at regular intervals. The selection of specific values from a range does not directly affect the experiments as the selections are final for all test generation methods. Seeded versions of SUT are executed using test cases generated by various methods to measure the test effectiveness. The methods used are compared to the code-coverage method.

As a reference, we use two methods of test generation: Random generation and unweighted method. The random generation method is based on random parameter permutation. The initial setup is very similar to the CCTG method, as prepared sets represent the parameters. Unlike the CCTG method, the random method does not rely on coverage information but approaches the selection of parameters for permutation entirely randomly. The selection of parameters for permutation and the value to which it should be permuted is determined randomly. First, a random integer in the range representing all the test case parameters is generated to select a permutation parameter. Secondly, a second random integer is generated within the range of all said parameter values minus one (the one being the previous parameter value - so as not to repeat the same test case).

In each study, a set of 100 test cases is generated. This is done for all three methods and is used on 20 different seeded faults. The entire process is then repeated five times by generating a new set of test cases. The test cases were generated multiple times to account for the random elements in their generation.

⁵ http provided the programs://sir.unl.edu

8

The code-coverage based method generally has the best results. For illustration, the results of each program are show in box-plot Figures 3a, 3b, and 3c.



(a) Flex % of faults found (b) Grep % of faults found (c) GZip % of faults found

Fig. 3: CCTG evaluation results in Box Plot

The Flex results are shown in Figure 3a and have the narrowest range for the code-coverage method the shortest box plot. This indicates that the coverage based test cases are of very similar quality for flex. The code coverage method also has an overall higher median. However, the random method shows a tall box-plot, reflecting a completely random approach to test generation.

Figure 3b for the Grep experiments has all box plots of similar size. This reflects the random elements in all test generation methods. However, it does not produce such dissimilar sizes as in the Flex experiments. The overall dispersion, while similar in size, is marginally more successful for the code coverage method.

Gzip experiment results are shown in Figure 3c. Here the smallest box plot represents the systematic method. While this does not correspond with results from other tests, it is not necessarily a surprise, as the systematic method has the lowest random factor of generation. The code coverage method again holds a marginally better median then the remaining two methods.

In all three cases, the medians are lower compared to the code coverage method. Most distributions in all figures are also not widely dissimilar, indicating even effectively test generating methods. It also shows the code coverage method as the most effective one.

5 Conclusion

This paper presented a new automated test case generation method based on the code coverage measure. The method's goal is to achieve automated test generation using the code coverage, which would also show improved performance at fault detection. Three case studies were implemented that compared out method against two other trivial approaches for test case generation. The results showed an overall improvement in the fault detection rate. The future goal is to work with a wider variety of parameters.

Acknowledgement

This research is conducted as a part of the project TACR TH02010296 Quality Assurance System for the Internet of Things Technology. The authors acknowledge the support of the OP VVV funded project CZ.02.1.01/0.0/0.0/16_019/0000765 Research Center for Informatics.

References

- Ahmed, B.S., Zamli, K.Z., Afzal, W., Bures, M.: Constrained interaction testing: A systematic literature study. IEEE Access 5, 25706–25730 (2017)
- Ahmed, B.S.: Code-aware combinatorial interaction testing. IET Software 13, 600– 609(9) (December 2019)
- Balint, A., Belov, A., Jrvisalo, M., Sinz, C.: Overview and analysis of the sat challenge 2012 solver competition. Artificial Intelligence 223, 120 – 155 (2015)
- Bures, M.: Pctgen: Automated generation of test cases for application workflows. In: Rocha, A., Correia, A.M., Costanzo, S., Reis, L.P. (eds.) New Contributions in Information Systems and Technologies. pp. 789–794. Springer International Publishing, Cham (2015)
- Denisov, A., Pankevich, S.: Mull it over: Mutation testing based on llvm. In: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 25–31 (April 2018)
- Du, Y., Pan, Y., Ao, H., Ottinah Alexander, N., Fan, Y.: Automatic test case generation and optimization based on mutation testing. In: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C). pp. 522–523 (July 2019)
- Gargantini, A., Petke, J., Radavelli, M.: Combinatorial interaction testing for automated constraint repair. In: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 239–248 (March 2017)
- Saumya, C., Koo, J., Kulkarni, M., Bagchi, S.: Xstressor : Automatic generation of large-scale worst-case test inputs by inferring path conditions. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). pp. 1–12 (April 2019)
- Xu, Z., Rothermel, G.: Directed test suite augmentation. In: 2009 16th Asia-Pacific Software Engineering Conference. pp. 406–413 (Dec 2009)
- Xu, Z., Kim, Y., Kim, M., Rothermel, G., Cohen, M.: Directed test suite augmentation: Techniques and tradeoffs. pp. 257–266 (01 2010)
- Zhang, J., Ma, F., Zhang, Z.: Faulty interaction identification via constraint solving and optimization. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing – SAT 2012. pp. 186–199. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)