Int. J. Advance Soft Compu. Appl, Vol. 10, No. 1, March 2018
ISSN 2074-8523

Code Coverage of T-way Test Suite Data on Distributed Environment

Zainal H. C. Soh[1], Syahrul A. C. Abdullah[2], Mohd A. Shafie[1] and Mohammad N. Ibrahim[1]

[1]Faculty of Electrical Engineering, UiTM Pulau Pinang, Penang, Malaysia

e-mail: zainal872@ppinang.uitm.edu.my, mohdaffandi370@ppinang.uitm.edu.my, mnizam@ppinang.uitm.edu.my

[2]Faculty of Electrical Engineering, UiTM Shah Alam, Selangor, Malaysia

e-mail: bekabox181343@salam.uitm.edu.my

Abstract

This paper present a code coverage analysis of a t-way test suite data using tuple space technology on network of PCs. Prior to testing of code coverage, the t-way test suite is generated using a distributed t-way test suite generation strategy, called TS_OP, and the generated test suite is parse with actual test data value into individual test case, testfile. The software under test (SUT) source code is load by Loader Manager along with the testfile that contain the actual test data for executing the test coverage into their respective partition and produces a test coverage result in term of class, method, block and line coverage. A case study of CGPA calculator as SUT is selected to measure the code coverage performance of t-way test suite with varying interaction strength, t on single and multiple machine environments. The distributed implementation analysis of distributed test suite code coverage is also done in term of speedup gained on a multiple machine environments. An encouraging result is obtained on code coverage and speedup for multiple machine environments as compared to single machine environment. Higher test coverage and speedup are obtained in higher machine environments.

Keywords: *T-way Testing, Test Suite Generation and Execution, Code Coverage, Map and Reduce, Tuple Space Technology.*

1 Introduction

The activities involved in software testing ranges from test planning to test conformance monitoring, with the former being the most critical activity as it defines the analysis and design of test data. The challenge in this first phase is in selection and production of quality test case. Lack of quality test can cause the unwanted interaction between software component remained undetected and could cause fatal and costly consequences in future. Many researchers have opted for a *t*-way (*t* indicate the interaction strength) testing system that sampled out a minimize test case and also guarantee to detect faulty interaction among *t* components of software under test. However, this research work extends the t-way testing system by considering a code coverage analysis of test case generated using the distributed t-way testing system on highly configurable software. The proposed distributed t-way testing system is also automated and integrate test suite generation and code coverage within a single system. The distributed implementation of t-way testing system approach is identified to resolve the intense computation. The proposed distributed t-way testing system is to be design and implement on distributed tuple space technology.

In complex and real software world, the number of interaction coverage between software components need to be tested are large and require a lot of resources in term of computing power and human energy to ensure safe and error free software system. The large number of size and combination of input parameters value are likely to be huge and can lead toward the combinatorial explosion problem. Hence, test suite generation will require a lot of computing power and memory resources. Therefore, a distributed test suite generation and test coverage on multiple machine environments is important to expedite the overall testing process.

This paper present a code coverage of distributed t-way test suite generation using tuple space technology on network of PCs. The remainder of the paper is organized as follows. Section 2 gives some insight on the topic and recently published related works. Section 3 describes the Code Coverage of Distributed T-Way Test Suite (CCDT) overall design approach. Section 4 discussed the experimental result of the code coverage of t-way test suite on single and multiple machine. Finally, section 5 draws our conclusions and point out the ideas for future extension of this work.

2 Related work

In *t*-way testing, a lot of research works were focused on test suite generation but not much was found on *t*-way test coverage. There are a few works had been carried out to test the interaction coverage along with the code coverage. K. Burr and W. Young [1] uses x4002cocos to evaluate the effectiveness of AETG [2] as well as to explain the various techniques such as code coverage, table based test automation and t-way test generation used in their testing. Yu Lei [3] implied that combinatorial testing is effective in terms of achieving high code coverage. J. Czerwonka [4] applied code coverage metrics to measure effectiveness of combinatorial test suites in their combinatorial test design. D. R. Kuhn et al [5] describes a variety of measures of combinatorial coverage that can be used in evaluating aspects of t-way coverage of a test suite. They also develop a connection between (static) combinatorial coverage and (dynamic) code coverage, such that if a specific condition is satisfied, 100 % branch coverage is assured. Using these results, they propose practical recommendations for using combinatorial coverage in specifying test requirements, and for improving estimates of the fault detection capacity of a test set.

Currently, D.R Kuhn [6] suggest and describes methods for estimating the coverage of, and ability to detect, t-way interaction faults of a test set based on a covering array. Wang et al [7] indicate that using combinatorial coverage can significantly reduce the number of requests need to be submitted while still achieving effective coverage of the navigation structure as guided by t-way coverage, with respect to code coverage, and found that the navigation structure exploration by Tansuo, in general, results in high code coverage. M. N. Borazjany [8] suggest that fault coverage estimation can be used for initial estimation of test set size, using measures of combinatorial coverage that can be computed with measurement tools and illustrate results of this computation for a variety of test problem configurations. Results can be used in scoping the number of tests and level of effort, and estimating residual risk from complex combinations not tested. In [9] this paper presented the approaches that could be employed for designing the regression test suite using combinatorial approach. They explained how the bench marking of the regression test suite could be done using the traditional approaches such as code coverage in addition to coverage gathered using combinatorial coverage measurement tools.

Above works indicate the useful *t*-way testing should comprise of both the *t*-way test suite generation and test suite execution module in term of fault detection, interaction coverage and test coverage conformance. The generated *t*-way test suite can either be manually or automatically integrated with code coverage test. Currently, only GTWay[10] has integrated an automatic test execution within their test generation strategy where the actual input parameter data is parsed into a symbolic data prior to test generation. This would help to improve the performance of the test suite generation. Before test generation begins, the interaction element is pre-generated and stored in a file where it is used to merge between all possible interaction element using backtracking algorithm during test suite generation. The test case is iteratively generated until all interaction elements are covered. The final test suite which consists of generated test case is then parsed into an actual test data and conveyed as the test data input for test suite execution module. The actual test data is loaded as a stub file and executed with software under test one at a time. After that, the test coverage result is obtained and compared for different interaction strength.

Basically, both the G-MIPOG[11] and MC_MIPOG[12] strategies did address the distributed test suite generation. However, a more thorough investigation revealed a number of limitations. Firstly, both strategies also used tightly coupled computation in generating the test suite. Therefore, any sudden failure involving any of the connected threads could halt the computation of test suite generation process or would produce the wrong test suite results. Secondly, both, G-MIPOG and MC_MIPOG did not address the distributed test suite execution. Finally, MC-MIPOG was found to be not scalable to heterogeneous multiple machine environment for highly distributed environment as it relies on the standard Java Threads library.

As for test suite execution support, the GTWay has integrated an automatic test execution within their test generation strategy. However, GTWay also does not provide distributed test suite execution on multiple machine environments. Having determined and discussed the aforementioned limitations, this paper opted to implement a code coverage of distributed t-way test suite generation and automatically integrated with test suite execution using a tuple space technology that can extend the computing work across multiple machines.

3 The CCDT Design Approach

In this section, a code coverage of distributed t-way test suite (CCDT) overall design approach is presented. In this approach, the main server is called the CCDT Master while the distributed processor is called the CCDT Worker. The CCDT is integrated with a test suite generators known as TS_OP and all generated symbolic test cases of the test suite generator are automatically routed into all participating CCDT Worker dedicated memory spaces using the hash table routing mechanism based on their identification (id) number.

To translate the symbolic test data into actual test data, CCDT Master initiate the conversion of test data by sending a command *parseTestData* at the same time to all the CCDT Worker to convert the symbolic value of each test case with an actual test input data value. Each CCDT Worker parses each test case value at their partitions with their respective actual test input data into an actual test case data and stored as testfile.flt. The test suite fault file, *testfile.flt* contains the class information; method information and a list of test case data value information and is stored in a dedicated partition current directory of a different physical machine.

Then the CCDT Master sends the command *testCodeCoverage* to the CCDT Worker to run the testfile.flt file. Each test case from the *testfile.flt* is loaded using the Loader Manager [11] to create the stubs file and then executes each stub file with loaded source code of SUT and EMMA code coverage tool. The CCDT Worker then iteratively executes the test case one at a time and produce a code coverage result until all the test cases in the testfile.flt are executed. Finally, the accumulated coverage results and then stores them into the *testresult* log file at each partition of participating workstation.

The complete algorithm for code coverage of distributed t-way test suite in CCDT Master is given below:

```
Algorithm CCDT Master (ActualDataSet, Final TS)
begin
1.   send parseTestData command to each CCDT Processor at their respective
     partition space to convert each symbolic data test case from Final TS at
     their partition space into an actual data test case;
2.   send testCodeCoverage command to each CCDT Processor at their
     respective partition space to execute each test case in testfile.flt with code
     driver to test SUT;
     end
```

The complete algorithm code coverage of distributed t-way test suite in CCDT Worker is given below:

```
Algorithm CCDT Worker(ActualDataSet, Final TS)
begin
1.   receive command parseTestData from CCDT Master to parse each
     symbolic data test case from Final TS at their respective partition space into
     an actual data test case;
2.   initialize and open testfile.flt as an empty file;
3.   read symbolic TS from partition space;
4.   while (Symbolic TS is not empty) do
```

```
     begin
5.        parse each test case;
6.        stored parsed test case into testfile;
end
7.   closed testfile;
8.   receive command executeTestData from TSE Master to execute each test
     case in testfile.flt with code driver to test SUT;
9.   initialize and open testresult log file;
10.  load a LoaderManager to create stub file for each test case and execute
     them with SUT and EMMA
11.  while (Test Case in testfile.flt is not empty) do
     begin
12.       create stub file for all test case
13.       execute each test case stub file with loaded SUT;
14.       stored test result into testresult;
          end
15.  closed testresult;
16.  log testresult;
   end
```

4 Result

In this section, there are two experiments have been carried out to access the code coverage performance of the distributed t-way test suite. The first experiment was carried out to measure the performance of distributed t-way test suite code coverage with varying interaction strength on single and multiple machine environments. The second experiment was carried out to access the speedup gained while running on multiple machine environments.

To measure all code coverage metrics and speedup gain on multiple machine environment for the code coverage of distributed t-way test suite strategy, a network of 6 identical and homogeneous PCs with the same operating system (Window 7), processing power (AMD PC Pentium Core 2 2.13GHz) and main memory capacity (4GB of RAM) interconnected using a 2950 Cisco switch was used with the GigaSpaces[13] middleware running on each workstation. The code coverage tool used is an open source EMMA[14] code coverage tool from SourceForge.

The CGPA calculator program was selected as SUT source code. The implemented Java class object of CGPA calculator program made of 1 class, 2 methods, 17069 blocks, and 992 lines. The two methods in the program are the *main()* and the *calculateCGPA()* methods. The *calculateCGPA()* method covers eight parameters of type char that correspond to the four current subject and four of their previous grades and one parameter of type double correspond to the previous semester CGPA point. The input parameter setting here is mixed input parameter as shown in Table 1. All of these experiments were carried out on single and multiple machine environments up to 6 machines.

Table 1: Input Parameters For CGPA Calculator

Input Parameter of CGPA Calculator	**Parameter**								
	Previous CGPA	**Current Grade**				**Previous Grade**			
		C1	*C2*	*C3*	*E1*	*C1*	*C2*	*C3*	*E1*
Parameter Value	2.5	A	A	A	A	D	D	D	D
	2.1	B	B	B	B	F	F	F	F
	2.0	C	C	C	C	-	-	-	-
	1.8	D	D	D	D				
	1.7	F	F	F	F				
	1.6	-	-	-	-				

4.1 Code Coverage with varying number of machines, n

In first experiment, the performance of code coverage of distributed t-way test suite with varying interaction strength on single and multiple machine environments. The result of t-way test suite code coverage for TS_OP test suite generation strategies in single machine environment is shown in Table 2. In a multiple machine environments, the accumulated code coverage results are shown in Table 3. The code coverage result was obtained from the merging or summation of all code coverage in from individual machine.

Table 2: Code Coverage Of CCDT in Single Machine Environment Of TS_OP

Interaction Strength, *t*	**Test Size**	**CCDT_SM**			
		Code Coverage			
		Class (%)	Method (%)	Block (%)	Line (%)
2	50	100	100	61	49
3	306	100	100	80	64
4	1729	100	100	89	75
5	8307	100	100	99	93
6	32138	100	100	100	100

Table 3: Cumulative Code Coverage Percentages for CCDT in Multiple Machine Environments Of TS_OP

Interaction Strength, *t*	**Overall Test Size**	**Test Size per machine**	**CCDT_MM**			
			Code Coverage			
			Class (%)	Method (%)	Block (%)	Line (%)
2	48	8	100	100	63	55
3	314	52	100	100	81	65
4	1735	289	100	100	92	78
5	8308	1384	100	100	99	93
6	32004	5334	100	100	100	100

Using data from Table 2 and Table 3, a graph of interaction strength versus coverage was also plotted for code coverage comparison between single and multiple machines as illustrated in Figure 1.

In comparison between single machine and multiple machines, the code coverage of multiple machines was equal or higher than single machine for similar value of interaction strength, *t*. The CCDT was integrated with TS_OP. For varying interaction strength from *t*=2 until *t*=6, the code coverage consistently produced higher code coverage for higher *t*. This was due to the bigger test suite size generated for multiple machines that led to higher code coverage in both block and line coverage. From Fig. 1, for single machine and multiple machines setting, the graph illustrated that the line and block coverage percentages increased as the interaction strength is increased for *t*=2, 3 and 4 and similar for *t*=5 and 6 whereas for class and method coverage percentages remained constant. For *t*=6, all test coverage criteria recorded 100 percent coverage for software under test.

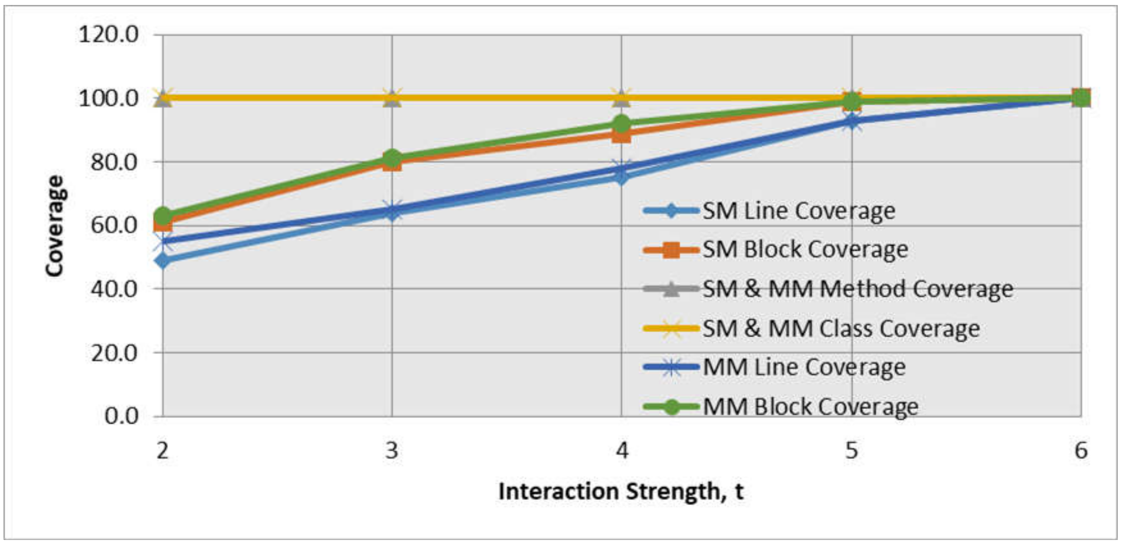

Fig. 1. The percentages of all coverage of SM and MM for CCDT of TS_OP.

For same interaction strength, *t*, the cumulative code coverage in a multiple machine environment was slightly different as compared to the code coverage in a single machine environment. For example, in interaction strength t=3, in single machine, the line coverage is 64% as compare to 65% in multiple machine environment. Slight difference in code coverage result was due to the non-deterministic nature of test suite generation strategy which led to a different test suite size and different individual test case within the generated final test suite. Different individual test case will cover all the specified *t*-way interaction element as well as different higher interaction elements, for example for *t*=2, other than covering all the pairwise interaction strength, the generated test case also covered a few interaction elements of higher *t* (i.e. 3, 4, etc). Thus, the coverage of different higher *t* interaction element within the test suite will lead to different code coverage. The different sets of test cases within the final test suite led to different code coverage

4.2 Speedup Analysis of CCDT

The second experiments were carried out to access the speedup gained for CCDT of TS_OP while running on a multiple machine environment. The input parameter used is fixed as depicted in Table 1 with interaction strength of 4. The result for testing time in single and multiple machines and with test size of their respective number of machine settings are shown in Table 4.

The overall testing time is made up of the test suite generation time and test suite execution time. The testing time speedup was obtained by dividing the single machine testing with multiple machine testing time. Using data in Table IV, the testing time speedup for CCDT of TS_OP is plotted as shown in Fig. 2. From Fig. 2, it can be deduced that the testing time in multiple machine will always produce a speedup as compared to single machine setting for TSE with TS_OP. However, the speedup is less significant for higher number of machines and nearly identical to other multiple machine environments.

Table 4 : Test Size Ratio and Speedup for CCDT with varying *n*

Number of machine, *n*	**CCDT**		**CCDT(SM/MM)**	
	Test Size	Testing Time	Test Size Ratio	Testing Time Speedup
1	1729	2988.53	1.00	1.00
2	1729	1862.56	1.00	1.60
3	1716	1696.34	1.01	1.76
4	1729	1653.97	1.00	1.81
5	1728	1800.28	1.00	1.66
6	1728	1670.28	1.00	1.79

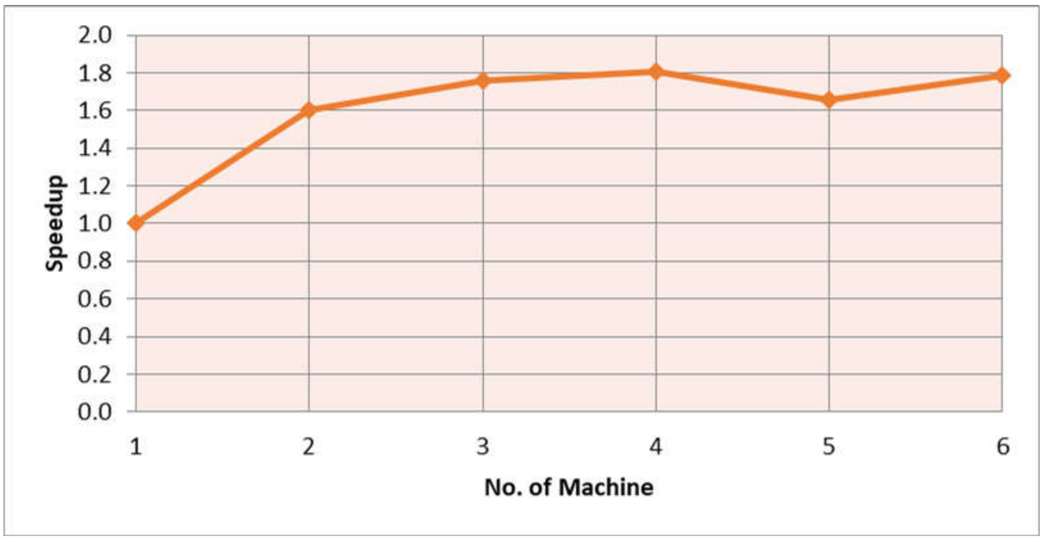

Fig. 2. The CCDT of TS_OP speedup vs number of machine.

5 Conclusion

This paper present a code coverage analysis for distributed t-way test suite evaluation with varying interaction strength from *t*=2 until *t*=6, the code coverage consistently produced higher code coverage for higher *t* in multiple machine environments. In a distributed environment setting, the cumulative code coverage in a multiple machine environment was slightly higher and different as compared to the code coverage in a single machine environment for the same interaction strength, *t*. Slight higher in code coverage result was due to the non-deterministic nature of test generation which led to a different test suite size and different individual test case within that test suite. The different sets of test cases within the test suite led to different code coverage. However, using code coverage in multiple machine and distributed environment did not reduce the code coverage results but instead due to the bigger size of test suite, the code coverage is increased in both block coverage and line coverage. Although, there was small amount of speedup in terms of the overall testing time with the increment number of PCs, the testing is always faster in multiple machine environment. In future works, the code coverage can be further test along with fault detection can be implemented on virtual cluster machines with large number of CPU with high RAM memory to minimise the network latency. This allow the system to support higher interaction strength with large input parameter within easy and controllable environment.

Acknowledgement

This research is partially funded by Lestari grant – “Variable Strength t-way Test Generation Strategy with Crash Recovery Support” (File No: 600-RMI/DANA 5/3/LESTARI (102/2015)) from Institute of Research Management & Innovation (IRMI), Universiti Teknologi MARA (UiTM), Malaysia. This research is also partially funded by the generous EScience fund grant – “Distributed T-Way Testing System using MapReduce Mechanism on Hadoop Cluster” from MOSTI and UiTM.

References

[1] K. Burr and W. Young, "Combinatorial test techniques: Table-based automation, test generation and code coverage," in Proc. of the Intl. Conf. on Software Testing Analysis & Review, 1998.

[2] M. Cohen, *et al.*, "The AETG system: an approach to testing based on combinatorial design," *Software Engineering, IEEE Transactions on,* vol. 23, pp. 437-444, 1997.

[3] Y. Lei*, et al.*, "IPOG: A General Strategy for T-Way Software Testing," in *Engineering of Computer-Based Systems, 2007. ECBS '07. 14th Annual IEEE International Conference and Workshops on the*, 2007, pp. 549-556.

[4] J. Czerwonka, "On Use of Coverage Metrics in Assessing Effectiveness of Combinatorial Test Designs," 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, Luxembourg, 2013, pp. 257-266.

[5] D. R. Kuhn, I. D. Mendoza, R. N. Kacker and Y. Lei, "Combinatorial Coverage Measurement Concepts and Applications," 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, Luxembourg, 2013, pp. 352-361

[6] Kuhn, D. R., Kacker, R. N., & Lei, Y. (2016). Measuring and specifying combinatorial coverage of test input configurations. Innovations in Systems and Software Engineering, 12(4), 249-261.

[7] Wang, W., Sampath, S., Lei, Y., Kacker, R., Kuhn, R., and Lawrence, J. (2016) Using combinatorial testing to build navigation graphs for dynamic web applications. Softw. Test. Verif. Reliab., 26: 318–346.

[8] M. N. Borazjany, L. Yu, Y. Lei, R. Kacker and R. Kuhn, "Combinatorial Testing of ACTS: A Case Study," 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, QC, 2012, pp. 591-600.D.

[9] H. Patil, A. , Goveas, N. and Rangarajan, K. (2015) Test Suite Design Methodology Using Combinatorial Approach for Internet of Things Operating Systems. Journal of Software Engineering and Applications, 8, 303-312

[10] M. F. J. Klaib*, et al.*, "G2Way A Backtracking Strategy for Pairwise Test Data Generation," in *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*, 2008, pp. 463-470.

[11] M. I. Younis*, et al.*, "A strategy for Grid based t-way test data generation," in *Distributed Framework and Applications, 2008. DFmA 2008. First International Conference on*, 2008, pp. 73-78.

[12] M. I. Younis and K. Z. Zamli, "MC-MIPOG: A Parallel t-Way Test Generation Strategy for Multicore Systems," *ETRI Journal,* vol. 32, pp. 73-83, Feb 2010 2010.

[13] GigaSpaces. (2011). *Website for GigaSpaces.*

[14] V. Roubtsov. (2012). *EMMA: a free Java code coverage tool.*