Towards Quality Improvement and Analysis of Combinatorial Testing

Eun-Hye Choi[1,a)] Osamu Mizuno[2,b)]

Abstract: Combinatorial testing is a widely-used testing technique to detect system failures caused by parameter interactions. This paper introduces our ongoing work to develop a systematic intelligent testing framework, which aims at improving and evaluating combinatorial testing by mining and analyzing software repository data.

Keywords: Combinatorial testing, *t*-way testing, SUT modeling, Prioritized test generation, Software repository.

1. Introduction

Combinatorial testing is a well-known black-box testing technique [18]. The process of combinatorial testing consists of (1) modeling of the *system under test (SUT)*, (2) constructing a combinatorial test suite from given the SUT, and (3) executing the test suite and analyzing the result. Although a lot of approaches have been proposed so far especially for test generation, there are still important problems remained for constructing a systematic and intelligent testing framework, which includes automatic SUT modeling, generation of test suites with higher quality, and thorough test analysis for real software systems.

Our goal is developing a fully automated testing framework illustrated in Fig. 1, which addresses the problems mentioned above, by mining and analyzing data in the software repository such as programs, tests, and bug reports. In this paper, we introduce our ongoing work on test modeling, design, and analysis in Sections 2, 3, and 4, respectively.

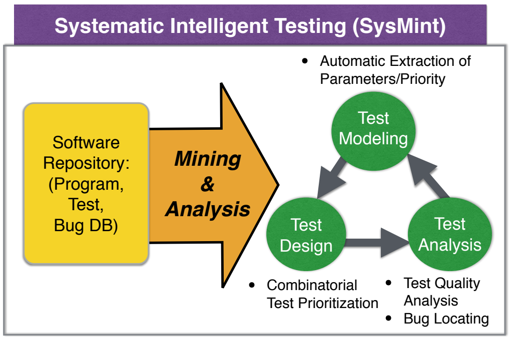

Fig. 1 Our testing framework.

Table 1 An SUT model.

Parameter	*Value*; *Weight*
<u>O</u>S	<u>W</u>in;2, <u>M</u>ac;1
<u>N</u>et	<u>W</u>ifi;1, <u>L</u>AN;1
<u>B</u>rowser	<u>I</u>E;1, <u>F</u>irefox;1, <u>C</u>hrome;4
Constraint	
OS=Mac → Browser≠IE	

Table 2 A pairwise test suite.

test	O	N	B
1	W	W	C
2	W	L	C
3	W	W	I
4	W	L	F
5	M	W	F
6	M	L	I

2. Combinatorial Test Modeling

The *System Under Test (SUT)* for combinatorial testing (CT) is generally modeled from parameters, their associated values from finite sets, and constraints between parameter-values. For example, the SUT model in Tab. 1 has three parameters; the first two parameters have two possible values and the other has three possibilities. Constraints among parameter-values exist in the SUT model when some parameter-value combinations cannot occur. The example SUT has a constraint such that (Mac, IE) is not allowed. *Prioritized* combinatorial testing (e. g., [2], [7], [9]), which aims at increasing the quality of combinatorial testing, takes SUT models with priority weights assigned to parameter-values. In the example SUT, the weight for *Win* is higher than that for *Mac*. Such a weight represents a relative importance in testing, e. g., occurrence probability, error probability, or risk.

To design combinatorial testing, we first need to construct the SUT model. However, in most cases, SUT modeling is manually performed and thus is burdensome for modern complicated systems. To support systematic modeling of SUTs, previous work [14], [15] proposed tree-structured modeling tools. There is another work [17], which presents an automatic elicitation of potential SUT constraints from documents.

Towards automatic SUT modeling, we are developing an automatic extraction of SUT priority weights from bug detection history [12] and code coverage of programs. Automatic extraction of SUT parameters, values, and constraints from previous test information and programs in software repository is included in our further work.

3. Combinatorial Test Design

A combinatorial *t-way test suite* (e. g., *pairwise*, when $t = 2$) for an SUT model is a test sequence to cover all possible *t*-way combinations of parameter-values that satisfies the constraint in

1 National Institute of Advanced Industrial Science and Technology (AIST), Ikeda, Japan
2 Kyoto Institute of Technology, Kyoto, Japan
a) e.choi@aist.go.jp
b) o-mizuno@kit.ac.jp

the SUT model at least once. Table 2 shows an example pairwise test suite for the SUT model in Table 1; it covers all possible 15 value pairs between two parameters.

Many algorithms and tools to efficiently construct small combinatorial test suites have been proposed so far [13]. Approaches to generate t-way test suites for SUT models with constraints include greedy algorithms (e. g., PICT [7] and ACTS [1]), heuristic search (e. g., CASA [10] and TCA [16]), BDD-based (e. g., FoCus [20]), and SAT-based approaches (e. g., Calot [21]). For prioritized t-way testing, several algorithms have been proposed, which generate a test suite where highly weighted parameter-values appear earlier [2], [15] or more frequently [9], [20].

To improve the quality of t-way testing under the limited testing resource, we are developing prioritized t-way generation algorithms. In [5], we proposed a priority-integrated combinatorial testing (called *pricot*), which generates small-sized test suites providing high-priority test cases early and frequently in a good balance. In [3], we presented a distance-integrated combinatorial testing (called *dicot*), which generates t-way test suites that achieve higher interaction coverage for higher interaction strengths t with low computational overhead by increasing not only the number of new combinations but also the *distance* (e. g., Hamming distance or a modified chi-square distance) between test cases.

4. Combinatorial Test Analysis

Fault detection abilities of t-way testing have been reported by several empirical studies so far [11], [22]. The results have shown that t-way testing with relatively small t (≤ 6) can detect most failures while reducing the number of test cases significantly compared to exhaustive (i. e., all combination) testing.

In [6], we further investigated the effectiveness of t-way testing on *code coverage*, which is one of the most important coverage criteria widely used for software testing. Our results using a collection of open source utility programs from the Software-artifact Infrastructure Repository (SIR) [8] showed that t-way testing with small t ($1 \leq t \leq 4$) efficiently covers more than 95% of code coverage achieved by exhaustive testing.

In [4], we investigated the fault detection effectiveness of prioritized combinatorial testing on the collection of open source utilities. Prioritized combinatorial test generation algorithms are classified into order-focused ([2], [15]) and frequency-focused ([7], [9]) approaches and their integration which we proposed in [5]. The algorithms have been evaluated using metrics called *weight coverage* and *KL divergence* but not sufficiently with the fault detection effectiveness. We presented a case study that evaluates the fault detection effectiveness with weight coverage and KL divergence and analyzes the correlation between them.

In addition, we are developing a method to locate faulty interactions from combinatorial test suites and testing results using machine learning [19].

5. Conclusion

Towards the quality improvement and evaluation of combinatorial testing, we are developing a testing framework aiming at fully automated combinatorial testing and analysis. In this paper, we introduced our work for this purpose. For SUT modeling, we are developing an automatic priority extraction. For test design, we are developing prioritized test generation algorithms that provide higher quality. For test analysis, we are evaluating multiple metrics for (prioritized) combinatorial testing. We are also developing a bug localization from the combinatorial tests and results.

Acknowledgments

This work is partly supported by JSPS KAKENHI Grant Number 16K12415.

References

[1] Borazjany, M. N., Yu, L., Lei, Y., Kacker, R. and Kuhn, R.: Combinatorial Testing of ACTS: A Case Study, *Proc. of ICST*, IEEE, pp. 591–600 (2012).

[2] Bryce, R. and Colbourn, C.: Prioritized interaction testing for pairwise coverage with seeding and constraints, *Information & Software Technology*, Vol. 48, No. 10, pp. 960–970 (2006).

[3] Choi, E., Artho, C., Kitamura, T., Mizuno, O. and Yamada, A.: Distance-Integrated Combinatorial Testing, *Proc. of ISSRE*, IEEE, pp. 93–104 (2016).

[4] Choi, E., Kawabata, S., Mizuno, O., Artho, C. and Kitamura, T.: Test Effectiveness Evaluation of Prioritized Combinatorial Testing: a Case Study, *Proc. of QRS*, IEEE, pp. 61–68 (2016).

[5] Choi, E., Kitamura, T., Artho, C., Yamada, A. and Oiwa, Y.: Priority Integration for Weighted Combinatorial Testing, *Proc. of COMPSAC*, IEEE, pp. 242–247 (2015).

[6] Choi, E., Mizuno, O. and Hu, Y.: Code Coverage Analysis of Combinatorial Testing, *Proc. of QUASoQ*, CEUR-WS, pp. 34–40 (2016).

[7] Czerwonka, J.: Pairwise Testing in the Real World: Practical Extensions to Test Case Generators, *Microsoft Corporation, Software Testing Technical Articles* (2008).

[8] Do, H., Elbaum, S. and Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact, *Empirical Software Engineering*, Vol. 10, No. 4, pp. 405–435 (2005).

[9] Fujimoto, S., Kojima, H. and Tsuchiya, T.: A Value Weighting Method for Pair-Wise Testing, *Proc. of APSEC*, pp. 99–105 (2013).

[10] Garvin, B. J., Cohen, M. B. and Dwyer, M. B.: Evaluating improvements to a meta-heuristic search for constrained interaction testing, *Empirical Software Engineering*, Vol. 16, No. 1, pp. 61–102 (2011).

[11] Kacker, R. N., Kuhn, D. R., Lei, Y. and Lawrence, J. F.: Combinatorial testing for software: An adaptation of design of experiments, *Measurement*, Vol. 46, No. 9, pp. 3745–3752 (2013).

[12] Kawabata, S., Choi, E. and Mizuno, O.: A Prioritization of Combinatorial Testing using Bayesian Inference, *Technical Report of IEICE (in Japanese)*, Vol. 115, No. SS2015-95, pp. 115–120 (2016).

[13] Khalsa, S. K. and Labiche, Y.: An orchestrated survey of available algorithms and tools for combinatorial testing, *Proc. of ISSRE*, IEEE, pp. 323–334 (2014).

[14] Kitamura, T., Yamada, A., Hatayama, G., Artho, C., Choi, E., Do, N. T. B., Oiwa, Y. and Sakuragi, S.: Combinatorial Testing for Tree-structured Test Models with Constraints, *Proc. of QRS*, IEEE, pp. 141–150 (2015).

[15] Kruse, P. and Luniak, M.: Automated Test Case Generation Using Classification Trees, *Software Quality Professional*, pp. 4–12 (2010).

[16] Lin, J., Luo, C., Cai, S., Su, K., Hao, D. and Zhang, L.: TCA: An Efficient Two-Mode Meta-Heuristic Algorithm for Combinatorial Test Generation, *Proc. of ASE*, ACM/IEEE, pp. 494–505 (2015).

[17] Nakagawa, H. and Tsuchiya, T.: A Search-based Constraint Elicitation in Test Design, *IEICE Transactions on Information and Systems*, No. 9, pp. 2229–2238 (2016).

[18] Nie, C. and Leung, H.: A survey of combinatorial testing, *ACM Computing Surveys*, Vol. 43, No. 2, p. 11 (2011).

[19] Nishiura, K., Choi, E. and Mizuno, O.: Fault Localization of Combinatorial Testing with Logistic Regression, *Proc. of FOSE (in Japanese)*, JSSST, pp. 243–244 (2016).

[20] Segall, I., Tzoref-Brill, R. and Farchi, E.: Using binary decision diagrams for combinatorial test design, *Proc. of ISSTA*, pp. 254–264 (2011).

[21] Yamada, A., Kitamura, T., Artho, C., Choi, E., Oiwa, Y. and Biere, A.: Optimization of Combinatorial Testing by Incremental SAT Solving, *Proc. of ICST*, IEEE, pp. 1–10 (2015).

[22] Zhang, Z., Liu, X. and Zhang, J.: Combinatorial testing on id3v2 tags of mp3 files, *Proc. of ICST*, IEEE, pp. 587–590 (2012).