Adaptive Combinatorial Testing

Changhai Nie
State Key Laboratory for Novel Software Technology
Nanjing University, China, 210093
changhainie@nju.edu.cn

Hareton Leung
Department of computing
Hong Kong Polytechnic University
cshleung@inet.polyu.edu.hk

Kai-Yuan Cai
Department of Automatic Control
Beijing University of Aeronautics and Astronautics, Beijing, China, 100083
kycai@buaa.edu.cn

***Abstract*—Combinatorial Testing (CT) has been proven to be effective in detecting and locating the interaction triggered failure in the last 20 years. But CT still suffers from many challenges, such as modeling for CT, sampling mechanisms for test generation, applicability and effectiveness. To overcome these issues of CT, adaptive combinatorial testing (ACT) is proposed in this paper, which improves the traditional CT with a well established adaptive testing method as the counter part of adaptive control and aims to make CT more flexible and practical. ACT can significantly enhance testing quality, software reliability and support testing strategy adjustment dynamically. To support further investigation, a preliminary form of concrete strategy for ACT is given as a heuristic guideline.**

***Index Terms*—Software Testing, Adaptive Testing, Combinatorial Testing, Adaptive Combinatorial Testing.**

I. INTRODUCTION

Combinatorial Testing (CT) aims to detect failure triggered by the interaction of different factors, such as inputs, features or components systematically. CT uses as small a test set as possible to cover as many value combinations of the factors in the Software Under Test (SUT) as possible. CT is normally static as it generates the test set before running the tests [1]. The expected performance of CT in practice has not been manifested very well due to the following problems [2]: 1) fail to identify the correct factors that may affect SUT, and fail to select the proper values for each factor; 2) do not know what kind of test suite should be used; 3) fail to identify constraints in SUT, which may invalid many test cases, and then miss the failure causing combinations; 4) cannot integrate the fault detection and location in CT; 5) cannot automate CT with the maximized efficacy, such as running test cases in a prioritized order, especially when the test strategy cannot be updated dynamically .

Cai has pointed out that the test set should not be fixed before test execution and should be adjusted in testing. He proposed Adaptive Testing (AT) to generate test cases in the process of test execution, guided by the collected test history, new understanding of SUT and some other heuristics. Cai has demonstrated that AT can be more effective in achieving high software quality in many situations [3]. AT has been applied in improving web service testing [6] and component testing [4], but it has not been applied to enhance CT. So we will enhance CT with AT concept and extend CT to be adaptive, called ACT.

In section 2 we introduce the background and related works; we present a preliminary ACT based on some existing work in section 3; lastly we offer our conclusion and the future work.

II. BACKGROUND AND RELATED WORK

A. Combinatorial Testing

Let a SUT has k parameters (factors) $p_i \in \mathrm{P}$, each p_i has a_i values (levels) in $V_i \in V$, $|V_i| = a_i$, $(1 \le i \le k)$. Let R be an interaction relation set, which includes all the subsets of parameters that may interact with each other. For example, if $\{p_1, p_3\} \in R$, parameter p_1 and p_3 can interact with each other by some combination in $V_1 \times V_3$. Let C be a constraints set, which includes all the parameter combinations which should be forbiden in testing.

For SUT, k-tuple $(v_1, v_2, ...y_k)$, $v_i \in V_i, (1 \le i \le k)$ is called a test case. And there are totally $a_1 \times a_2 \times .. \times a_k$ test cases without considering constraints. If a forbiden combination $Comb$ formed by $v_l \in V_l$, $v_m \in V_m$ and $v_n \in V_n$, denoted as $Comb$= (-, … , v_l , …, v_m, … , v_n, … ,-), and $Comb \in C$, the number of total usable test cases must exclude all the test cases that include this combination. Here we can see that a key problem of CT is that SUT has a huge usable test cases space, which is almost impossible to test all. So CT tries to sample a small number of test cases known as covering array.

Definition 1 t-way CT for SUT uses a combinatorial object called t-way covering array: $MCA(N; t, k, (a_1, a_2, \cdots, a_k))$, as test set, where t is the strength of coverage, k is the number of factors, and a_i is the number of levels for each factor p_i. The covering array is an $N \times k$, with the following properties [2]: (1) Each column i $(1 \le i \le k)$ contains only elements from a set V_i of parameter p_i with $|V_i| = a_i$; (2) The rows of each sub-array $N \times \mathrm{t}$ cover all t-tuples of values from the columns at least once. When $a_1=a_2=…=a_k=v$, the t-way covering array can be denoted as $CA(N; t, k, v)$.

TABLE Ⅰ: THE ONLINE SYSTEM, sut

Web Brow.	Oper. Sys.	Conn. Type	Mem.
Netscape	Windows	LAN	256MB
IE	Macintosh	PPP	512MB
Mozilla	Linux	ISDN	1GB

For instance, an Internet-based software system sut may support end-users with a variety of web browsers, operating systems, connection types and memory configurations, as

978-0-7695-5039-8/13 $26.00 © 2013 IEEE
DOI 10.1109/QSIC.2013.22

shown in Table Ⅰ. To exhaustively test all possible combinations needs 3^4=81 test cases. By employing 2-way CT (Table Ⅱ), we can reduce the number of test cases for testing sut to 9.

TABLE Ⅱ: A 2-way COVERING ARRAY FOR sut

No.	Web Brows.	Oper. Sys.	Conn.Type	Mem.
1	Netscape	Windows	LAN	256MB
2	IE	Macintosh	LAN	512MB
3	Netscape	Macintosh	PPP	1GB
4	Mozilla	Linux	LAN	1GB
5	Netscape	Linux	ISDN	512MB
6	IE	Linux	PPP	256MB
7	Mozilla	Windows	PPP	512MB
8	IE	Windows	ISDN	1GB
9	Mozilla	Macintosh	ISDN	256MB

Another important problem for CT is when a test case finds a defect, how to identify which combination in this test case trigger the failure? For example, a defect is found when we test *sut* with (Netscape, Windows, LAN, 256MB). As this test case includes the follow combinations: (Netscape, -, -, -), (-, Windows, -, -), (-, -, LAN, -), (-, -, -, 256MB), (Netscape, Windows, -, -), (-, Windows, LAN, -), (-, -, LAN, 256MB), (Netscape, -, LAN, -), (-, Windows, -, 256MB), (Netscape, -, -, 256MB), (-, Windows, LAN, 256MB), (Netscape, -, LAN, 256MB), (Netscape, Windows, -, 256MB) and (Netscape, Windows, LAN, -), totally 2^4-1=15, it is a challenge to identify the failure-introducing combination from the failed test case during failure diagnosis.

A typical test procedure for CT has been given [1] as shown in Figure1. It has the following practical problems:

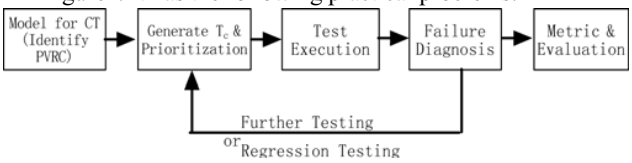

Fig. 1. A Traditional Testing Procedure for CT

- The first step is to identify all the parameters P, all the values for each parameter V, the relation between parameters R and the constraints among parameters C. This is a very important step which will determine the effectiveness of CT. But it is almost impossible to build a perfect model at the beginning of CT. Modeling for CT should be refined as testing progresses and be adaptive.
- At the second step, a test set is generated and ordered off-line. We normally do not know what kind of covering array we should generate, because we cannot determine R and C completely. Also we do not know how to permute the test cases due to lack of valuable run-time information. So the second step should be adjusted online based on dynamic information.
- Test execution sequence is fixed and determined by the second step, without consideration of any usable information collected during execution. The test execution information is often collected on-line, but analyzed off-line.
- Failure diagnosis is usually independent and requires further testing. It does not integrate with the former steps and makes use of related execution information. Its cost needs to be reduced and efficacy should be improved.

All the above problems imply that CT should be adaptive.

B. Adaptive Testing (AT)

AT is a relatively new software testing technique proposed by Cai [3], which results from the application of feedback and adaptive control principles in software testing. It can be treated as the software testing counterpart of adaptive control.

AT can be understood as a framework as shown in Figure.2. A_t represents an action or a test case, and Z_t represents the test result. A_t and Z_t form part of the test history H_t, that is, H_t={A_1, Z_1, A_2, Z_2, …, A_t, Z_t} The software under test serves as the controlled object modeled as a controlled Markov chain, whereas the software testing strategy serves as the corresponding controller. Unlike conventional CT, which is applied to the software under test without an explicit optimization goal, AT designs an optimal testing strategy to achieve an explicit optimization goal given a priori. At the beginning of software testing, our knowledge of the SUT is limited. As the software testing proceeds, more testing data are collected and our understanding of the SUT is improved. Software parameters of concern may be estimated and updated, and the software testing strategy is accordingly adjusted dynamically.

There are two feedback loops in the adaptive testing framework in Figure 2. The first feedback loop consists of the SUT, the database (history of testing data) and the testing strategy, in which the history of testing data is used to generate the next test cases by a given testing policy or test data adequacy criterion. The second feedback loop consists of the SUT, the database, the parameter estimation scheme and the testing strategy, in which the history of testing data is also used to improve or change the underlying testing policy or test data adequacy criterion.

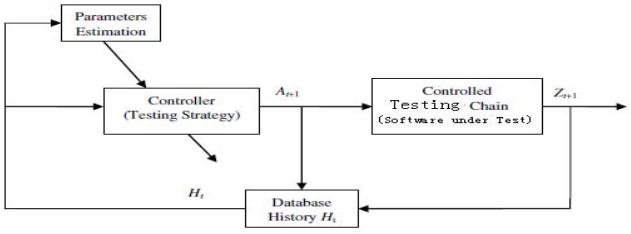

Fig. 2. Framework of Adaptive Testing

C. Related work

To enhance random testing, Chen et al proposed adaptive random testing [10]. ART is based on various empirical observations and make use of any available information to guide random testing.

As services keep on changing, testing needs to be adaptive. Bai et al. proposed an adaptive testing framework which can continuously learn and improve the built-in test strategies. This framework allows different test cases to be selected based on

the recent test results, and uses a windowing mechanism to evaluate and select test cases [6].

Yuan et al used adaptive test generation for GUI testing. Feedback is used to uncover constraints between events resulting in fewer infeasible test cases and higher test adequacy [11].

Cai et al have shown that the software component should be tested in an adaptive manner in the sense that the software defect detection rates are estimated on-line by using testing data collected during testing to improve test case selections [4].

III. ADAPTIVE COMBINATORIAL TESTING

In this section, we present our preliminary concrete design for ACT. Although CT has employed some adaptive feature, and the model of AT built by Cai has been well defined and studied, AT has not been applied to CT. We next give the detail design of each component in Figure 2 for ACT. Firstly, we give the meaning of notations. A_t represents a test case of SUT like $(v_1,v_2,\ldots,v_k)$, Z_t is its testing result, which has three possible values: failed, passed and invalid. H_t is the test history consists of test case and its execution result, that is, $H_t=\{A_1, Z_1, A_2, Z_2, \ldots, A_t, Z_t\}$. Next we will present each component of ACT.

A. Parameter Estimation

In the parameter estimation component, an early model of SUT for CT will be determined first. It includes Parameters, Values for each parameter, the interaction Relations among parameters and their Constraints (called PVRC model). This initial information can be obtained by reviewing some development documents, communicating with programmers and designers or through static analysis, such as slicing. This model will be refined and updated during testing. There are some algorithms to learn and analyze the test history and rebuild the PVRC model for CT online. The updated model will be used to change the current test strategy.

B. Controller

The controller component is a kernel part of ACT. At the beginning, it generates test suite with the *increasing adaptive covering array*, then if a test case failed in testing, it invokes the *failed test case oriented strategy*; if a generated test case is invalid, it invokes the *constraint oriented strategy*.

1) Increasing adaptive covering array: As the traditional CT requires choosing the strength of covering array first, but there is no scientific or historical basis for doing so. If the strength is set too large, resources will be wasted or the testing process may not complete before the next release; if the strength is set too small, the covering array will be insufficient to classify specific failures, and the entire testing process may need to be rerun from scratch. To address these issues, Fouche et al proposed to use incremental adaptive covering array in CT. It begins with a low strength, and continually increases it as resources allow, or as required from the poor classification results. At each stage, the previous tests are reused. This allows failures due to only one or two configuration settings to be found and classified as early as possible, and also limits duplication of work when multiple covering arrays must be used [5].

For example, to test the online system *sut*, we can first test it with test cases shown in Table Ⅰ, a 1-way covering array that covers all the parameter values. Then we generate 8 additional test cases as shown in Table Ⅱ to achieve 2-way covering array together with Table Ⅰ (except test case 1, which is same as the first test case in Table Ⅰ).

Increasing adaptive covering array can have different implementation; for example, we can use another way to implement this kind of incremental testing as follows: firstly, generate a t'-way covering array (t'>t); when testing SUT, select each test case with a greedy algorithm from this t'-way covering array to achieve a t-way covering as soon (small) as possible; then if SUT still needs more test cases, continue select additional test cases from this t'-way covering array to achieve t+1-way covering array. For example, Table 2 gives a 2-way covering array. We can first run test case 1, 2, 4, 6, 8 to achieve 1-way testing, and then run the remaining test cases 3, 5, 7, 9 to achieve 2-way testing.

Different implementations of *increasing adaptive covering array* have different properties, such as total cost and the rate of fault detection. More implementations and their properties will be studied in the future.

2) Failed test case oriented strategy: this strategy aims to avoid missing some combination from testing as proposed in [7] and locate the failure inducing combination [2, 8].

For masking effects – failures that perturb execution so as to prevent some behaviors from being exercised, Emine et al presented a feedback driven adaptive combinatorial testing approach to prevent the harmful consequences of masking effects. At each iteration, the approach detects potential masking effects, isolates their likely causes, and then schedules the set of t-way option combinations that are being masked for testing in the subsequent iteration. The process iterates until each and every t-way option setting combination is present in at least one configuration in which the test passed or failed with a non-option-related cause, or the combination is marked as failure inducing [7].

For example, if *sut* failed only in the running of the third test case Ct3= (Netscape, Macintosh, PPP, 1GB) in Table Ⅱ, the schema （Netscape,-,-,-）, (-,Macintosh,-,-), (-,-,PPP,-) and (-,-,-,1GB) were not masked because each of them had been presented in the passed test cases in Table Ⅱ. But (Netscape, Macintosh, -, -), (-, Macintosh, PPP, -), (-, -, PPP, 1GB), (Netscape, -, PPP, -), (-, Macintosh, -, 1GB) and (Netscape, -, -, 1GB) were masked, because they were not presented in any passed test case and were just covered by the failed test case where none of the combinations was marked as failure-inducing. To achieve 2-way coverage, we need to regenerate some new test cases to cover these masked 2-way combinations.

Fault location for CT is actually a procedure of fault characterization. It identifies which combination can trigger a failure. The information of fault characterization can help diagnose the root cause of a failure. As mentioned in section Ⅱ, a failed test case $(v_1, v_2, \ldots, v_k)$ of SUT can have 2^k-1

different combinations that may trigger the failure. Thus, it is a big challenge to locate fault.

We have used adaptive testing in fault location for CT [2, 8]. For a failed test case $(v_1,v_2,\ldots,v_k)$ of SUT, we design a set of k additional test cases, like $(*,v_2,\ldots,v_k)$, $(v_1,*,\ldots,v_k)$, and $(v_1,v_2,\ldots,*)$, where * represents a new value to replace the original one in the failed test case. Then we test SUT with this additional test set; if some test cases failed again, we continue testing SUT for each failed test case in the same way. The procedure is iterated until the failure causing combination can be identified. [9, 12] proposed some improved methods for additional test set generation based on Delta debugging and suspicious ranking respectively.

3) Constraint oriented strategy: this strategy detects the latent constraints missing from the early model of SUT with some additional test case. If a test case becomes invalid in testing, it means that there is some unknown constraint when we generate test cases by the model of SUT. We should find out this constraint. After we identify the missing constraint, we need to design some new test cases to test all the valid combinations masked by the invalid test case.

For example, if a test case $(v_1,v_2,\ldots,v_k)$ of SUT is invalid, to identify the constraint, we design a set of k additional test cases, like $(*,v_2,\ldots,v_k)$, $(v_1,*,\ldots,v_k)$, and $(v_1,v_2,\ldots,*)$, where * represents a new value to replace the original one in the invalid test case. We run this test set, analyze its result，according to the testing result of each test case (which can be passed, failed or invalid), then make further testing in the same way for the new invalid test case until we can identify the constraint and test all the valid combinations. More effective methods need to be developed to handle this issue.

C. Controlled Markov Chain

The controlled Markov chain component gives the procedure of running SUT with the generated testing cases. Testing result is observed in this step. We should study the information (besides failed, passed or invalid) that is useful for analysis and further application.

D. Database History

The database history component collects test history data and makes preliminary analysis. It includes failure-inducing analysis and constraints analysis. Parameters and their values are also refined in this part.

E. Advantages of ACT

ACT employs an adaptive mechanism in CT, with factors and their values, constraints and testing strategy adjusted dynamically. It can refine factors and their values to make them more appropriate, and deal with constraints to avoid the missing of related combinations. It uses a dynamically increasing test set, called incremental adaptive covering array to test in an economical way. When a defect is detected, it will be tuned to be fault location oriented testing.

ACT improves the traditional CT with adaptive mechanism and can be viewed as a combination of CT and AT or the application of AT in CT. ACT aims to make CT more flexible and practical. It can significantly enhance testing quality, software reliability and yield learning from the testing history.

IV. CONCLUSION AND FUTURE WORK

In this paper we proposed a model for ACT, which improves CT with the well studied AT, and briefly reviewed the existing adaptive mechanisms in CT, such as incremental adaptive covering array [5], adaptive fault location [8,9] and feedback driven adaptive CT [7]. The proposed ACT enhances the procedure of CT to be adaptive, that is, modeling of SUT, test strategies and fault location methods should be adjusted in an adaptive way. This is important and necessary to support the complete automation of CT with the aim to achieve maximized efficiency.

To make ACT more practical, more advanced adaptive mechanisms need to be developed in the future, which should address how to evolve the model of SUT, how to test SUT in an incremental way and how to locate the faults effectively. More evidence should be collected in empirical studies and more powerful support tools need to be developed.

REFERENCES

[1] Nie, C and Leung H. 2011. A Survey of Combinatorial Testing. ACM Computing Survey, 43(2), Article 11, 1-29

[2] Nie, C. and Leung H. 2011. Minimal Failure Causing Schema for Combinatorial Testing. ACM Transaction on Software Engineering and Methodology, 20(4), Article 15, 1-38

[3] Cai, K. 2002 Optimal software testing and adaptive software testing in the context of software cybernetics, Information and Software Technology, Vol. 44, pp. 841–855.

[4] Cai, K., Chen, T. Y., Li, Y., Ning, W. and Yu, Y. T. 2005: Adaptive testing of software components. SAC: 1463-1469

[5] Fouche, S., Cohen, M. B. and Porter, A. 2009 Incremental covering array failure characterization in large configuration spaces, in Proc. ISSTA, 2009, pp.177-188.

[6] Bai, X., Chen, Y. and Shao, Z. 2007. Adaptive Web Services Testing. COMPSAC (2) 2007: 233-236

[7] Dumlu, E., Yilmaz, C., Cohen, M. B. and Porter, A. A. 2011 Feedback driven adaptive combinatorial testing. In Proceedings of ISSTA'2011. pp.243-253

[8] Shi, L., Nie, C. and Xu, B. 2005. A software debugging method based on pairwise testing. In Proceedings of the International Conference on Computational Science (ICCS2005), pages 1088-1091, 2005.

[9] Zhang, Z. and Zhang, J. 2011. Characterizing failure-causing parameter interactions by adaptive testing. In Proceeding of ACM International Symposium on Software Testing and Analysis (ISSTA 2011), pages 331-341, 2011.

[10] Chen, T. Y., Kuo, F.-C., Merkel, R. G.and Tse, T. H. 2010. Adaptive random testing: The art of test case diversity. J. Syst. Softw. 83, 1, 60–66.

[11] Yuan, X., Coheb, M. B. and Memon, A. M. 2009. Towards Dynamic Adaptive Automated Test Generation for Graphical User Interfaces. IEEE International Conference on Software Testing Verification and Validation Workshops:263-266

[12] Ghandehari, L., Lei, Y. and et al. 2012. Identifying Failure-inducing Combinations in a Combinatorial Test Set. In Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST 2012).