Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 14, no. 6, June 2018, pp. 1300-1307
DOI: 10.23940/ijpe.18.06.p21.13001307

Pinpoint Minimal Failure-Inducing Mode using Itemset Mining under Constraints

Yong Wang[a], Liangfen Wei[b,*], Yuan Yao[a], Zhiqiu Huang[c], Yong Li[c], Bingwu Fang[c], and Weiwei Li[c]

[a]*School of Computer and Information, Anhui Polytechnic University, Wuhu, 241000, China*
[b]*Department of Computer Engineering, Anhui Sanlian University, Heifei, 230601, China*
[c]*College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, 210016, China*

Abstract

A minimal failure-inducing mode (*MFM*) based on a t-way combinatorial test set and its test results can help programmers identify root causes of failures that are triggered by combination bugs. However, an *MFM* for systems containing many parameters may be affected by masking effects to result in coincidences correct in practice, which makes pinpointing *MFS* more difficult. An approach for pinpointing *MFM* and an iterative framework are proposed. The identifying *MFM* approach first collects combinatorial test cases and their testing results, then mines the frequent itemset (suspicious *MFM*) in failed test cases, and finally computes suspiciousness for each *MFM* belonged to *close pattern* via contrasting frequency in failed test cases and successful test cases. Through the iterative framework, *MFM* is pinpointed until a certain stopping criterion is satisfied. Preliminary results of simulation experiments show that this approach is effective.

Keywords: combinatorial testing; minimal failure-inducing mode; itemset mining; program debugging

(Submitted on March 21, 2018; Revised on April 27, 2018; Accepted on May 28, 2018)

1. Introduction

Combination testing, which samples software input domain to trigger software failure caused by combinatorial fault, has been proven to be a very efficient test technique in practice. The main issue is to construct a minimal test suite to satisfy certain test coverage criteria. However, combinatorial testing only determines the combination parameter values that cause program failure rather than locates the root cause of the failure. If we can pinpoint the minimal failure-inducing model that triggers failures, we can use this information to aid the debugging process.

In recent years, how to use test suite and their testing results to diagnose the root cause of failures has attracted much attention [1,2,8,10]. Nie et al. [10] proposed the minimum failure-causing schema, which also is called the minimal failure-inducing mode, and highlighted the *MFS* to help locate and understand the bug(s) in source code [8]. To pinpoint *MFM*, Xu et al. [12] proposed a one-by-one replacement technique via heuristic methods to generate new failed test case to narrow down the possible search domain for *MFM*. Zhang et al. [13] proposed a fault interaction characteristic (FIC) positioning method. They used the failed test case as a seed to generate an additional test case set, adaptively modified the additional test case set, performed an enhanced testing for the under test software, and repeatedly iterated until the minimal failure-inducing mode was found. Ghandehari et al. [4] proposed an MFM localization technique based on combination suspiciousness and context suspiciousness. Niu et al. [11] constructed a specific tuple relationship tree (*TRT*) to describe the relationship of all the specific value combinations. Using *TRT* can effectively reduce the number of additional test cases generated and provide accurate *MFM*.

* Corresponding author.
E-mail address: yongwang@ahpu.edu.cn

However, the above methods assume that if a combination parameter value causes the failure of the software, the testcases containing the combination must trigger the failure necessarily. However, the assumption cannot be held in practice. Dumlu et al. [3] first pointed out that in the real system, the masking effects will cause the test cases containing the *MFM* to be successfully executed only occasionally. Therefore, in the real system, even if the *MFM* is covered, the software failure is intermittent. Although there are existing masking effects, the *MFM* must exist in the failed test cases and seldom exist in successful test cases according to the software RIPR model [9]. Therefore, we can use data mining techniques [7] to pinpoint the *MFM*. In this paper, we propose a method to pinpoint *MFM* via itemset mining under constraints. We first collect many test cases and their results based on combinatorial testing process, and then a suspicious *MFM* ranking list is generated to satisfy some properties of the whole set of patterns via itemset mining under constraint.

1. Preliminaries

1.1. Basic concepts

In this section, we give some definitions that are used in our approach.

We assume the software under testing has *n* input parameters/configures, the input domain can be denoted by $V = \{V_1, V_2, ..., V_n\}$.

Definition 1. *Parameter*, A parameter V_i is a set of discrete values. Let d_i be the domain of parameter V_i. Then, the parameter is $V_i = \{d_1, d_2, ..., d_k\}$.

Definition 2. Test case, A test case is a map that assigns a parameter value to each parameter.

Let *T* be all possible test cases for a under testing software. It is clear that $|T| = |d_1| \times |d_2| \times ... \times |d_k|$.

Table 1. Failed test cases and frequent itemsets for the example

Client browser	OS for Server	OS for Client	Database
IE(1) Netscape(2)	Client OS(1) Unix(2) OS/360(3)	Windows(1) MacOS(2) -	MySQL(1) Oracle(2) DB2(3)

Definition 3. *k-value mode.* For a test case T_i, a tuple [-, ..., b_{i1}, ..., b_{ik}, ...], where v_{ij} set a fixed value, '–' means that the corresponding parameter can take any parameter value, is called *k-value* mode ($n \geq k > 0$).

As *k=n*, the *k-value mode* is a test case. For example, [-,*WinXp*, -, *NewUser*] is a *2-value mode.*

Definition 4. *k-value MFM. A k-value MFM* is a minimal *k-value mode*, and it is the root cause of failures.

Generally, we assume that a program executed with a test case containing a *k-value mode* must trigger a software failure. However, if a software is affected by *Masking Effects*, a test case contained a *k-value MFM* is executed, and the program will accidentally execute successfully.

Definition 5. *Masking Effects* refer to a certain *k-valued MFM* that was accidentally executed successfully by its execution environment.

Dumlu et al. [3] pointed out that *Masking Effects* may be the reason that, due to affects by other parameter values, the software returned early in procedure of the execution or unexpectedly caused the program to run without reaching the interface location for the *MFM*.

Definition 6. *Invalid test case. An invalid test case* is an infeasible test due to some constraints for input parameter values. In combinatorial testing, we automatically generate test suites based on certain test criteria. Considering the constraints, the test suites contain some invalid test cases. In practice, we should filter those invalid test cases to perform testing.

1.2. Assumptions

Assumption 1. The software under testing is deterministic. Namely, the software always generates the same output when executed with given input values.

Assumption 2. There exists a test oracle to generate the result of a test execution, i.e., 'successful' or 'failed.'

Assumption 3. Failure-inducing mode should involve no more than *t* parameters, where *t* is the strength of the initial combinatorial testing.

2. Frequent itemset mining under constraint

This section gives a brief overview of frequent itemset mining in data mining community [6].

Let Y be a set of different literals (namely items), and Γ = {1, …, m} is a set of transaction identifiers. An itemset (pattern, or mode) is a *nonNull* subset of Y. The language of patterns corresponds to $L_\gamma = 2^\gamma / \varphi$. A transaction dataset is $D \subseteq \Upsilon \times \Gamma$.

Definition 7. *Frequency.* Let a coverage of a pattern *p* be the identifier in which *p* occurs, namely, $\mathrm{cov}erage_D(P) = \{t \in \Gamma \mid \forall i \in p, (i,t) \in D\}$.

The frequency of a pattern p is the number of its coverage, namely $fr_D(p) = |coverage_D(p)|$.

Pattern-mining under constraint focuses on generating all patterns *p* for L_Y to meet with a query *q*(*p*). Generally, we define this task of the query as $Th(q) = \{p \in L_\gamma \mid q(p) = True\}$. A common constraint is the minimal frequent support, which can be defined as min_{fr} : $frequency_D(p) \geq min_{fr}$.

In our application for pinpointing *MFM*, it appears highly appropriate to find contrasts between subsets of transactions, such as successful and failed test cases in fault localization community. The growth rate is a well-used contrast measurement. Let *D* be a dataset partitioned into two subsets D_1 and D_2:

Definition 8. *Growth Rate* The growth rate of a pattern *p* from D_1 to D_2 is

$$m_{gr}(p) = \frac{|D_2| \times frequency_{D1}(p)}{|D_1| \times frequency_{D2}(p)} \tag{1}$$

3. Our Approach

3.1. Framework

This approach focuses on pinpointing *MFM* based on itemset mining. Therefore, there should exist many test cases for combination testing. Considering there exists an imbalance between failed test suites and successful test suites, we need to add the number of failed test cases. The main steps are the following:

- *Step 1:* Collect the basic test suite and additional test suite.
- *Step 2:* Generate the frequent item based on the FP-growth algorithm.
- *Step 3:* Rank frequent items by contrasting failed test cases to successful test cases.

Considering the cost of additional testing, the three steps can be performed iteratively when we do not satisfy the results. The programmer can decide whether to stop or not at the end of each iteration according to their limited resource.

Algorithm 1 : Our Approach Framework

Input: Software under testing *p*, Combination test suite Γ, Frequent threshold α and Suspiciousness Metric ρ_b

Output: Minimal failure-inducing Mode Ω;

1: //*step 1* Collect additional test suite

2: $(e) \leftarrow run\_Program(P, \Gamma)$;

3: $(\Gamma^+, \Gamma^-), \leftarrow divide\_T(\Gamma, e)$

4: $(\Gamma_a^+, \Gamma_a^-), \leftarrow generate\_AdditionT(\Gamma^-)$

5: //Generate frequent item based on FP-growth algorithm

6: $(\Gamma^{+}_{item}, \Gamma^{-}_{item}), \leftarrow generate\_\Gamma item(\Gamma^{+}, \Gamma^{-}, \Gamma^{+}_{a}, \Gamma^{-}_{a})$

7: $(MFM_s, freq^{-}(MFM_s)) \leftarrow FP-growth(\Gamma^{-}_{item}, \alpha)$

8: //Step 3:Ranking MFM

9: $MFM_s \leftarrow close\_Pattern(MFM_s)$

10: $freq^{+}(MFM_s) \leftarrow compute\_\ freq^{+}(MFMs, \Gamma^{+}_{item})$

11: $S \leftarrow rho_b(MFM_s, freq^{+}, freq^{-})$

12: $\Omega \leftarrow sort(S)$;

13: return Ω;

3.2. Additional test cases Generation

The cost of the problem of *MFM* localization mainly is effort of testing. Therefore, on one hand, we hope to reduce the number of tests for software testing. On the other hand, we need to perform additional testing for *MFM* localization. To balance the cost of testing, we should localize *MFM* using as few test cases as possible. In software testing, a *basic choice coverage, BCC* is defined in Definition 9.

Definition 9. *Basic choice coverage (BCC)*: A base choice block is chosen for each parameter value, and a base test is formed by using the base choice for each parameter value. Subsequent test cases are chosen by holding all but one base choice and changed using its non-based choice value.

For example, there exists three partitions with blocks [1,2], [1,2,3], [1,2] for three parameters. Supposing base choice block are '1', '3', '1', the base choice test is [1,3,1], and the following additional tests would be generated: [2,3,1], [1,1,1], [1,2,1], [1,3,2].

The basic choice test cases can often be viewed as an important test case. In our approach, we viewed each failed test case as a basic choice test case. We generate additional test cases via *BCC*. Obviously, those additional test cases are more likely to be a failed test case because those test cases are more similar to failed test cases. Due to our goal, we get one failed test case in additional testing each time, add the test case to the dataset, and run our approach. If we cannot get a good result, the process of additional testing is performed iteratively.

3.3. MFM Mining

When we get test cases and their testing results, we first convert test cases into transaction dataset, which is suitable for frequent itemset mining. We use an example, which is shown in Table 2, to illustrate our idea. In the example, a test case such as $t_1 = [2, 1, 2, 1]$ cannot be viewed directly as a transaction database. We modify this test as a transaction '*a*2', '*b*1', '*c*2', '*d*1', which can be viewed as a transaction named Γ_{item}. As shown in Table 3, we give failed test cases and frequent items for the example.

Table 2. Test cases and their results for the example

T_{id}	a	b	c	d	Test Result
1	2	1	2	1	successful
2	1	1	1	2	successful
3	2	1	1	3	successful
4	1	2	2	1	failed
5	2	2	1	2	successful
6	1	2	2	3	failed
7	2	3	1	1	successful
8	1	3	2	2	successful
9	2	3	2	3	successful
10	2	2	2	2	failed
11	1	1	2	1	successful
12	1	2	1	1	successful
13	1	2	2	2	successful
14	2	2	2	3	failed
15	1	1	2	3	successful
16	1	2	1	3	successful

To contrast between subsets of transactions, Γ is divided into disjoint subsets Γ^+ and Γ^-, where Γ^+ denotes the set of successful test cases and Γ^- denotes the set of failed test cases.

Table 3. Failed test cases and frequent itemsets for the example

T_{id}	failed testcases	(ordered)Frequent Items
1	a=1,b=2,c=2,d=1	b2,c2,a1
2	a=1,b=2,c=2,d=3	b2,c2,a1,a3
3	a=2,b=2,c=2,d=2	b2,c2,a2
4	a=2,b=2,c=2,d=3	b2,c2,a2,d3

In the context of *MFM*, there exist masking effects. *MFM* would be in failed cases, and few appear in successful tests cases. Therefore, we divide two subsets, Γ^+_{item} and Γ^+_{item}, for test cases based on their testing results. Based on Γ^+_{item}, we mine frequent itemset given a frequent threshold value α. We use the *FP-growth* algorithm to mine frequent itemset in failed test transactions. *FP-growth* was proposed by Han et al. [7]. The algorithm counts occurrence of items (attribute-value pairs) in the dataset and stores them to a 'header table' in the first pass. In the second pass, it builds the *FP-tree* structure by inserting instances. Items in each instance have been sorted by descending order of their frequency in the dataset, so that the tree can be processed quickly. Items in each instance that do not meet minimum coverage threshold are discarded. If many instances share most frequent items, *FP-tree* provides high compression close to tree root.

3.4. MFM Ranking

The set of *frequent itemset* contains redundancy about *frequency*. Given a frequency $freq$, two mode m_i and m_j are said to be equivalent if $freq(m_i) = freq(m_i)$.

A set of equivalent modes forms an equivalent class about *frequency*. The largest mode about the set inclusion of an equivalence class is called a *closepattern*.

Definition 10. *Closed pattern* A pattern $p_i \in L_\gamma$ is closed pattern w.r.t, a measure m $\forall p_i \in L_\gamma$, $p_i \not\subset p_j$ $\Rightarrow m(p_j) \neq m(p_i)$.

For instance, [a1, b2, c3]:3 is a closed pattern for [a1, b2]:3, [a1, c3]:3, [b2, c3]:5, [a1]:3, [b2]:3, [c3]:3.

In the spectrum-based fault localization community, there exist many suspiciousness measurements. A suspiciousness measurement called *Tarantula* is defined as Formula 2.

Definition 11. *Tarantula Measurement* is defined as Formula 2, which can compute the suspiciousness of a program entity e.

$$\rho_{\text{Tar}(e)} = \frac{\frac{failed(e)}{totalfailed}}{\frac{passed(e)}{totalpassed} + \frac{failed(e)}{totalfailed}} \tag{2}$$

Formula 2 is very similar to Formula 1, which measures the occurrence of a given pattern from a dataset to another one. We modify Formula 2 to compute the frequency of suspicious *MFMs* in Γ^+_{item} and Γ^-_{item}. The new suspiciousness measurement is defined as Formula 3. Finally, we sort those suspicious *MFMs* according to the suspiciousness measurement.

$$\rho_{MFM} = \frac{\frac{freq^-(MFM)}{totalfailed}}{\frac{freq-(MFM)}{totalpassed} + \frac{freq-(MFM)}{totalfailed}} \tag{3}$$

4. Experiment and Results

We conducted simulation experiments to verify the effectiveness of our approach. Due to our iterative framework, the result of our approach is a ranking list. To facilitate comparison with other methods, we use top-1 as the result of our approach based on iterative framework. Overall, the simulation experiment mainly answers the following questions:

- Question 1: Whether our approach is effective or not for systems without Masking Effects;
- Question 2: Whether our approach is effective or not for systems in the presence of Masking Effects.

4.1. Whether our approach is effective or not for systems without Masking Effects

For research question 1, we assume that there are three systems under test, containing 8, 10, and 12 parameters respectively. There exist 3 parameter values for each parameter. The test cases were generated using the *IPOG* algorithm recommended by the combination testing tool *ACTS* [5]. For example, for the system under test containing 8 parameters, the number of test cases for 2-way is 15, the number of test cases for 3-way is 60, and the number of test cases for 4-way is 191.

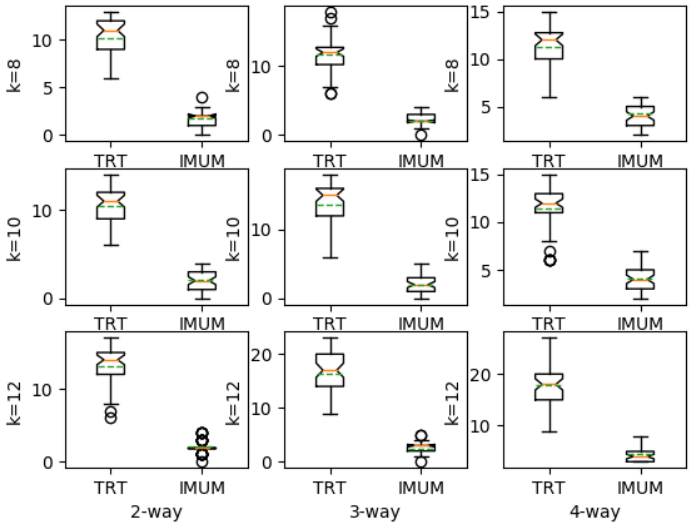

Figure 1. Effectiveness Comparison with different setting for systems without masking effects

In order to evaluate the effectiveness of our approach for the systems without masking effects, we use the tuple relationship tree model (*TRT*) proposed in [11] as a benchmark. Niu et al. [11] propose a relational tree model for identified *MFM*. Specifically, the method first constructs a relational tree to record all the tuples to be measured and the relationship between them, and then pinpoints *MFM* based on the tuple relationship. The *TRT* method gives 4 test case selection strategies, and we use its optimal test case selection strategy (abbreviated path method) as a comparison standard. Unlike our approach, the *TRT* approach generates an accurate *MFM*. According to the definition of *MFM*, it needs to determine that all the sub-tuples of *MFM* are correct tuples. This method is obviously not applicable for systems with masking effects. For systems without masking effects, a test case containing *MFM* fails.

We simulate the process of combination testing and set the result for a testing according to whether it contains an *MFM* or not. Then, we divide test cases into two sets: successful set and failed set. We use the iterative framework mentioned above to get the number of additional testing. For the three systems with different parameters, we use different *MFMs* to repeat 100 times and record the additional testing number. The results are shown in Figure 1. In the Figure, our approach is labeled as *IMUM*, and the *TRT* approach is labeled as *TRT*. Obviously, our approach is superior to the *TRT* method. It is worth noting that the *TRT* method uses an accurate approach and needs to check that all sub-modes of the *MFM* are correct tuples. This process would increase additional testing number. According to [11], the *TRT* approach is effective for systems without masking effects. However, for systems with masking effects, this approach would hardly achieve ideal results.

4.2. Whether our approach is effective or not for systems in the presence of Masking Effects

For this research question, we mainly focus on the effectiveness of our approach for systems with masking effects. Considering different probabilities of masking effects, we assume that the software under testing contains 10 parameters. This system has existing masking effects, and the test cases containing the *MFM* are randomly executed correctly with a percentage of 10% and 30%. In the same setting, we compare the additional testing number between systems with masking effects and without masking effects.

We also use the *ACTS* tool to generate 2-way, 3-way, and 4-way test cases, and we simulate *t-MFM* with a single t-value in the presence of masking effects. Suppose you set masking effects too, that is, there is a 30% probability that the test case containing the *MFM* is executed successfully. For each test case that contains an *MFM*, we call a random function that sets the test case as a successful test case with a probability of 30%.

Similarly, we use the iterative framework of our approach to get the additional testing number. For different systems that contain different parameters, different *MFMs* (contained 2-way, 3-way, and 4-way *MFM*) were injected. We used different *MFMs* to repeat 100 times and recorded the additional testing number for each time.

The results are shown in Figure 2. In Figure 2, the y-axis index is the additional testing number. It is easy to see that with the increased probability of masking effects, the additional test number is increased significantly. In the experimental process, we found that for systems with existing masking effects, their results are highly dependent on the number of failed test cases. Therefore, our approach performs better in a system containing more failed test cases.

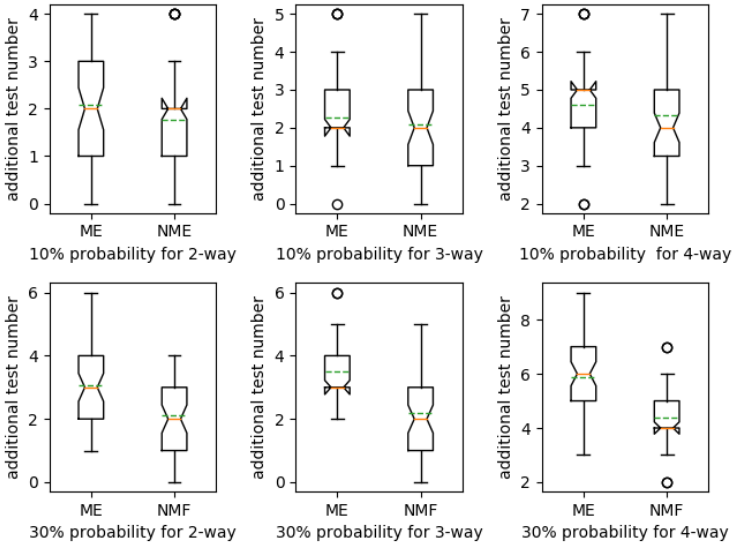

Figure 2. Effectiveness Comparison with different setting for systems contained masking effects.

5. Conclusions

This paper proposes an approach of pinpointing the minimum failure-inducing mode based on itemset mining under constraint. This approach first collects test cases, addition test cases and their testing results, then mines the frequent itemset (suspicious *MFM*) in failed test cases, and finally computes suspiciousness for each close pattern via contrasting frequency in failed test cases and successful test cases and sorts them. Preliminary results of simulation experiments show that this method is effective. In particular, this method has good robustness for systems with masking effects. However, only *single-MFM* scenarios are considered in our simulation experiments. In future works, it is necessary to further consider *multi-MFM* scenarios. Additionally, we would also consider further validating the effectiveness of our approach in actual systems.

Acknowledgements

This work was supported in part by the National Key Research and Development Program (Grant No. 2016YF-B1000802) of China, the National Natural Science Foundation (Grant No.61772270,61562087) of China, the Anhui University Natural

Science Foundation Key Project (Grant No. KJ2016A252, KJ2018A0116), and the Natural Science Foundation of Jiangsu Province (Grant No. BK20170809).

References

1. J. Chandrasekaran, H. Feng, Y. Lei, et al. Applying combinatorial testing to data mining algorithms[C]//Software Testing, Verification and Validation Workshops (ICSTW), 2017 IEEE International Conference on. IEEE, 2017: 253-261.
2. J. Chandrasekaran, L S. Ghandehari, Y. Lei, et al. Evaluating the Effectiveness of BEN in Localizing Different Types of Software Fault[C]//Software Testing, Verification and Validation Workshops(ICSTW), 2016 IEEE Ninth International Conference on. IEEE, 2016: 26-34
3. E. Dumlu, C. Yilmaz, M. B. Cohen, et al. Feedback driven adaptive combinatorial testing[C]//Proceedings of the 2011 International Symposium on Software Testing and Analysis. ACM, 2011: 243-253.
4. L S G. Ghandehari, Y. Lei , T Xie, et al. Identifying failure-inducing combinations in a combinatorial test set[C]//Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on. IEEE, 2012: 370-379.
5. L. S. Ghandehari, J. Chandrasekaran, Y. Lei, et al. BEN: A combinatorial testing-based fault localization tool[C]. Proceeding of 2015 IEEE Eighth International Conference on Software Testing Verification and Validation Workshops (ICSTW), 20151-4.
6. T. Guns, S. Nijssen,L. De Raedt. Itemset mining. a constraint programming perspective[J]. Artif Intell.175(12), 1951C1983
7. J. Han, J. Pei, M. Kamber. Data mining: concepts and techniques[M]. Elsevier, 2011.
8. N. Li, Y Lei, H R. Khan, et al. Applying combinatorial test data generation to big data applications[C]//Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on. IEEE,2016: 637-647.
9. N. Li, J. Offutt. Test oracle strategies for model-based testing[J]. IEEE Transactions on Software Engineering, 2017, 43(4): 372-395.
10. C. Nie, H. Leung. The minimal failure-causing schema of combinatorial testing[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2011, 20(4): 15.
11. X. Niu, C. Nie, Y. Lei, et al. Identifying failure-inducing combinations using tuple relationship[C]//Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on. IEEE, 2013: 271-280.
12. B. W. Xu, C. H. Nie, L. Shi, et al. A software failure debugging method based on combinatorial design approach for testing[J]. CHINESE JOURNAL OF COMPUTERS-CHINESE EDITION, 2006, 29(1): 132((in Chinese)).
13. Z. Zhang, J. Zhang. Characterizing failure-causing parameter interactions by adaptive testing[C]//Proceedings of the 2011 International Symposium on Software Testing and Analysis. ACM, 2011: 331-341.