Generating Combinatorial Test Suite for Interaction Relationship

Wang Ziyuan Nie Changhai Xu Baowen
School of Computer Science and Engineering, Southeast University, Nanjing, 210096, China
Jiangsu Institute of Software Quality, Nanjing, 210096, China
{wangziyuan, changhainie, bwxu}@seu.edu.cn

ABSTRACT
Combinatorial testing could detect the faults triggered by the interactions among factors in software. But in many cases, the pair-wise, *N*-way and even the variable strength combinatorial testing may lead test suite redundancy and fault detect ability decreasing, because these methods do not make sufficient consideration on the actual factors interaction. In this paper, a new interaction relationship based combinatorial testing model was proposed to cover the actual factor interactions in software by extending the conventional combinatorial testing model and IO relationship testing model. The new method may be more effectively than existed combinatorial testing methods without decrease of the fault detect ability. Furthermore, two test suite generation algorithms for interaction relationship based combinatorial testing were also presented. Finally, we compared our algorithms with some similar test generation algorithms in IO relationship testing model, and the experience result showed the advantage of our algorithms.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging –*Testing tools*

General Terms
Algorithms, Experimentation.

Keywords
Software testing, combinatorial testing, interaction relationship, test generation

1. INTRODUCTION
As a complex logic system, software may be affected by many factors, such as system configurations, internal events, external inputs etc. The software faults may be triggered by not only some single factors but also the interactions of these factors. So as an effective testing approach, combinatorial testing, which can detect the faults triggered by these factors and their interactions, was widely used in practical.

Many works have been done in the field of combinatorial test generation. For example, some heuristic methods to generate combinatorial test suite were proposed by Cohen D M et al [1], Lei Y et al [2], Tung Y W et al [3], and Cohen M B et al [4] respectively. Some algebraic test generation approaches were proposed by Kobayashi N and Tsuchiya T [5], William A W [6]. And there are also some search-based methods such as simulated annealing algorithm [7], generic algorithm and ant colony algorithm [8] being proposed. Schroeder P J et al compared the fault detection effectiveness of N-way combinatorial testing and random testing [9]. In 2006, Zhang J et al proposed a backtrack-based algorithm for combinatorial test generation [10] and Yilmaz C et al research the advantage of combinatorial testing in size of test suite and fault detection effectiveness [11].

All works on combinatorial testing, including pair-wise, *N*-way ($N \geq 2$) and even variable strength combinatorial testing, are on the assumption that there are pair-wise or *N*-way interaction among any two or *N* factors. But as there are not always interactions among any *N* factors in software, the test suite used by conversional combinatorial testing may still have redundancy and its fault detection ability could be also strengthened farther more. So if we can make use of more information about actual factors interaction relationship in software to design test suite, the size of test suite may be reduced while the fault detection effectiveness may be increased. Some interaction relationships in software, such as input-output (IO) relationships, have been discussed by Schroeder P J et al detailedly. And some corresponding test generation algorithms to cover the input-output relationships were also given [12][13] [14][15].

In this paper, we extend the model of conventional combinatorial testing and generalize the model of IO relationship based testing, to propose a new combinatorial testing model that interaction relationship based combinatorial testing, as the supplement of pair-wise, *N*-way and variable strength combinatorial testing. Rather than, we also research the method to generate combinatorial test suite to cover the interaction relationship, and present a serial of new combinatorial test generation algorithms.

The remainder of this paper is organized as follows. Section 2 describes the basic model and definitions. Section 3 reviews related works and analyze the existed problems. Two test suite generation algorithms are presented separately in section 4 and section 5. Section 6 compares our test generation algorithms with others. Finally, make a conclusion and point out the future works.

SOQUA'07, September 3-4, 2007, Dubrovnik, Croatia

2. DEFINITIONS

Suppose that the software system under test (SUT) may be affected by n factors (or parameters) and each factor f_i has $a[i]$ ($1 \le i \le n$) discrete valid values. Let $F=\{f_1, f_2, \ldots, f_n\}$ denote the set of factors, and $V_i=\{1, 2, \ldots, a[i]\}$ ($1 \le i \le n$) denotes the value set of f_i without loss of generality. In this paper, we also suppose that all factors are independent, which means that there are not constraints between factors and their values.

Definition 1 The n-tuple $test= (v_1, v_2, \ldots, v_n)$ ($v_1 \in V_1, v_2 \in V_2, \ldots, v_n \in V_n$) is a test case for the SUT.

Next, we discuss the interactions among factors. As the factor of software may interact with each other, a group of factors that have interaction with each other can be formed as a subset r of F. That means there is a $|r|$-way interaction among the factors in this subset, where $|r|$ is the cardinality of r. So for any subset such as this r, the combinatorial test suite is needed to cover all valid value combinations of factors in it. Similar as before, if we abstract each one of all t interactions in software as a subset of F, then there is a collection of these subsets $R=\{r_1, r_2, \ldots, r_t\}$. And a combinatorial test suite for this software should cover all t interactions that associate with subsets in R. For example, for a given software with n factors, a pair-wise combinatorial test suite should cover $|R|=n\times(n-1)/2$ different 2-way (or pair-wise) interactions and $R=\{\{f_i, f_j\} | f_i, f_j \in F$ and $i \ne j\}$, and a N-way combinatorial test suite cover $|R|=C^N_n$ different N-way interactions and $R=\{\{f_{i1}, f_{i2}, \ldots, f_{iN}\} | f_{i1}, f_{i2}, \ldots, f_{iN} \in F\}$.

Definition 2 The subset $r_k \in R$ ($k=1, 2, \ldots, t$) is named as an interaction coverage requirement, and the collection R is the interaction relationship of SUT.

For simplicity, we support that: (i) each coverage requirement $r_k=\{f_{k,1}, f_{k,2}, \ldots, f_{k,nk}\} \in R$ ($k=1, 2, \ldots, t$) have n_k factors where $n_k>1$; (ii) for any two requirements $r_{k1}, r_{k2} \in R$ ($k_1 \ne k_2$), r_{k1} is not subset of r_{k2} and r_{k2} is not subset of r_{k1}; (iii) two factors $f_i, f_j \in F$ ($i \ne j$) interact with each other if and only if there exist a coverage requirement $r \in R$ that $f_i, f_j \in r$.

For example, consider an object-oriented system with 4 class clusters (factors) $F=\{A, B, C, D\}$ and each class cluster has 2 concrete classes (values), which is shown as class diagram in Figure 1. The interaction relationship of this system can be described as $R=\{r_1, r_2, r_3\}$ where $r_1=\{A, B, C\}$, $r_2=\{A, D\}$ and $r_3=\{C, D\}$. That is class A, B and C determine the Client jointly and class D associates with class A and C respectively.

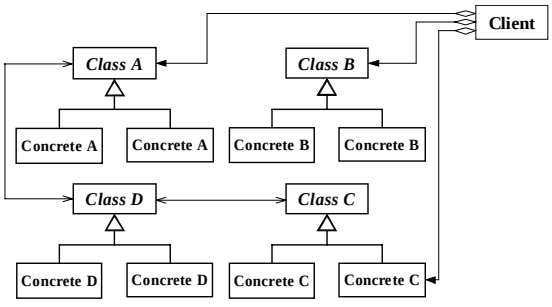

Figure 1. Class diagram

Definition 3 Given $A=(a_{i,j})_{m\times n}$ is a $m\times n$ array, in which the j-th column denotes the factor f_j of the SUT and all the elements of this column come from the finite set V_j ($j=1, 2, \ldots, n$), that is $a_{i,j} \in V_j$. If there is a $m\times n_k$ sub-array A_k, which composed of the columns of A that corresponding to the factors in interaction coverage requirement $r_k \in R$, contains all n_k–way values combinations of factors in r_k, then we say that A covers the interaction coverage requirement r_k. If A satisfies all coverage requirements in R, then we say that A covers the interaction relationship R, and A is a covering array that covers R.

For a given SUT, the combinatorial test suite T that covers interaction relationship R could be obtained easily from the covering array A that covers R. So we suppose that combinatorial test suite and covering array are equivalent in this paper.

Definition 4 Let T is combinatorial test suite that covers interaction relationship R of SUT, if it contains minimum possible number of test cases, then T is the optimal combinatorial test suite that covers R, or the optimal test suite.

Differ from the conventional combinatorial testing method, interaction relationship based combinatorial testing do not need to generate test suite to cover all N-way interactions but only need to cover all actual interactions. In this paper, we assume that the actual interaction relationship has been obtained, which could be obtained from many ways, such as reviewing design documents, source code analysis, or interviewing with programmers etc.

3. RELATED WORKS

Combinatorial testing, especially pair-wise testing, have been widely used in practical successfully. But Bach J and Schroeder P J pointed out that all such successful cases were on the ground that analyzing the characteristic of software in detail and obtaining the factors interaction relationship sufficiently [16].

However, in traditional model of combinatorial testing, there is a assumption that there are interactions among any N($N \ge 2$) factors at least and generate N-way test suite or variable strength combinatorial test suite on this hypothesis. But in reality, there are few software systems have this property. So if we do not consider the character of SUT sufficiently when generating test suite, the effectiveness of testing may be reduced: (i) if we choose N as a large number, the test suite may be redundant; (ii) there are may be some interactions influence more than N factors in a software, that means some valid combinations may not be covered by N-way test suite and some bugs may not be detected. Above all, to increase the effectiveness of combinatorial testing, it is necessary to mine the actual interaction relationship among factors in software before test generation.

To solve the problem of test generation for software with complex input-output relationship, Schroeder P J et al proposed the model of input-output relationship based testing method and give three different test generation algorithms [12][13][14]. The first one is a brute force approach to find a minimum test suite. The second one, the Union algorithm, designs a serial of test suite for each output variable to cover the interaction among the associative inputs variables respectively, and then takes the union of them to get a final test suite. This algorithm is very simple that the time complexity is only $O(\sum^t_{k=1}\prod_{fi \in rk} a[i])$ under the model that defined in this section 2, but can not generate small test suite. And the last

one, Greedy algorithm, selects an unused test case that covers the greatest number of uncovered combinations of input values each times until all interactions have been covered by the selected test suite. It may generate a much smaller test suite than the second one, but with a bad time and space performance for it must check all test cases in a huge search space. And its time complex is as much as $O(|T|\times(\prod_{i=1}^{n} a[i])\times(\sum_{k=1}^{t}\prod_{fi\in rk} a[i]))$ in our model. So in 2003, they proposed a problem reduction method, which is based on the color graph, to make the Greedy algorithm become more efficient [15]. But as the authors mentioned, this reduction method is suitable only when relationships are "simple" that the number of edges in associated color graph is much smaller than that of complete graph with the same number of edges. And the algorithm with reduced technique may generate some redundancy test cases.

In recent years, we have made much study on combinatorial testing. And the characteristic of factors interaction relationship has also been considered in our works. For a special case that the interaction exists only among neighbor factors, the neighbor factors combinatorial testing method was proposed and a serial of optimal neighbor factor combinatorial test suite generation algorithms for different cases were presented [17]. It is evident that the neighbor factors *N*-way testing is a special case of interaction relationship based combinatorial testing when the interaction relationship $R=\{\{f_i, f_{i+1},\ldots, f_{i+N-1}\}|f_i, f_{i+1},\ldots, f_{i+N-1}\in F\}$. And its effectiveness has been shown in the application of testing for railway signaling systems.

In this paper, we extend the model of input-output relationship based testing to the interaction relationship based combinatorial testing, which is a little more general than the former. Rather than, the more important works of this paper are two new test generation algorithms for interaction relationship based combinatorial testing with higher performance.

4. GENERATING TEST SUITE IN COVERAGE REQUIREMENT ORDER

By analyzing the disadvantage of Union algorithm proposed by Schroeder P J, a better combinatorial test generation algorithm was given in this section.

4.1 Analyzing Union Algorithm

The Union algorithm may not create a small test suite because the size of generated test suite depends on how the values are assigned to "don't care" factors, which are the factors that do not belong to current coverage requirement. For example, considering a SUT with $F=\{2^4\}$, that is there are 4 factors and each factor has 2 discrete possible values. For two covergae requirements $r_1=\{f_1, f_2\}$ and $r_2=\{f_3, f_4\}$ in *R*, design two test suite $T_1=\{(1, 1, 1, 1), (1, 2, 1, 1), (2, 1, 1, 1), (2, 2, 1, 1)\}$ and $T_2=\{(1, 1, 1, 1), (1, 1, 1, 2), (1, 1, 2, 1), (1, 1, 2, 2)\}$ to cover r_1 and r_2 respectively, where all "don't-care" factors are assigned as 1. And then take the union of them to get $T= T_1\cup T_2=\{(1, 1, 1, 1), (1, 2, 1, 1), (2, 1, 1, 1), (2, 2, 1, 1), (1, 1, 1, 2), (1, 1, 2, 1), (1, 1, 2, 2)\}$ with size 7. However, there is a test suite $T'=\{(1, 1, 1, 1), (1, 2, 1, 2), (2, 1, 2, 1), (2, 2, 2, 2)\}$ with size 4 can also cover r_1 and r_2. We can conclude from above example that, if there is not a better strategy to determine the values of "don't-care" factors when generating test cases for a given requirement, the final test suite may be redundant.

Hence, we propose a new test generation algorithm to avoiding the disadvantage of Union algorithm. In our algorithm, all the positions that corresponding to each "don't-care" factor will not be assigned until a coverage requirement which include this factor is dealt. This strategy may avoid the redundancy of combinations in "don't-care" factors and reduce the size of test suite evidently.

4.2 Test Generation Algorithm

For a given interaction relationship $R=\{r_1, r_2,\ldots, r_t\}$, the test generation algorithm in coverage requirement order will selects a coverage requirement from *R* and generates an initialize test suite for it firstly. Then extend this test suite to satisfy another selected requirement. This step will be repeatedly until all requirements in *R* have been selected out and satisfied by the final test suite. The detail of this algorithm is shown as follows:

Algorithm 1. Test Generation Algorithm in Coverage Requirement Order (ReqOrder)

```
Input: F={f1, f2,..., fn}, R={r1, r2,...,rt}
Output: Test suite T that cover R
begin
  r=SelectReq(R);                    //select a requirement r∈R
  Create a test suite T with ∏fi∈r a[i] empty test cases;
  Fill all combinations of factors in r into T;
  R = R-{r};
  Fdeal={fi|fi∈r1};                  //factors have been assigned
  while R ≠ ∅
    r=SelectReq(R);
    if rk∩Fdeal=∅ then
      if ∏fi∈r a[i]>|T| then
        Add ∏fi∈r a[i]-|T| empty test cases into T;
      Fill all combinations of factors in r into T;
    else
      //computing the needed appearance time of
      //combinations of factors in r∩Fdeal
      need=∏fi∈r−Fdeal a[i];
      for each combination com of factors in r∩Fdeal
        Let count is appearance times of com in T;
        if count<need then
          for j = count to need-1
            if there are test cases in which com could be added
            into then                //scan all test cases
              Select a test case that match com mostly from
              above test cases;
              Add com into selected test case;
            else
              Add need-j empty test cases into T;
              Fill com into these need-p test cases;
              break;
        if r- Fdeal ≠ ∅ then
          Fill need different combinations of factors in r-Fdeal
          into test cases that contain com;
    R = R-{r}; Fdeal= Fdeal ∪r;
end
```

(i) Selects a coverage requirement from *R* by a given strategy. Without loss of generality, let it be r_1. Construct a test suite *T* with size $m_1=\prod_{fi\in r1} a[i]$ and fill all m_1 combinations of factors in r_1 into it. As we mentioned before, all positions corresponding to "don't-care" factors will be empty.

(ii) Selects another coverage requirement $r \in R$ that have not been dealt before. Without loss of generality, we assume that the k-th requirement that will be dealt is $r_k=\{f_{k,1}, f_{k,2}, \ldots, f_{k,nk}\}$ (k=1, 2, …, n). And the intersection between r_k and $F_{\text{deal}}=r_1 \cup r_2 \cup \ldots \cup r_{k-1}$, which is set of factors that belong to r_1, r_2, …, r_{k-1}, can be described as $r_k \cap F_{\text{deal}}=\{f_{k,1}, f_{k,2}, \ldots, f_{k,p}\}$.

(iii) If $r_k \cap F_{\text{deal}}=\varnothing$, that means all factors in r_k have not been assigned before and all corresponding positions are still empty. Check whether the size of T is smaller than the number of combinations of factors in r_k. If it is, add $\prod_{fi \in rk} a[i]$-$|T|$ empty test case into T firstly. And then, fill all $\prod_{fi \in rk} a[i]$ combinations into T as step one.

(iv) If $r_k \cap F_{\text{deal}} \neq \varnothing$, to cover all n_k-way combinations of factors in r_k, all p-way combinations in set $CombSet=\{(v_{k,1},\ldots,v_{k,p}) | v_{k,1} \in V_{k,1},\ldots,v_{k,p} \in V_{k,p}\}$ must be covered and appear at least $need=\prod_{fi \in rk - F\text{deal}} a[i]$ times in T. So for each combination in $CombSet$, count its appearance times $count$ firstly, and then complete this combination to make it appear $need$ times in T if $count<need$. The succeeding process is to fill $need$ different combinations of factors in r_k -F_{deal} into test cases that contain each combination in $CombSet$. All above process will guarantee that T satisfies r_k.

(v) Similar as step two, three and four, one coverage requirement will be selected and dealt each time, until all requirements in R have been satisfied by T. And some more detailed material is in Algorithm 1.

After the process of ReqOrder algorithm, note that there may be some positions that have not been assigned in T. But these unassigned positions do not reduce the coverage ability of test suite for interactions. Thus we name the values of these positions as "don't care" values and assign any valid value $x \in V_i$ in them.

Next we analyze the time complexity of algorithm 1. The number of combinations of factors in r_k (k=1, 2, …, t) is $\prod_{fi \in rk} a[i]$. And when filling each combination into test suite, all test cases in the suite should be scanned. So the worst time complexity when deal with r_k is $O(|T| \times \prod_{fi \in rk} a[i])$, and the worst time complexity of ReqOrder algorithm is $O(|T| \times \sum_{k=1}^{t} \prod_{fi \in rk} a[i])$ without considering the process of selecting requirement.

In description of algorithm 1, the function of SelectReq is to select a coverage requirement that have not been satisfied by test suite. At beginning, when all requirements in R have not been satisfied, it selects a requirement in which there are most combinations of factors. And in the other steps, the selected coverage requirement should have at least one factor that belongs to at least one requirement that have been dealt previously, and the number *assign_factor_comb_num*/*total_comb_num* is as great as possible, where the numerator is the number of combinations of these assigned factors in selected requirement and the denominator is the number of combinations of all factors in requirement. If there are not coverage requirement that satisfies this property, select randomly. Above calculation could be done in constant time, so this process do not effect the overall time complexity of ReqOrder algorithm.

At last, an example will be given to explain the process of RrqOrder algorithm. Consider a SUT with $F=\{3\times2\times2\times2\times3\}$ (it means that there are 5 factors totally and $|V_1|=3$, $|V_2|=|V_3|=|V_4|=2$, $|V_5|=3$), $R=\{r_1, r_2, r_3, r_4\}$ where $r_1=\{f_1, f_2\}$, $r_2=\{f_2, f_3, f_4\}$, $r_3=\{f_4, f_5\}$, $r_4=\{f_1, f_5\}$. The Figure 2 shows the process of generating test suite by ReqOrder algorithm, and the order of selected requirements is r_2, r_1, r_4, r_3. In the first two steps, r_2, r_1 are satisfied separately. And the next two steps deal with r_4. Because all combinations in r_3 have been covered after the fourth step, the algorithm stops here. In this example, the "don't care" values are not determined for simpleness.

–	1	1	1	–	1	1	1	1	–	1	1	1	1	–	1	1	1	1	1
–	1	1	2	–	2	1	1	2	–	2	1	1	2	–	2	1	1	2	1
–	1	2	1	–	3	1	2	1	–	3	1	2	1	–	3	1	2	1	1
–	1	2	2	–	–	1	2	2	–	1	1	2	2	–	1	1	2	2	2
–	2	1	1	–	1	2	1	1	–	1	2	1	1	–	1	2	1	1	3
–	2	1	2	–	2	2	1	2	–	2	2	1	2	–	2	2	1	2	2
–	2	2	1	–	3	2	2	1	–	3	2	2	1	–	3	2	2	1	2
–	2	2	2	–	–	2	2	2	–	2	2	2	2	–	2	2	2	2	3
										3	–	–	–	–	3	–	–	–	3

Figure 2. Process of ReqOrder

5. GENERATING TEST SUITE IN PARAMETER ORDER

In this section, we apply the in-parameter-order strategy, which has been successfully used in pair-wise test generation [2] and N-way combinatorial test generation [18], on the generation of combinatorial test suite for interaction relationship.

When generate combinatorial test suite with test generation algorithm in parameter order, an initial test suite will be constructed for a sub-system with only small number of factors firstly. Then extend the test suite by adding a new factor into this sub-system to get a test suite for the new sub-system, and this process will repeat until all n factors have been added into the sub-system to make it become the complete system. The detail is shown as follows:

Algorithm 2. Test Generation Algorithm in Parameter Order (ParaOrder)

Input: $F=\{f_1, f_2, \ldots, f_n\}$, $R=\{r_1, r_2, \ldots, r_t\}$
Output: Test suite T that satisfy R
begin
  f_i=SelectFactor(F);
  Create a test suite T with $a[i]$ empty test cases;
  Assign $a[i]$ values of f_i into T;
  $F=F-\{f_i\}$; $F_{\text{deal}}=\{f_i\}$;
  while $F \neq \varnothing$
    f_i=SelectFactor(F);
    $Inters=\{r_k \cap (F_{\text{deal}}+\{f_i\}) | f_i \in r_k, k=1, 2, \ldots, t\}$;
    $CombSet=\{(v_{i,1},\ldots,v_{i,j}) | v_{i,1} \in V_{i,1},\ldots,v_{i,j} \in V_{i,j}, \{f_{i,1},\ldots, f_{i,j}\} \in Inters\}$;
    //extending T horizontally
    for j = 1 to | T |
      Check all factors (except f_i) in elements of $Inters$ in j-th test case;
      if all above positions are not empty **then**
        for v = 1 to $a[i]$ //select value of f_i greedily
          Let $count_v$ be the number of combinations in $CombSet$ that will be covered by T if v is selected as the value of f_i in j-th test case;
        Select a value v with the largest $count_v$;
        $CombSet=CombSet-$\{combinations that covered by the

```
        j-th test case};
        if CombSet=∅ then break;
    //add new test cases to extending T vertically
    for each combination com in CombSet
      if there exist test cases in T that com can be added into
      then
        Select a test case that match the com mostly from above
        ones, and fill com into selected test case;
      else
        Add a new empty test case into T;
        Fill com into this test case;
    F=F-{f_i }; F_deal=F_deal∪{f_i };
end
```

(i) Select a factor and assume it is f_1 without loss of generality. Construct a test suite T with $a[1]$ empty test cases, and fill all $a[1]$ different values of f_1 into the first column of T.

(ii) Select another factor that has not been assigned in T. Without loss of generality, here we assume that the final sequence of selected factors is $f_1, f_2,\ldots, f_n$. So the factor f_i will be selected to be assigned in i-th iteration. We select all coverage requirements that contain f_i, and take the intersection between them and the set $\{f_1, f_2,\ldots, f_i\}$ to get a set of these intersections $Inters=\{r_k \cap \{f_1, f_2,\ldots, f_i\} \mid k=1, 2,\ldots, t, \text{ and } f_i \in r_k\}$. Consequently, the set of combinations of the factors in elements of *Inters* can also be obtained, that is $CombSet=\{(v_{i,1},\ldots,v_{i,j}) \mid v_{i,1} \in V_{i,1},\ldots,v_{i,j} \in V_{i,j}$ and $\{f_{i,1},\ldots, f_{i,j}\} \in Inters\}$. It is clear that all combinations in *CombSet* must be covered by T when deal with f_i,.

(iii) Extend test suite horizontally firstly. For each test case $test_k$ (k=1, 2,…, $|T|$) in T, if all positions corresponding to factors that interact with f_i and have been dealt in previous iterations are not empty, we choose a value v of f_i from V_i to make $test_k$ cover most combinations in *CombSet* and then refine it. Above process will stop when $k=|T|$ or $CombSet=\varnothing$. Note that there may be multiple values could make $test_k$ cover the same maximum number of uncovered combinations, therefore we select the one that appear least times since the beginning of horizontal extending. If there are still multiple candidates, select the first one in the turn $test_{k-1}[i]+1$, $test_{k-1}[i]+2,\ldots, a[i], 1,\ldots, test_{k-1}[i]$, where the $test_{k-1}[i]$ denote the value of f_i in test case $test_{k-1}$.

(iv) Check whether all combinations in *CombSet* have been covered by T. If not, the vertical extending is still needed by adding some new test cases into T to cover all uncovered combinations. Note that there may be test case in which above uncovered combinations could be added. So for each uncovered combination in *CombSet*, firstly attempt to find test cases that some positions corresponding to this combination are empty while all others match this combination, then find out the one that the number of empty positions is the least and fill this combination into it. If there are not above test cases, add an empty test case and copy the combination into it.

(v) Deal with all factors in turn similarly as step two, three and four. And one coverage requirement will be satisfied once all factors in it have been dealt, hence all coverage requirements in R will be satisfied by T after all factors in F haven been assigned. This process is also shown as Algorithm 2. And after above process, deal with the positions that have not been assigned similarly as mentioned before.

Next we analyze time complexity. When deal with factor f_i (i=2,…, n), the construction of set *Inters* need take the intersection of each coverage requirement and the set of all determined factors, hence the time complexity of this step is $O(\sum_{k=1}^{t}(|r_k|+i))$. The worst time complexity $O(\sum_{k=1}^{t}\prod_{fi\in rk} a[i])$ of constructing *CombSet* is equal to its size. So the worst time complexity of horizontal and vertical extending is $O(a[i]\times|T|\times\sum_{k=1}^{t}\prod_{fi\in rk} a[i])$ and $O(|T|\times\sum_{k=1}^{t}\prod_{fi\in rk} a[i])$ respectively. Hence the overall theoretical worst time complexity of ParaOrder is $O(|T|\times(\sum_{i=1}^{n} a[i])\times(\sum_{k=1}^{t}\prod_{fi\in rk} a[i]))$ without considering the process SelectFactor.

In description of algorithm 2, the function of SelectFactor is to select a factor that haven not been dealt as given strategy. Here we define the rule of factor selection is to choose the one that have greatest number of values. And if there multiple factors have with the same maximum number, select by their subscript as ascending. It means that the order of selected factors can be sorted before the running of algorithm, and its time complexity is $O(n\times\log(n))$ which dose not effect the overall time complexity of ParaOrder algorithm.

For the system that mentioned in section 4, an example of horizontal extending and vertical extending is shown in Figure 3. As the rule of factor selection, the final order of factors is f_2, f_3, f_4, f_1, f_5. Here we assume that f_2, f_3 have been dealt, and the current factor is f_4. Three parts of Figure 2 are initial test suite before dealing with f_4, horizontal extended test suite and vertical extended test suite separately. And the fourth one is the final generated test suite.

–	1	1	–	–	–	1	1	1	–	–	1	1	1	–	1	1	1	1	1
–	1	2	–	–	–	1	2	2	–	–	1	2	2	–	2	1	2	2	2
–	2	1	–	–	–	2	1	1	–	–	2	1	1	–	3	2	1	1	3
–	2	2	–	–	–	2	2	2	–	–	2	2	2	–	1	2	2	2	3
										–	1	1	2	–	3	1	1	2	1
										–	1	2	1	–	1	1	2	1	2
										–	2	1	2	–	2	2	1	2	3
										–	2	2	1	–	3	2	2	1	2
															2	–	–	–	1

Figure 3. Process of ParaOrder

6. EXPERIMENT

To evaluate the effectiveness of our algorithms, a combinatorial test generation tool was developed. Rather than two algorithms proposed in this paper, two test generation algorithms in IO relationship testing model were also included in it. In the experiment, we will compare these four algorithms in the field of size of generated test suite and execution time.

The implementation of Union and Greedy algorithm is as the description in [13]. In Union algorithm, when construct test suite for a given coverage requirement, the value of all "don't-care" factors will be assigned randomly. And in Greedy algorithm, the problem reduce technique was not implemented for simplicity. To save the execution time when searching the best test case that covers greatest number of uncovered combinations, a back-track method based on solution space tree [19] was used in the implementation of Greedy.

In experiment, the factor set and the valid value collections of these factors will be given firstly. We choose two factor set

Table 1. Compare different algorithms for the systems with $F=\{3^{10}\}$ and different size of interaction relationship

$\|R\|$	ReqOrder	ParaOrder	Union	Greedy
10	153 1”	104 1”	502 1”	104 241”
20	148 1”	105 1”	858 1”	110 393”
30	151 1”	131 1”	1599 1”	122 489”
40	160 1”	133 1”	2057 2”	134 613”
50	169 1”	146 2”	2635 2”	138 741”
60	176 1”	148 1”	3257 2”	143 918”

Table 2. Compare different algorithms for the systems with $F=\{2^3\times3^3\times4^3\times5\}$ and different size of interaction relationship

$\|R\|$	ReqOrder	ParaOrder	Union	Greedy
10	154 1”	144 1”	505 1”	137 342”
20	187 1”	162 1”	929 1”	158 467”
30	207 1”	190 2”	1861 2”	181 816”
40	203 1”	191 2”	2244 1”	183 1163”
50	251 1”	205 2”	2820 2”	198 1332”
60	250 1”	213 3”	3587 2”	207 1521”

$F_1=\{3^{10}\}$ and $F_2=\{2^3\times3^3\times4^3\times5\}$ to represent the set with uniform factors and mixed factors respectively.

The next step is to determine the interaction relationship and the coverage requirements in it. The collection of all coverage requirements that will be used in experiment is described in appendix. All 60 requirements that generate randomly in this collection satisfy all required properties of interaction relationship that mentioned in section 2. Note that there are two reasons that the strength of each coverage requirement is selected as 2~4. The first one is that Kuhn D R et al discovered that the failure-triggering fault interaction numbers in many software systems are usually not bigger than 4 to 6 [20] and most faults are triggered by the interactions with low strength [21]. And the second one is that the size of test suite will be not mainly determined by the effectiveness of different algorithms, but some few coverage requirements that with high strength such as 5 or 6. It is also clear that the reason why the number of requirements is 60 is that the number of all 2-way, 3-way and 4-way interactions for a system with 10 factors is 45,120 and 210 respectively. So the size 60 is appropriate for the property that there is not subsumption between different coverage requirements.

There will be 6 iterations in experiment for each factor set. In the first iteration, we select first 10 coverage requirements in collection of coverage requirements to form the interaction relationship. And then, the interaction relationship will be modified by adding 10 following coverage requirements each time. And in the sixth iteration, there will be 60 coverage requirements in interaction relationship.

Table 1 and Table 2 display the sizes of generated test suite and the time consumed for the generation. We compare four algorithms by the sizes of generated test suite firstly. The sizes of test suites generated by the Union algorithm are much bigger than that generated by other three algorithms. ReqOrder is worse than ParaOrder and Greedy, but much better than Union. The sizes of test suite that generated by ParaOrder are worse that that of Greedy in most times, but the gap is very small and sometimes the former may be better.

Next, we compare the theoretical time complexity and actual running time of four algorithms. The worst time complexity of Union, ReqOrder, ParaOrder and Greedy are $O(\sum^{t}_{k=1}\prod_{fi\in rk} a[i])$, $O(|T|\times\sum^{t}_{k=1}\prod_{fi\in rk} a[i])$, $O(|T|\times(\sum^{n}_{i=1}a[i])\times(\sum^{t}_{k=1}\prod_{fi\in rk} a[i]))$ and $O(|T|\times(\prod^{n}_{i=1}a[i])\times(\sum^{t}_{k=1}\prod_{fi\in rk} a[i]))$ respectively, which are ordered as ascending. The experiment data also supports this theoretical result. Greedy need several hundreds or even thousands seconds though we use a black-track method to improve its effectiveness, while the running time of all other three algorithms are limited into 1 or 2 second for all different inputs.

7. CONCLUSION

This paper proposed a new interaction relationship based combinatorial testing method by extending the conventional combinatorial testing model and generalizing the IO relationship testing model. As a more general case of combinatorial testing, interaction relationship based combinatorial testing method may reduce the size of test suite without decrease of the fault detect ability. The reason is that there are not always interactions among any factors and the strength of these interactions are not always equal. The neighbor factors combinatorial testing method, which can be considered as a special case of interaction relationship based combinatorial testing, has been proved to be effective than *N*-way combinatorial testing in the application of testing for railway signaling systems [17]. Besides, two test generation algorithms were also proposed and compared with similar algorithms in IO relationship testing model by experiment. The experience data shows the advantage of our algorithms.

The studies on combinatorial testing have obtained many results, but there are also many problems: firstly, existed model of combinatorial testing do not consider the constraint between the factors and their values sufficiently, but there may be constraint between them in many cases; secondly, the study on combinatorial testing based fault location and debugging techniques is still lacking. Corresponding, these are following problems are needed to study in the future works: combinatorial testing method in the case that there are constraints among factors; combinatorial testing based fault location and debugging; fault detection ability of combinatorial testing, etc. Besides, automatic tools for combinatorial testing, which may support the automation of test generation, test execution, test measurement, and fault location etc, are also needed to be designed and developed.

8. ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (60425206, 60633010, 60403016, and 60503033), Natural Science Foundation of Jiangsu Province (BK2005060), High Technology Research Project of Jiangsu Province (BG2005032), Excellent Talent Foundation on Teaching and Research of Southeast University, and Open Foundation of State Key Laboratory of Software Engineering in Wuhan University.

9. REFERENCES

[1] Cohen D M, Dalal S R, Fredman M L, et al. The AETG system: An approach to testing based on combinatorial design. *IEEE Trans. on Software Engineering*, 1997, 23, 7: 437-444.

[2] Tai K C, Lei Y. A test generation strategy for pairwise testing. *IEEE Trans. on Software Engineering*, 2002, 28, 1: 109-111.

[3] Tung Y W, Aldiwan W S. Automating Test Case Generation for the New Generation Mission Software System. In *Proceedings of IEEE Arospace Conf*., 2000, pp. 431-437.

[4] Colbourn C J, Cohen M B, and Turban R C. A deterministic density algorithm for pairwise interaction coverage. In: *Proceedings of IASTED International Conference on Software Engineering (SE2004)*, Innsbruck, Austria, 2004, 345-352.

[5] Noritaka Kobayashi, Tatssuhio Tsuchiya, Tohru Kikuno. A new method for constructing pair-wise covering designs for software testing. *Information Processing Letters*, 2002, 81, 2: 85-91.

[6] Williams A W. *Software component interaction testing: Coverage measurement and generation of configurations.* Ph.D Thesis, Ottawa-Carleton Institute for Computer Science, School of Information Technology and Engineering, University of Ottawa, Canada, 2002.

[7] Cohen M B, Colbouns C J, Ling A C H. Augmenting simulated annealing to build interaction test suites. In: *Proceedings of 14th International Symposium on Software Reliability Engineering (ISSRE 2003)*, Denver Colorado, November 2003: 394-405.

[8] Toshiaki Shiba, Tatsuhiro Tsuchiya, Tohru Kikuno. Using artificial life techniques to generate test cases for combinatorial testing. In: *Proceedings of 28th International Computer Software and Applications Conference (COMPSAC2004)*, HongKong, China, 2004, 72-78.

[9] Schroeder P J, Bolaki P, Gopu V. Comparing the fault detection effectiveness of n-way and random test suite. In: *Proceedings of 2004 International Symposium on Empirical Software Engineering (ISESE2004)*, Redondo Beach, California, 2004, 49-59.

[10] Yan Jun, Zhang Jian. Backtracking algorithms and search heuristics to generate test suites for combinatorial testing. In: *Proceedings of 30th Annual International Conference on Computer Software and Applications (COMPSAC06)*, Volume 1, Sept. 2006:385 – 394.

[11] Yalmaz C, Cohen M B, Porter A A. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Trans. on Software Engineering*, 2006, 32, 1: 20-34.

[12] Schroeder P J and Korel B. Black-box Test Reduction Using Input-output Analysis. In: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'00)*, Portland, Oregon, Aug, 2000:21-22.

[13] Schoeder P J, *Black-box test reduction using input-output analysis*. Ph.D. Thesis, Department of Computer Science, Illinois Institute of Technology, Chicago, IL,USA, 2001.

[14] Schroeder P J, Faherty P, Korel B. Generating expected results for automated black-box testing. In: *Proceedings of 17th IEEE International Conference on Automated Software Engineering (ASE'02)*.

[15] Cheng C, Dumitrescu A, Schroeder P J. Generating small combinatorial test suites to cover input-output relationships. In: *Proceedings of Third International Conference on Quality Software*, Nov 2003: 76-82.

[16] Bach J, Schroeder P J. Pairwise testing: a best practice that isn't. In: *Proceedings of 22nd Pacific Northwest Software Quality Conference*, 2004, 180-196.

[17] Nie Changhai, Xu Baowen, Wang Ziyuan, Shi Liang. Generating optimal test set for neighbor factors combinatorial testing. In: *Proceedings of Sixth International Conference on Quality Software*, Oct 2006: 259-265.

[18] Changhai Nie, Baowen Xu, Liang Shi, Guowei Dong. Automatic test generation for n-way combinatorial testing. In: *Proceedings of Second International Workshop on Software Quality (SOQUA2005)*, Fair and Convention Center, Erfurt, Germany, 2005. *Lecture Notes in Computer Science* 3712, 2005: 203-211.

[19] Nie Changhai, Xu Baowen, Shi Liang, Wang Ziyuan. A new heuristic for test suite generation for pairwise testing. In: *Proceedings of 18th International Conference on Software Engineering and Knowledge Engineering (SEKE2006)*.

[20] Kuhn D R, Reilly M J. An investigation of the applicability of design of experiments to software testing. In: *Proceedings of 27th NASA/IEEE Software Engineering Workshop*, NASA Goddard Space Flight Center, 2002, 91-95.

[21] Kuhn D R, Wallace D R. Software fault interaction and implication for software testing. *IEEE Trans. on Software Engineering*, 2004, 30, 6: 1-4.

APPENDIX

The collection of coverage requirements in experiment is shown as below. There are 10 factors in both F_1 and F_2, that is $F=\{f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}\}$. We could describe these 10 factors by its sequence number, and describe the factor set as $F=\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ for short.

COLLECTION={{1, 2, 7, 8}, {0, 1, 2, 9}, {4, 5, 7, 8}, {0, 1, 3, 9}, {0, 3, 8}, {6, 7, 8}, {4, 9}, {1, 3, 4}, {0, 2, 6, 7}, {4, 6}, {2, 3, 4, 8}, {2, 3, 5}, {5, 6}, {0, 6, 8}, {8, 9}, {0, 5}, {1, 3, 5, 9}, {1, 6, 7, 9}, {0, 4}, {0, 2, 3}, {1, 3, 6, 9}, {2, 4, 7, 8}, {0, 2, 6, 9}, {0, 1, 7, 8}, {0, 3, 7, 9}, {3, 4, 7, 8}, {1, 5, 7, 9}, {1, 3, 6, 8}, {1, 2, 5}, {3, 4, 5, 7}, {0, 2, 7, 9}, {1, 2, 3}, {1, 2, 6}, {2, 5, 9}, {3, 6, 7}, {1, 2, 4, 7}, {2, 5, 8}, {0, 1, 6, 7}, {3, 5, 8}, {0, 1, 2, 8}, {2, 3, 9}, {1, 5, 8}, {1, 3, 5, 7}, {0, 1, 2, 7}, {2, 4, 5, 7}, {1, 4, 5}, {0, 1, 7, 9}, {0, 1, 3, 6}, {1, 4, 8}, {3, 5, 7, 9}, {0, 6, 7, 9}, {2, 6, 7, 9}, {2, 6, 8}, {2, 3, 6}, {1, 3, 7, 9}, {2, 3, 7}, {0, 2, 7, 8}, {0, 1, 6, 9}, {1, 3, 7, 8}, {0, 1, 3, 7}}.