I.FAILURE-INDUCING COMBINATIONS IDENTIFYING MODEL

We will introduce our failure-inducing identifying model with an example SUT, The SUT consists of 4 parameters, each having 3 values. We generated a 2-way covering array to test it. The result is listed in Table III.

TABLE I. EXECUTED TEST CONFIGURATIONS

No.	Test configuration				Result
1	1	1	1	1	Pass
2	1	2	2	2	Pass
3	1	3	3	3	Pass
4	2	1	2	3	Pass
5	2	2	3	1	Fail
6	2	3	1	2	Pass
7	3	1	3	2	Pass
8	3	2	1	3	Pass
9	3	3	2	1	Pass

A. Existed Maximal right tuples and Existed Minimal bug tuples

How can we determine which tuples are unknown, i.e., neither been selected to test nor can be determined by tuples that had been tested so far according to proposition3 or 4, during identifying process? If we select a tuple that can already be determined so far without any extra information, then the behavior of generating test cases to test the tuple we select is a waste of computing resources. So we need record the tuple that had been tested so far along with its testing result to help us avoiding such wasting.

We called the tuple we selected and tested so far *existed tuple*, if the tuple is tested to be right tuple , then we called the tuple *existed right tuple*, and if the tuple is tested to be wrong tuple, we called the tuple *existed bug tuple*.

But the question is shall we need to record all of them? The answer is: no. We just need to record the *minimal existed bug tuples*, i.e., the existed bug tuples that hasn't have any child tuple that is existed bug tuple, and *maximal right tuples*, i.e., the existed right tuples that hasn't have any parent tuple that is existed right tuple. The reason is obviously, as we just need take use of them to determine whether a tuple is unknown tuple or not. And as we already know, if a tuple is neither the father of any *existed bug tuple* nor the child of any *existed right tuple*, then the tuple is an *unknown tuple*. So for the first condition, we can use *minimal existed bug tuples* instead of all the *existed tuples* to test whether the chosen tuple is a bug tuple. We take the followed example to illustrate the correctness.

Assume that we have a tuple t_a to determine, and we find that t_a is the parent of t_b, which is an existed bug tuple, and then we can determine t_a is a bug tuple, which is not unknown. But assume we also have t_c which is the child of t_b and t_c is also an existed bug tuple. It is easy to see that t_a is must also be the parent of t_c. Then we can just record t_c instead of recording both t_c and t_b. This will not influence the determination of tuple t_a, for that we can still determine the t_a is a bug tuple because it must be the father of t_c which is an existed bug tuple.

The reason why we just record the maximal existed right tuple is similar to the minimal wrong tuple. So we need compress the existed wrong tuples and existed right tuples when we add some newly identified tuples to them to ensure them are always the minimal and maximal, i.e., we need delete some tuples when we add newly identified tuples in the two sets.

What we need consider next is: how to initialize this two data sets, add new tuples to these two data sets and compress this two data sets

Firstly, the initialization: the initial process initializes the two data through known result of test cases. The initial minimal bug tuple is must be the tuple which comprise of all the parameter of the failed test case, which in this case is: [2, 2, 3, 1]. The initial right tuple of this failed test case are which existed in these passed test cases. If there are no passed test cases this data set is null.

Secondly, the adding process shall be executing each time a newly unknown tuple is selected and tested. If the tuple is tested to be right tuple, then we add it tuple the *existed right tuples*, otherwise we will add it to the *existed bug tuples*. After the adding process, we will do the compress process to make sure the two data sets to be *minimal existed bug tuples* and *maximal existed right tuples*.

The compress process is simple; we just need go through the tuples of the *existed bug tuples* (*existed right tuples*) and delete these tuples which has child tuples (father tuples) that are also in the e*xisted bug tuples* (*existed right tuples*).

B. Get the tuples that is unknown

As we have recorded this two data sets. Then how we get an unknown tuple?

Still, we should make sure the tuple we get must satisfy the followed two principles:

1. Must not be the parent tuple of any *existed minimal bug tuples.*
2. Must not be the child tuple of any *existed maximal right tuples.*

So, for the first principle, we should make the tuple satisfy that: for each existed minimal bug tuple, this tuple must not contain it, i.e., not contain at least one parameter of it. In fact, we don't need list every tuple that satisfy the situation; we just need give the tuples that has maximum number of parameters and satisfy the situation stimulatingly(We called c*andidate maximum head tuples*). Then according to the proposition 4, we can know that each child of this kind of tuple is must also satisfy the situation.

The algorithm that gets the *candidate maximum head tuples* is described in Algorithm 1. The main idea is that we want to make gradual variations of an existed bug tuple to a tuple that don't have any child tuple that is existed bug tuple. The original existed bug tuple we chosen is the root tuple, i.e., the tuple which comprise of all the parameter of the failed test case (line 1). We do that because we want to get the unknown tuple with maximum number of parameters. Next, we will deep into the iteration of our algorithm, in each turn, we will do that: we firstly get an existed bug tuple from the existed bug tuples set (line 5). And then for each tuple need to be altered this iteration, we will remove a parameter of it which is also in the existed bug tuple we get previously (line 8 ~ 10). We will add the newly generated tuple to a list (line 11) and then compress it (line 12) as the list of tuples need to be changed for the next iteration (line 13). The compression process is just deleting tuples that has father tuple which is also in the variations. (Note: this compression is different from the compression of existed bug tuples, for this compression is to get the head tuples as big as possible, while the existed bug tuples compression is to get tuples as small as possible)

Algorithm 1 Get the candidate maximum head tuples

```
1:  define getCandiMaxiHeads(existedBugtuples):
2:      Tuple root = getRoot()
3:      List itr
4:      itr.add(root)
5:      for bug in existedBugtuples:
6:          List nextItr
7:          while((tuple = itr.pop()) != NULL):
8:              for param in bug:
9:                  temp = tuple.copy()
10:                 temp.remove(param)
11:                 nextItr.add(temp)
12:         compress(nextItr)
13:         itr = nextItr.
14:     Return itr
```

For example, for failed test case: (2,2,3,1), and assume that the existed minimal bug tuples we get the current iteration is [2,-,3,-] and [-,2,3,-].

Then our process of getting *candidate maximum head tuples* is as Figure 1:

According to figure 1, we can see that our process totally has two iterations for we just have two existed bug tuples. At the first iteration, our tuples that need to be altered for this turn is the root tuple [2, 2, 3, 1]. And then for the first existed bug tuple [2,-,3,-],we get two variations as the bug tuple has two parameters. Each variation is a tuple that removing one parameter that is in the [2,-,3,-] of the original tuple [2,2,3,1].They are: [-,2,3,1] and [2,2,-,1]. The last step of the first iteration is compressing the two variations, and because we just have two variations and they have no relationships, so we keep them all. The second iteration is similar to the first iteration. The only differences are that the tuples need to be altered for this turn is the result of last iteration: [-,2,3,1] and [2,2,-,1], and for the second existed bug tuple [-,2,3,-] , we get four variations: [-,-,3,1] , [-,2,-,1], [2,-,-,1],[2,2,-,1]. Among them, the tuple [2,2,-,1] are the parent tuple of [-,2,-1] and [2,-,-,1], so at the compression, we only get two tuples:[-,-,3,1] and [2,2,-,1].

Then for the second principle, similar to the first principle, we should make the tuple satisfy that: for each *existed maximal right tuple*, this tuple must not be the child of it, i.e., must contain at least one parameter which not in it. Still, we don't need list every tuple that satisfy the situation; we just need give the tuples that has minimal number of parameters and satisfy the situation stimulatingly (We called *candidate minimal tail tuples*). Then according to the proposition 3, we can know that each father of this kind of tuple is must also satisfy the situation.

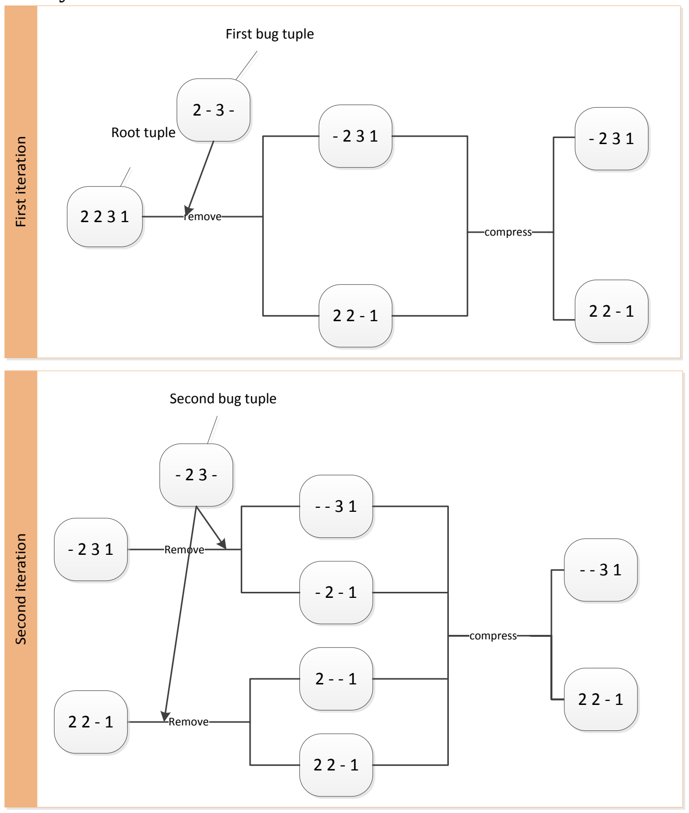

Fig. 1. Two iteration

The algorithm that gets the *candidate minimum tail tuples* is described in Algorithm 2. It is very similar to the algorithm 1except that we get started from an empty tuple i.e., the tuple which comprise of none the parameter (line 1), for we want to get the unknown tuple with minimum number of parameters. Another difference is we will get a reverse tuple of the existed right tuple (line 6), and then we will add one parameter of it to the tuple that need to be altered at current iteration. (line 9 ~ 11). This step is ensuring that we will have a tuple contain at least one parameter which is different from the existed right tuple. There is still a process of compression (line 13) for this algorithm, but different from algorithm 1, it will delete these tuples which has child tuples that is also in the variations at current iteration.

Algorithm 2 Get the candidate minimal tail tuples

```
1:  define getCandiMiniTails(existedRightTuples):
2:      Tuple empty = getEmpty()
3:      List itr
4:      itr.add(empty)
5:      for right in existedRightTuples:
6:          reverse = right.reverse()
7:          List nextItr
8:          while((tuple = itr.pop()) != NULL):
9:              for param in reverse:
10:                 temp = tuple.copy()
```

```
11:                 temp.add(param)
12:                 nextItr.add(temp)
13:             compress(nextItr)
14:             itr = nextItr.
15:         Return itr
```

For example, still for failed test case: (2,2,3,1), and assume that the existed maximal right tuples we get the current iteration is [-,2,-,1] and [2,-,-,1].

Then our process of getting the tuples not contains at least one of the existed bug tuples is listed as follows:

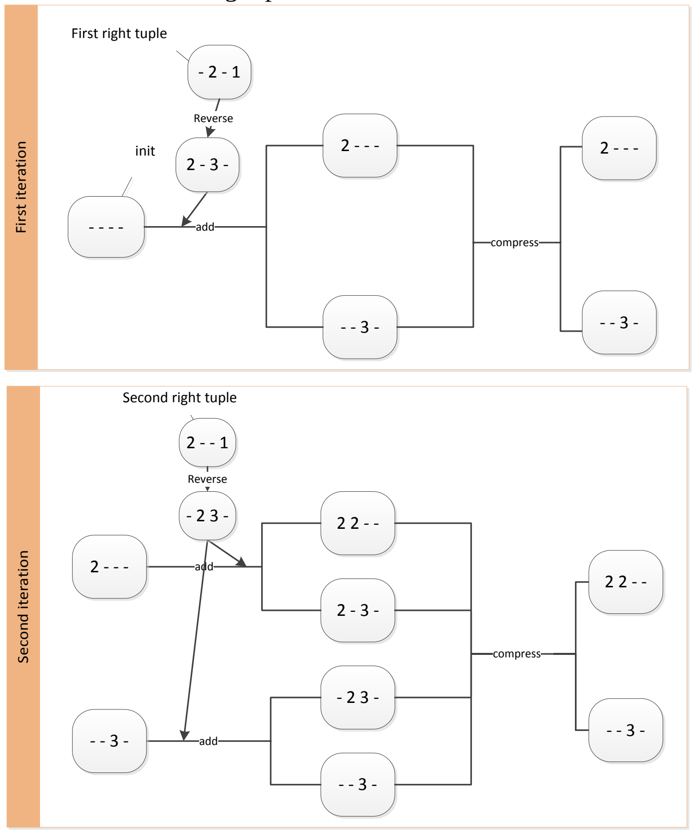

Fig. 2. Two iteration

We can see that it is also very similar to the figure 1, so we will not describe it again.

As we have got this two set of tuples: *candidate maximum head tuples* and *candidate minimum tail tuples*. So choosing an unknown tuple which stratify the two principles is transferring to choose a tuple that must be the child (or the original)of at least one *candidate maximum head tuple* and the father (or the original)of at least one *candidate minimum tail tuple*. As long as we go through this two set, we can easily find the unknown tuples.

C. Select strategy

The order of selecting unknown tuples to be tested has a significant influence on the performance of our algorithm. Before we describe this strategy we give definitions as follows:

Definition 6 A *path* is a sequence of tuples in which every tuple is the direct parent of the tuple that follows.

For example, [2,2,3,1]→[2,2,3,-]→[2,-,3,-]→[2,-,-,-] is a path.

Moreover, we called a path the *unknown path* if every tuple in this path is an unknown tuple. And we called a path the *longest unknown path* when this path is an unknown path that has the max number of tuples.

Our selecting strategy is based on the followed two steps.

1. Get the longest unknown paths

When select tuples to be tested，we always want to select as few as possible. Get a path of tuples and then select tuple from the path is good for reducing the number of tuples needed to be tested, for that if we determine a tuple of the path, other tuples will be determined according to proposition 3 or 4. So naturally we get the greedy strategy, i.e., selecting the longest unknown paths and from which we will choose tuple to be tested every time.

The algorithm that describes getting the longest unknown paths is listed in Algorithm 3. In this algorithm, we firstly get the maximum candidate bug tuples and minimum tail tuples. (Line 2~ 3)And then for each candidate existed bug tuple and candidate existed right tuple, we will compute their distance (line 7 ~ 9), we define the distance as follows:

$$\mathrm{distance}(\mathrm{t_a},\mathrm{t_b}) = \begin{cases} -1, & t_b \text{ is not child of } t_a \\ size(\mathrm{t_a}) - size(t_b)\,, & otherwise \end{cases}$$

We will get two tuples which has the maximum distance (line 10 ~12) and generate a path based on them (line 13).

Algorithm 3 Get longest paths

```
1:      define getLongest(existedBugtuples, existedRight):
2:          canHeads = get1(existedBugtuples)
3:          canTails = get1(existedRighttuples)
4:          max = 0
5:          maxHead = null
6:          maxTail = null
7:          for head in canHeads:
8:              for tail in canTails:
9:                  if distanceof(head, tail) > max:
10:                     max = distanceof(head,tail)
11:                     maxHead = head
12:                     maxTail = tail
13:         return sequnce(maxHead, maxTail)
```

As for the previous example, we will get the candidate heads:[-,-,3,1] and [2,2,-,1].And candidate tails:[2,2,-,-] and [-,-,3,-].Then we will find([2,2,-,1] , [2,2,-,-]) and ([-,-,3,1], [-,-,3,-])is both the have the maximum distance: 1. So either of them can be chosen, e.g., we chose ([2,2,-,1] , [2,2,-,-]) Then will generate a path of tuples based on them. But since the distance is 1, which means we will not necessarily generate additional tuples. So at last, we will get the longest path ([2,2,-,1] , [2,2,-,-]).

Note: if the distance is bigger than 1, e.g. ([2,2,-,1] , [2,-,-,-]), we need generate additional tuples , which must be the child of the head tuple and parent of tail tuple, to get a path, e.g., ([2,2,-,1] , [2,-,-,1], [2,-,-,-])

2. Binary search

As could get the longest path from existed bug tuples and existed right tuples, then how we choose a tuple to be tested at the current turn as we have one longest path? Consider the followed condition: if we get a tuple from the path, and test it to be right tuple, then the tuples after this one in the path will be determined to be right tuple, if this tuple is determined to be bug tuple, then the tuples before it in the path will be determined to be bug tuples. This condition is very similar to the application of binary search exactly. So our select strategy is also very similar to binary search, which is described in algorithm 4.

Algorithm 4 Select

```
define select (existedBugTuples,existedRightTuples):
    if( head > tail):
        longest = getLongest(bugtupls,righttuples)
        head = 0
        tail = len(longest) -1
        middle = head
    else:
        middle = (head+tail)/2
    return longest[middle]
```

D. Identifying process

We will make the following assumption to facilitate the process of confirming the states of unknown tuples in the TRT:

Assumption: The generated extra test configuration will not introduce new bug tuples. (This assumption will not always be right. We will remove this assumption later.)

Based on this assumption, we will get the following lemma.

Lemma 1: For a tuple, we generate an extra test configuration that contains this tuple. If the extra test configuration passes, then the tuple is a right tuple. If the extra test configuration fails, then the tuple is a bug tuple.

Proof. According to Definition 2, it is obvious that this tuple is a right tuple when the extra test configuration passes.

When the extra configuration fails, this is a bug tuple (or there exists no bug tuple and this test configuration would not fail).because the assumption says that this extra configuration will not introduce new bug tuples. ∎

The process of confirming all the unknown tuples is described in Algorithm 5. Firstly, our approach initialize the existed bug tuples and existed right tuples (line 2 ~ 3). And then deep into the loop: we will select a tuple using the select algorithm previously (line 5). If there are no tuple selected this turn, the algorithm will break the loop and ended (line 6 ~ 7). Otherwise, we will generate a test case to test the selected tuple. After we test it, we will do either the process: “UpdateAfterRight” (line 11) or “updateWrong” (line 13), which will change the tail or head in the algorithm before.

Algorithm 5 Identifying process

```
1:  define identify(TRT):
2:      initExistedBugTuples()
3:      initExistedRightTuples()
4:      while true:
5:          tuple   =   select(existedB, exitstedR)
6:          if tuple == NULL:
7:              break;
8:          test_config = gen_extra(tuple)
9:          result =   exec(test_config)
10:         if result == fail:
11:             updateWrong()
12:         else:
13:             updateRight()
```

We will give a complete example as follows: