Beyond OFOT: A more efficient and effective MFS Identification method

Xintao Niu, Changhai Nie, *Member, IEEE,* and

Abstract—Combinatorial testing (CT) aims to detect the failures which are triggered by the interactions of various factors that can influence the behaviour of the system, such as input parameters, configuration options, and specific events. Many studies in CT focus on designing a elaborate test suite (called covering array) to reveal such failures. When a interaction failure is detected, it is desired to identify the root cause of that failure, i.e., the failure-inducing interactions. Although covering array can assist testers to systemically check each possible factor interactions (without masking effects), however, it provides weak support to locate the failure-inducing interactions. Hence, more auxiliary information are needed to filter out those failure-irrelevant interactions and obtain the failure-inducing ones. Recently some elementary researches are proposed to handle the failure-inducing interaction identification problem. However, some issues, such as unable to identify multiple failure-inducing interactions, and generating too many additional test cases, negatively influence the practicability of these approaches. In this paper, we propose a novel failure-inducing identification approach which aims to handle those issues. The key of our approach is to search for a proper factor interaction at each iteration to check whether it is failure-inducing or not until all the interactions in a failing test cases are checked. To reach this target, we maintain two data structures to guide the selection, i.e., CMINFS(represent for candidate minimal faulty schemas) and CMAXHS(represent for candidate maximal healthy schemas). With these two data set, an optimal interaction can be selected to check so that we can reduce as many additional test cases as possible, as well as not ignore any unchecked-interactions, e.g., the interactions with large number of factors. Our approach will repeat updating the interactions in these two set after a interaction is checked. Moreover, we conduct empirical studies on both widely-used real highly-configurable software systems and synthetic softwares. Results showed that our approach obtained a higher quality at the failure-inducing interaction identification, while just needed a smaller number of additional test cases.

Index Terms—failure-inducing interactions, fault location, debugging aids, combinatorial testing

1 INTRODUCTION

THe behavior of modern software are affected by many factors, such as input parameters, configuration options, and specific events. To test such software system is challenging, as in theory we should test all the possible interaction of these factors to ensure the correctness of the System Under Test (SUT)[1]. When the number of factors is large, the interactions that are needed to check increase exponentially. Hence it is, if possible, not practical to conduct such exhaustive testing. Combinatorial testing (CT) is a promising solution to handle the combinatorial explosion problem. Instead of testing all the possible interactions in a system, it focus on checking those interactions with number of involved factors no more than a prior number. Commonly when applying CT, a elaborate test suite (called covering array) will be generated to cover the valid interaction that needs to be checked. Although covering array is effective and efficient as a test suite, however, it provides weak support to distinguish the failure-inducing interactions from all the interactions.

Consider the following example [2], Table 1 presents a pair-wise covering array for testing an MS-Word application in which we want to examine various pair-wise interactions of options for 'Highlight', 'Status Bar', 'Bookmarks' and 'Smart tags'. Assume the third test case failed. We can get five pair-wise suspicious interactions that may be responsible for this failure. They are respectively (Highlight: Off, Status Bar: On), (Highlight: Off, Bookmarks: Off), (Highlight: Off, Smart tags: Off), (Status Bar: On, Bookmarks: Off), (Status Bar: On, Smart tags: Off), and (Bookmarks: Off, Smart tags: Off). Without additional information, it is difficult to figure out the specific interactions in this suspicious set that caused the failure. In fact, considering that the interactions consist of other number of factors could also be failure-inducing interactions, e.g., (Highlight: Off) and (Highlight: Off, Status Bar: On, Smart tags: Off), the problem becomes more complicated. Generally, to definitely determine the failure-inducing interactions in a failing test case of n factors, we need to check all the 2^n-1 interactions in this test case, which is not possible when n is a large

- *Xintao Niu and Changhai Nie are with the State Key Laboratory for Novel Software Technology, Nanjing University, China, 210023. E-mail: niuxintao@gmail.com, changhainie@nju.edu.cn*
- *Hareton Leung is with Department of computing, Hong Kong Polytechnic University, Kowloon, Hong Kong. E-mail: hareton.leung@polyu.edu.hk*
- *Jeff Y. Lei is with Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, Texas. E-mail: ylei@cse.uta.edu*
- *Xiaoyin Wang is with Department of Computer Science, University of Texas at San Antonio. E-mail: Xiaoyin.Wang@utsa.edu*

Manuscript received April 19, 2005; revised September 17, 2014.

TABLE 1
MS word example

id	*Highlight*	*Status bar*	*Bookmarks*	*Smart tags*	**Outcome**
1	On	On	On	On	PASS
2	Off	Off	On	On	PASS
3	Off	On	Off	Off	Fail
4	On	Off	Off	On	PASS
5	On	Off	On	Off	PASS

number.

To address this problem, prior work [3] specifically studied the properties of the failure-inducing interactions in SUT, based on which additional test cases were generated to identify them. Other approaches to identify the failure-inducing interactions in SUT include building a tree model [4], adaptively generating additional test cases according to the outcome of the last test case [5], ranking suspicious interactions based on some rules [6], and using graphic-based deduction [7], among others. These approaches can be partitioned into two categories [8] according to how the additional test cases are generated: *adaptive*–additional test cases are chosen based on the outcomes of the executed tests [3], [6], [9], [5], [10], [11], [12]or *nonadaptive*–additional test cases are chosen independently and can be executed parallel [4], [8], [7], [13].

These approaches, however, are essentially approximate solutions to failure-inducing identification (Theoretically, definite solution is of exponential computational complexity). Hence, many issues may affect their effectiveness when applied in practice. Generally, *non-adaptive* approaches usually can accurately identify the failure-inducing interactions, even when there are multiple ones in the SUT. Their effectiveness are based on some mathematical objects [8], [7], [13]. The shortcomings of these approaches are they are very ad-hoc, that is, they are restricted to many limitations, such as the number of failure-inducing interactions must be given as well as the number of the maximal factors involved in a failure-inducing interaction. Moreover, these approaches usually consume many test cases [5]. *Adaptive* approaches are much more flexible, and they mainly focus on one failing test case. Commonly they are needs much less test cases than *non-adaptive* approaches. However, some problems, such as cannot handle multiple failure-inducing interactions (especially they are have overlapping factors), and cannot handle the newly introduced failure-interactions in the additional generated test cases, negatively affect their performance (both precise and recall).

In this paper, we propose a novel failure-inducing interaction identification approach (*adaptive*) which aims to alleviate these issues. Our approach is based on the notions of we first give the properties of faulty schemas and healthy schemas. Furthermore, we propose the notion CMINFS(represent for candidate minimal faulty schemas) and CMAXHS(represent for candidate maximal healthy schemas). Through using them we can easily check whether some schemas are healthy or faulty according existed checked schemas. For these schemas can not be checked by exited checked schemas, we generate an newly configuration to test, (do we need to describe it ?????)and the state pass or fail indicate that whether the schema is healthy or faulty. Note that if newly faulty schemas introduced from the extra test configurations, then the identify result is influenced. We take it into account and solve it by introduce the feedback machinery into our approach.

We studied existed works which also target this problem. To propose a clear view of the characteristics of exited works and our work, we summary a list of metrics include constraints, limitations, complexities and so on. Through an comprehensive analysis, we have list the results in our paper. Besides these theoretical metrics, we also conduct empirical studies of these approaches. These experiment objects consists of a group of simulated toy softwares, the Small-Scale and Medium-scale Siemens Software Sets and two large highly-configurable softwares: Apache and MySQL. Our results shows that the failure-inducing schemas do existed in software, and our approach can get the best performance when identify them compared with existed works.

contributions of this paper: 1)we study the failure-inducing schema to analysis its properties. 2)we propose a new approach which can identify the failure-inducing schemas effectively. 3)we classified existed works, and give an comprehensive comparison both on theoretical and empirical metrics. 4)give an general identify framework when facing real failure-inducing problems, and give advices on when an which circumstances to choose which algorithm.

The remainder of this paper is organized as follows: Section 2 introduce some preliminary definitions and propositions. Section 3 presents some limitations of exiting failure-inducing interactions identification approaches which motivate this paper. Section 4 describe our approaches for identify failure-inducing schemas. Section 5 give the comparisons in theoretical metrics. Section 6 describe the experiments design on real subjects. Section 7 summarize the related works. Section 8 conclude this paper and discuss the future works.

2 PRELIMINARY

Before we talk about our approach, we will give some formal definitions and proposals first, which is helpful to understand the background of our description of our approach.

2.1 Definitions

Assume that the SUT (software under test) is influenced by n parameters, and each parameter c_i has

a_i discrete values from the finite set V_i, i.e., $a_i = |V_i|$ (i = 1,2,..n). Some of the definitions and propositions below are originally defined in and .

Definition 1 (test configuration)**.** *A test configuration of the SUT is an array of* n *values, one for each parameter of the SUT, which is denoted as a* n*-tuple (v_1, v_2...v_n), where $v_1 \in V_1$, $v_2 \in V_2$ ... $v_n \in V_n$.*

Definition 2 (test oracle)**.** *the test oracle is that whether it is a test configuration pass or fail. For a clear discuss, we didn't take the multiple output state into account, but we believe our method can easily extend to the multiple oracles.*

However, our discuss is based on the SUT is a deterministic software, i.e., will not pass this time and fail that time. We can run our approach multiple times to eliminate this problem, which, however is beyond the scope of this paper.

Definition 3 (faulty)**.** *by this definition, the constraints can also be a faulty, which didn't run as our expected, and it may have a high privilege than the fault caused by the functional code.*

Definition 4 (schema)**.** *For the SUT, the* n*-tuple (-,v_{n_1},...,v_{n_k},...)is called a* k*-value schema (k ¿ 0) when some k parameters have fixed values and the others can take on their respective allowable values, represented as "-". In effect a test configuration its self is a k-value schema, which k is equal to n. Furthermore, if a test configuration contain a schema, i.e., every fixed value in this schema is also in this test configuration, we say this configuration hit this schema.*

Definition 5 (subsume relationship)**.** *let s_l be a* l*-value schema, s_m be an* m*-value schema for the SUT and $l \leq m$. If all the fixed parameter values in s_l are also in s_m, then s_m* subsumes *s_l. In this case we can also say that s_l is a* subschema *of s_m and s_m is a* parent-schema *of s_l. Additionally, if m = l + 1, then the relationship between s_l and s_m is direct.*

Definition 6 (healthy,faulty and pending)**.** *A* k*-value schema is called a* faulty schema *if all the valid test configuration hit the schema trigger a failure. And a* k*-value schema is called a* healthy schema *when we find at least one passed valid test configuration that hits this schema. In addition, if we don't have enough information of a schema,i.e., we don't sure whether it is a healthy schema or faulty schema, we call it the pending schema.*

Note that, in real software context, there existed cases that a test configuration hit a faulty schema but passed. We call this case the *coincidental correctness*, which may be caused by other factors which influence the execution result. We will not discuss this case in this paper.

Definition 7 (minimal faulty schema)**.** *If a schema is a faulty schema and all its subschemas are healthy schemas, we then call the schema a* minimal faulty schema*(*MINFS *for short).*

Note that this is the target to identify,for that this will give us the most precise while enough information to help the developers to inspect the scope of the source code.

Definition 8 (maximum healthy schema)**.** *If a schema is a healthy schema and all its parent-schemas are faulty schemas, we then call the schema a* maximum healthy schema*(*MAXHS *for short).*

Definition 9 (candidate minimal faulty schema)**.** *If a schema is a faulty schema and satisfy the followed condition: 1.none of its subschemas are faulty schemas, 2. at least one subschema is pending schema.(is need discuss!!!!!! one or none is okay?)We then call the schema a* candidate minimal faulty schema*(*CMINFS *for short).*

Definition 10 (candidate maximum healthy schema)**.** *If a schema is a healthy schema and and satisfy the followed condition: 1.none of its parent-schemas are healthy schemas, 2. at least one subschema is pending schema. We then call the schema a* candidate maximum healthy schema*(*CMAXHS *for short).*

2.2 Propositions

Propositions 1. *All the schemas in a passed test configuration are healthy schemas. All the subschemas of a healthy schemas are healthy schemas.*

Propositions 2. *If schema s_a subsumes schema s_b ,schema s_b subsumes schema s_c, then s_a subsumes s_c.*

Propositions 3. *All the parent-schemas of a faulty schema are faulty schemas.*

Propositions 4. *All the subschemas of a healthy schema are healthy schemas.*

3 Motivation of the approach

3.1 Multiple MFS

3.1.1 Overlapping MFS

3.2 High degree MFS

3.3 Too many test cases

3.4 Needs too many computing resources

3.5 Introducing of newly MFS

A big table that consists of many

4 The approach to identify the minimal failure-inducing schemas

Based on these definitions and propositions, we will describe our approach to identify the failure-inducing schemas in the SUT. To give a better description, we will start give an approach with an assumption , and later we will weak the assumption.

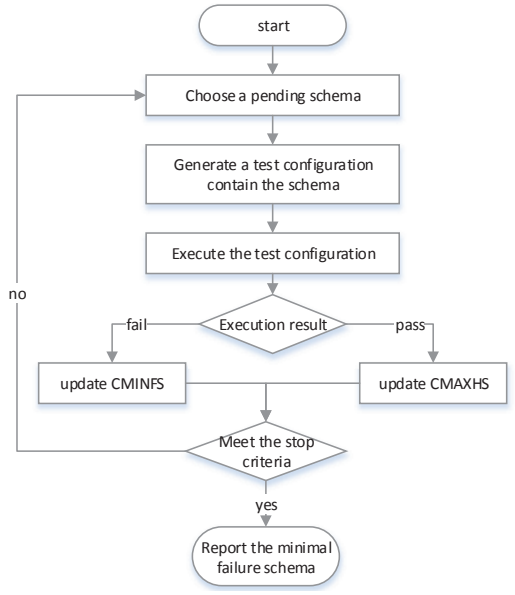

Fig. 1. Overview of approach of identifying MFS

4.1 additional failure-inducing not be introduced

Assumption 1. *Any newly generated test configuration will not introduce additional failure-inducing schemas.*

There are similar assumptions defined in[][], however, it is a strong assumption, which we will change later. And still for this assumption, we can get the followed lemma which can help us to identify whether a pending schema is a healthy schema or a faulty schema.

Lemma 1. *For a pending schema, we generate an extra test configuration that contains this schema. If the extra test configuration passes, then this schema is a healthy schema. If the extra test configuration fails, then the schema is a faulty schema.*

Proof: According to definition of healthy schema, it is obvious that this schema is a healthy schema when the extra test configuration passes.

When the extra configuration fails, this is a faulty schema (or there exists no faulty schema and this test configuration would not fail because the assumption says that this newly generated configuration will not introduce additional faulty schemas). □

4.2 framework to identify the minimal faulty schema

As we can identify a pending schema to be healthy schema or faulty schema by generating newly test configurations, then the approach to identify the minimal faulty schema is clear. Fig.1 shows an overview of our approach. We next discuss each part of the approach in more detail.

4.2.1 Choosing a pending schema

Before we think getting a pending schema, we should assume an general scenario. That is, we have already make sure some schemas to be healthy schemas and some schemas to be faulty schemas, then assume that we still don't meet the stopping criteria, we will choose a pending to check next. But first, we should make sure what schema is pending schema?

In effect, the schemas we can't make sure whether are faulty schemas nor healthy schemas are our wanted. Step further, we sperate this condition into two parts: 1. can't make sure it is faulty schema. As we already know some faulty schemas, then we just make the schema that first not be any one of these faulty schemas and not be the parent-schema of any one of these faulty schemas(As if not, it must be a faulty schema according to the proposition 3). 2 can't make sure it is healthy schema. Similar to the first condition,we already know some healthy schemas, then we just make the schema that first not be any one of these healthy schemas and not be the subschema of any one of these healthy schemas. It is obvious a pending schema must meet both the two conditions.

However, this is not the end of story of checking a pending schema. With the process of identifying, more and more faulty schemas and healthy schemas will be identified. It is not a small number, as it can reach to O(2^n). So both for space and time restriction, we should not record all the faulty schemas and heathy schemas. Then we should find another way to check the pending schema.

To eliminate this problem, we propose an method which can check a pending schema with a small cost. It need to record the CMINFS and CMAXHS all through the process. Instead record all the faulty schemas and healthy schemas, CMINFS and CMAXHS are rare in amount, which can help to largely reduce the space need to record. Then according to the followed two propositions, we can easily check a pending schema.

Propositions 5. *If a schema is neither one of nor the parent-schema of any one of the CMINFS, then we can't make sure whether it is a faulty schema.*

Proof: Take a schema s_a, a CMINFS set S_{cminfs} and a faulty schema S_{fs} set which determined now. It is note that any element $fs_i \in S_{fs}$ must meet that either $fs_i \in S_{cminfs}$ or fs_i be the parent-schema of one of the S_{cminfs}. Assume that s_a is neither the one of nor the parent-schema of any one of the S_{cminfs}. To proof the proposition, we just need to proof that s_a is neither the one of nor the parent-schema of any one of the S_{fs}.

As s_a is neither the one of nor the parent-schema of any one of the S_{cminfs}, so it is not one of S_{fs}. Then we assume s_a is the parent-schema of one of S_{fs}, say, fs_j. As fs_j must meet either $fs_j \in S_{cminfs}$ or fs_j be the parent-schema of one of the S_{cminfs}. So s_a is

the parent-schema of one of S_{cminfs} according to the proposition 2, which is contradict. So the proposition is correct. □

Propositions 6. *If a schema is neither one nor the the subschema of any one of the CMAXHS, then we can't make sure whether it is a healthy schema.*

We ignore this proof as it is very similar to the previous one.

Up to now, we can judge whether a schema is a pending schema, but in effect, there are many pending schemas in a test configurations, especially at the beginning of our process. To choose which one has an impact on our approach. To better illustrate this problem, we consider the followed example:

Assume the failing test configuration: (1 2 1 1 1 2 1 2), that the CMINFS set: (1 2 1 1 - 2 - -) (- 2 - - 1 2 - 2), the CMAXHS set:(- 2 - - - - - -) (- - - - 1 - - -). We list some pending schemas followed (not all, as the number of all the pending schemas is too much that is not suitable to list here):

(1 2 1 - - 2 1 2) (- 2 1 1 - 2 - 2) (1 2 1 1 - - - -) (- 2 1 1 - 2 - -) (1 2 1 - - - - -) (- 2 - 1 - - 2 -) (1 2 - - - - - -) (- - 1 1 - - - -) (1 - - - - - - -) (- - 1 - - - - -).

Choose what is really different. If we choose (1 2 1 - - - - -), assume we check it as a healthy schema. Then all its subschemas, such as (1 2 - - - - - -) (1 - - - - - - -) (- - 1 - - - - -), are healthy schemas. It means that we did not need generate newly test configurations for them. But if we check it as a faulty schema, we can make sure all its parent-schemas are faulty schemas, in this case, they are (1 2 1 1 - - - -) and (1 2 1 - - 2 1 2) need no newly test configurations to test.

Let's look at this problem from another angle. Take the schema as a integer number, these parent-schemas of a schema is like the integer numbers bigger then this number, and these subschemas of a schemas is like the numbers smaller than this number. Take a float number as the metric, then, we can describe the faulty schema as the number bigger than this metric, and healthy schema as number smaller than this metric. So the minimal faulty schema is just the number most approximate the metric and bigger than the metric.

It seems like a search problem scenario. Then can we directly apply the efficiently algorithm binary-search? the answer is no, because there are schemas that neither parent-schema or subschema relationship, such as(1 2 1 - - 2 1 2) (- 2 1 1 - 2 - 2). So to utilize the binary search technique, we should make some change. First, should give the followed definitions:

Definition 11 (chain)**.** *A chain is an ordered sequences of schemas in which every schemas is the direct parent-schema of the schema that follows. Moreover,if all the schemas in a chain are pending schemas, we call the chain a pending chain.*

This definition is similar to the path in []

As all the schemas in a chain are have relationships, then we can apply binary search technique. As we all know, the longer the pending chain, the better performance binary search technique will get. So we should choose a pending chain as longer as possible each iteration.

To get a longest chain, we need to ensure that the head schema of this chain do not have any parent-schema which is a pending schema(called up pending schema), and the tail schema do not't have any subschema which is a pending schema (called down pending schema). The algorithm that get the up pending schemas and down pending schemas are list in algorithm 1 and algorithm 2.

As showed in algorithm 1, we start from the failing test configuration $\mathcal{T}$, assign it to the *rootSchema*(line 1)and then add it to a *lists*(line 3). We then do some operation (line 4 -line 12)to this lists and at last get these pending schemas in lists as up pending schemas.(line 13) This operation consists of two iteration:

1. successively get one *CMINFS* in $\mathcal{S}_{\mathcal{CMINFS}}$. (line 4). Define a temple value *nextLists* which initialize an empty set(line 5).We then execute the second iteration. After that, we will eliminate these same schemas list in *nextLists* (line 10)and then assign to *lists* (line 11).

2. Successively get one schema in *lists*(line 6). And then mutant it according to the *CMINFS* to a set of schemas(line 7). Add them to the *nextLists* (line 8).

The mutant procedure for a schema is just remove one value in it which this value is also in *CMINFS*. This procedure will result in *k* mutant schemas if the *CMINFS* is a *k*-value schema. By doing this, any mutant schema will not be the parent-schema of the corresponding *CMINFS*.

After this two iteration, the schemas in the *lists* will not be the parent-schema of any *CMINFS* in the $\mathcal{S}_{\mathcal{CMINFS}}$.

Fig.2 gives an example to the algorithm 1.

Algorithm 2 is a bit different from algorithm 1. As our target is to get the minimal value pending schema, we get started from a 0-value schema(line 1). Then to make schemas not to be the subschema of any *CMAXHS* in $\mathcal{S}_{\mathcal{CMAXHS}}$, we should let the schemas must contain at least one value that is not in the corresponding *CMAXHS*. So the strategy is to get a reverse schema of the *CMAXHS* which this schema consists of all these values in the failed configuration except these are in *CMAXHS*(line 5). And mutant the schema by adding one value in the reversed schema to make the schma not to be the subschema of the correspinding *CMAXHS*(line 8). After two iteration similar to Algorithm 1, we will get all the schemas that meet that not to be subschema of any *CMAXHS* in $\mathcal{S}_{\mathcal{CMAXHS}}$ from which we choose these pending schemas as down pending schemas(line 14).

Fig.3 gives an example to the algorithm 2. we can see.

Algorithm 1 getting up pending schema

Input: $\mathcal{T}$ ▷ failing test configuration
$\mathcal{S}_{\mathcal{CMINFS}}$ ▷ set of CMINFS
$\mathcal{S}_{\mathcal{CMAXHS}}$ ▷ set of CMAXHS
Output: $\mathcal{S}_{\mathcal{UPS}}$ ▷ the set of up pending schemas
▷ %comment: initialize%

1: $rootSchema \leftarrow \mathcal{T}$
2: $lists \leftarrow \emptyset$
3: $lists \cup \{rootSchema\}$
4: **for each** $CMINFS$ in $\mathcal{S}_{\mathcal{CMINFS}}$ **do**
5: $nextLists \leftarrow \emptyset$
6: **for each** $schema$ in $lists$ **do**
7: $S_{candidate} \leftarrow mutant_r(schema, CMINFS)$
8: $nextLists \leftarrow nextLists \cup S_{candidate}$
9: **end for**
10: $compress(nextLists)$
11: $lists \leftarrow nextLists$
12: **end for**
13: $\mathcal{S}_{\mathcal{UPS}} \leftarrow \{s | s \in lists \wedge s\ is\ pending\}$

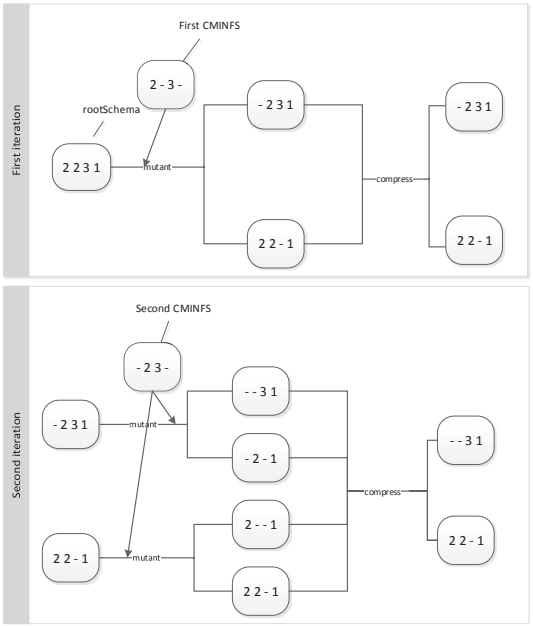

Fig. 2. example of getting up pending schemas

Algorithm 2 getting down pending schema

Input: $\mathcal{T}$ ▷ failing test configuration
$\mathcal{S}_{\mathcal{CMINFS}}$ ▷ set of CMINFS
$\mathcal{S}_{\mathcal{CMAXHS}}$ ▷ set of CMAXHS
Output: $\mathcal{S}_{\mathcal{DOWNS}}$ ▷ the set of down pending schemas
▷ %comment: initialize%

1: $initschema \leftarrow ()$
2: $lists \leftarrow \emptyset$
3: $lists \cup \{initschema\}$
4: **for each** $CMAXHS$ in $\mathcal{S}_{\mathcal{CMAXHS}}$ **do**
5: $reverse \leftarrow\ reverse(CMAXHS)$
6: $nextLists \leftarrow \emptyset$
7: **for each** $schema$ in $lists$ **do**
8: $S_{candidate} \leftarrow mutant_a(schema, reverse)$
9: $nextLists \leftarrow nextLists \cup S_{candidate}$
10: **end for**
11: $compress(nextLists)$
12: $lists \leftarrow nextLists$
13: **end for**
14: $\mathcal{S}_{\mathcal{DOWNS}} \leftarrow \{s | s \in lists \wedge s\ is\ pending\}$

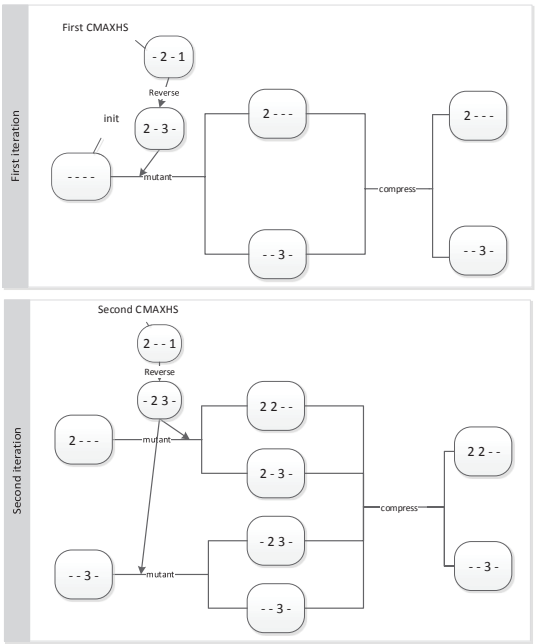

Fig. 3. example of getting down pending schemas

After we can get the up pending schemas and down pending schemas, then the algorithm of getting the longest chain can be very simple,which is list in Algorithm 3. In this algorithm we can see that we just search through the up pending schemas and down pending schemas(line 7 - 8) to find the two schemas which has the maximum distance(line 9 - 13). The distance of a k-value schema and a l-value schema (k¿l) is defined as:

$$distance(S_k, S_l) = \begin{cases} -1, & S_k\ is\ not\ parent - schema\ of\ S_l \\ k - l, & otherwise \end{cases}$$

Then we just use *makechain* procedure to generate the longest chain. The *makechain* procedure is very simple, it just repeat adding one schema by keeping all the value in down pending schema and removing one factor of the previous schema.

The last step of getting the schema is just choose the schema from the longest chain. To clearly describe the approach, we discuss it in the overall identifying algorithm which is list in the Algorithm 4. As we have talked previously, we first judge the end criteria, if meet we will report the minimal faulty result. Otherwise we will do the loops. In the loop, we

Algorithm 3 Finding the longest pending schema

Input: $\mathcal{T}$ ▷ failing test configuration
$\mathcal{S}_{\mathcal{CMINFS}}$ ▷ set of CMINFS
$\mathcal{S}_{\mathcal{CMAXHS}}$ ▷ set of CMAXHS
Output: $\mathcal{CHAIN}$ ▷ the chain

$\mathcal{S}_{\mathcal{UPS}} \leftarrow getUPS(\mathcal{T}, \mathcal{S}_{\mathcal{CMINFS}}, \mathcal{S}_{\mathcal{CMAXHS}})$
$\mathcal{S}_{\mathcal{DOWNS}} \leftarrow getDOWNS(\mathcal{T}, \mathcal{S}_{\mathcal{CMINFS}}, \mathcal{S}_{\mathcal{CMAXHS}})$
$max \leftarrow 0$
$head \leftarrow NULL$
$tail \leftarrow NULL$
$chain \leftarrow NULL$
for each up in S_{UPS} **do**
  for each $down$ in S_{DOWNS} **do**
    if $distance(up, down) \geq max$ **then**
      $max \leftarrow distance(up, down)$
      $head \leftarrow up$
      $tail \leftarrow down$
    end if
  end for
end for
$\mathcal{CHAIN} \leftarrow makechain(head, tail)$

first judge if it is the beginning or headIndex greater than tailIndx, then we will update the CMINFS and CMAXHS, followed generated the longest chain and initial the headIndex, tailIndex and middleIndx. If not we will get the middleIndex indicate the half. Then will select the schema with index of middleindex in the longest chain. And then generate a test configration and exectute the SUT under it. If the result passed, we let tail = middle - 1 else head = middle + 1. By doing this and the previous set the middleIndex we can apply the binary searh to the indentying. It is noted that we intitally let middle = 0. which we want know if this chain has faulty schema as soon as possible.

4.2.2 generate a new test configuration

To generate a new test configuration to test the schema. The generated test configuration must meet the followed rules:

1. must contain the selected schema.
2. must not
3. constraint(system-wide constraint and test case specific constraint, i.e., masking effect)

The first one is easy to meet. We just keep the same value which are in the schema are also in the test configuration. We must meet the second condition for that if we contain another one, combine this then this test configuration will contain an parent-schema of this selected schema in the original test configuration, and it will confuse us whether it is indicate this schema or its parent-schemas dedicate this result. To fulfil this condition, we need to choose other available values in the SUT which are different from the original test configuration. the third one is that we should consider the constraints

Algorithm 4 identify process

Input: $\mathcal{T}$ ▷ failing test configuration
$\mathcal{S}_{\mathcal{CMINFS}}$ ▷ set of CMINFS
$\mathcal{S}_{\mathcal{CMAXHS}}$ ▷ set of CMAXHS

while *hasn't meet the end criteria* **do**
  if *the beginning* or $headIndex > tailIndex$ **then**
    $update(\mathcal{S}_{\mathcal{CMINFS}}, \mathcal{S}_{\mathcal{CMAXHS}})$
    $longeset \leftarrow getLongest(\mathcal{T}, \mathcal{S}_{\mathcal{CMINFS}}, \mathcal{S}_{\mathcal{CMAXHS}})$
    $headIndex \leftarrow 0$
    $tailIndex \leftarrow length(longest) - 1$
    $middleIndex \leftarrow 0$
  else
    $middleIndex \leftarrow \frac{1}{2} \times (tailIndex + headIndex)$
  end if
  $\mathcal{SCHEMA} \leftarrow longest[middleIndex]$
  generate a extra test configration $\mathcal{T}'$ *contain* $\mathcal{SCHEMA}$
  execute SUT under $\mathcal{T}'$
  if *the test configration passed* **then**
    $tailIndex \leftarrow middleIndex - 1$
  else
    $headIndex \leftarrow middleIndex + 1$
  end if
end while
report the minimal faulty schemas

4.2.3 execute SUT under the test configuration

In real software testing scenario, when we test a SUT under a test configuration, there may be many possible testing state: such as pass the testing assertion, don't pass the testing the assertion but with different failure type, can't complete the testing. To get a clear discussion, in this paper we just use *pass* represent the state that pass the testing assertion and *fail* represent all the remained state.

4.2.4 update information

The update information is followed when the current chain is checked over. Then before we generate another longest chain, we should update the CMINFS and CMAXHS set. In fact, we just need the CMINFS and CMAXHS in the longest chain.

4.2.5 stop criteria

The stop criteria is clear, our algorithm stops when there are no pending schemas left, for that when we once can checked all the schemas of a test configuration, we can get the minimal faulty schemas, which is the target of our algorithm. And whether there are pending schemas can be easily checked by that if we can't generate a longest chain (the length must greater than 1), there must be no pending schemas.

4.2.6 report the reuslt

In fact, the last in the CMINFS lists is must be the minimal faulty schemas. For that if these schema

in the CMINFS is not minimal faulty schema, then there must be some pending schemas. However, the algorithm stop when there are no pending schemas remained. So at last these in the CMINFS must be the minimal faulty schemas.

4.2.7 new ideas

you should first identify a failure-inducing schema, and then find others. initial the find tuples, and then using the same algorithms as others. when identify is confirm , then choosing new longest path.

4.3 example

We will give an complete example listed in Table 2. Assume that a SUT is . constaints.

4.4 Without Safe Values Assumption

Up to now our algorithm is based on the assumption that additional generated test configuration does not introduce newly faulty schema. We will give an augment algorithm this section to eliminate this assumption. Our augment algorithm is inspired by the feedback machinery in the controlling system, which in high level we will validate the identify result at the end of the aforementioned algorithm, and if the schema is validated as a minimal faulty schema, we will end the algorithm, otherwise we will repeat the algorithm again to adjust the result. The detail of our augment algorithm is list in Algorithm 5. We can find the first part of augment algorithm is an unlimited loop, in the loop we first use previous identify algorithm to identify the MFS, and then we will check the result, the check process is just to addtional generate another test configurations and execute. When we find the MFS is not right, which means our process introduce newly MFS, then we first label this MFS as an healthy schema, and then update the CMAXHS. After check, if we find at least one MFS is not right, we then empty the CMINFS, and then readd these right identified MFS to this set, and then reprocess this process untill all the MFS is validated as right. The second part of our algorithm is just we look through the generated test cases one by one, if it failed, and did not contain the MFS we identified in the first part, which means it introduce newly faulty schema, and we will rerun this process to find these introduced MFS.

Table 3 illustrate this augment algorithm with an example.

Algorithm 5 augment identify process

Input: $\mathcal{T}$ ▷ failing test configuration
$\mathcal{S}_{\mathcal{CMINFS}}$ ▷ set of CMINFS
$\mathcal{S}_{\mathcal{CMAXHS}}$ ▷ set of CMAXHS

while *true* **do**
  $\mathcal{S}_{\mathcal{MFS}} = Identify\_process(\mathcal{T})$
  for each MFS in $\mathcal{S}_{\mathcal{MFS}}$ **do**
    if $validate(MFS) = fail$ **then**
      $updateHealthySchemas(S_{CMAXHS}, MFS)$
    end if
  end for
  if *at least one* MFS *is not correct* **then**
    $empty(S_{CMINFS})$
    add all the vaildated MFS *in the*$\mathcal{S}_{\mathcal{CMINFS}}$
  else
    break
  end if
end while
for each $testCase$ in $\mathcal{EXTRATESTCASES}$ **do**
  if $testCase$ *introduce newly* MFS **then**
    $augment\_identify\_process(testCase)$
  end if
end for
report the minimal faulty schemas

5 Evaluation with simulated SUT

In this section, we designed a simulated SUT of which the number of parameters and the value of each parameter can both be customized. In addition, we can also manually inject faulty schemas in the simulated SUT. We will conduct a series of experiments with this simulated SUT in this section. The goal of our experiments is to evaluate the efficiency and effectiveness of our approach compared with other existed techniques. The main reason why we use simulated program is that we can easily run a bench of experiments with various states of a SUT, i.e., different parameters and different MFSs. By doing so we can thoroughly learn the performance of each algorithm without biases.

As we discussed in the background section, the fault privilege properties just effect the way we determine a configuration is fail or pass. It doesn't effect the performance of each algorithm. So to be simple and clear, we omit the fault privilege in the simulated SUT , i.e., all the fault in the SUT has the same privilege. This is a ideal scenario which may not exist in real softwares, in the empirical studies of section 7 we will deal with the real faults in some open-source softwares with different privileges.

5.1 comparison algorithms

There are several algorithms aim to identify the MFS, they can be classified as non-adaptive methods and adaptive methods. The first set of methods do not need additional test configurations and can identify the MFSs when given an executed test configurations while the second one will generate some more additional test configurations to facilitate the process of identifying MFSs in a failing test configuration. Our algorithm is belong to the second part. To make the comparison clear and fair, we just choose the

TABLE 2
An example of identifying

$\mathcal{SCMINFS}$	$\mathcal{SCMAXHS}$	Longest Chain	Choosing Schema	Generating Test Configuration	Execution result
1	1	1	1	1	false
1	1	1	1	1	pass
1	1	1	1	1	false

TABLE 3
An example of augment identifying

$\mathcal{SCMINFS}$	$\mathcal{SCMAXHS}$	Longest Chain	Choosing Schema	Generating Test Configuration	Execution result
1	1	1	1	1	false
1	1	1	1	1	pass
1	1	1	1	1	false

algorithms which all belong to the second part as the comparison subject, i.e., adaptive methods. These algorithms is listed as follows:

(1)OFOT, proposed by Nie[], it change one factor of a test configuration one time to generate a newly configuration and then analysis the MFSs according to the executed result of these generated configurations. (2)FIC-BS, proposed by Zhang, using a delta debugging strategy to isolate the MFSs (3) LG, proposed by Martine[], use a locatable graph to represent the faulty schema and adopt divide and conquer strategy to find the MFSs. (4) SP[], proposed by Ghandehari, they generate configurations for these mostly suspicious schemas and give a rank of these schemas at last based on the suspicious "degree" of them (5) CTA, proposed by Shakya[], it combine the OFOT strategy and the classified tree technique which first applied in characterize the MFS in [] to analysis the MFSs. (6)RI, proposed by Li[], it is also using delta debugging, but the factors needed to change in a iteration are different from FIC-BS at some conditions.(7)IterAIFL, proposed by Wang[], a mutant based on OFOT, which changes different number of factors instead of one each iteration for a test configuration. (8)TRT and TRT-NA, two approaches proposed in our previous work[], using a structured tree model to analysis the MFSs in a failing configurations. More details of these algorithms will be discussed in the related works.

Some parameters of these algorithm are assigned: for SP, the degree of MFSs we set is 2(except for the last group of experiment, which is 4 instead), for CTA,the confidence factor of classified tree algorithm is 0.25 as same as [].

5.2 evaluation metrics

There are three facts we care about each algorithm: 1)How many additional test configurations do an approach need to generate? 2)How many MFSs can an approach identify for a SUT? 3)Does all the schema identified by an approach are correctly? We will define two formula to represent the second and third facts respectively, i.e., recall and precise:

$$recall = \frac{correctly\ Identiied\ MFS}{all\ the\ MFS\ in\ SUT}$$

$$precise = \frac{correctly\ Identified\ MFS}{all\ the\ identified\ MFS}$$

The recall is percentage of the correctly identified MFSs of an algorithm of all the MFSs in the SUT. The bigger the metric the better an approach can perform on finding MFSs in a SUT. There are some factors may influence this metric of an algorithm, mainly are whether an approach consider a failing configurations contain multiple MFSs, whether consider if there exist overlapped MFSs in a failing configurations and whether consider the newly introduced MFSs of the additional generate configurations. The precise is the percentage of the number of correctly identified MFSs of algorithm of all the number of MFSs this algorithm identified. This metric measure the accurately of an approach and the bigger it is the better the approach performs. This metric is related to whether a algorithm consider the influence if an generated test configuration contain newly MFSs. If the approach did not consider so, it may make a wrong judgement to determine some schemas in the original configuration to be faulty schemas which in fact are not.

Of course these factors we mentioned are just affect these the no-machine learning algorithm, as for the machine learning algorithms,i.e., CTA, the factor that matters is the classified algorithms itself.

5.3 experiment setups

We will carry out the followed five experiment in this section:

1)For the first one, we will give a set of SUTs with 8 parameters, each parameter has 3 values, all of them only have one MFS, the difference between each SUT in this set is the MFS we inject in it. All the MFSs has

the degree of 2. There are $\binom{8}{2} = 28$ possible MFSs we can inject, so the number of set of SUTs is 28. For each SUT in this set, we will feed each algorithm the same failing configurations contain the MFS we inject, and then let these algorithm identify the MFS in it. We will record the additional configurations each algorithm needed. As the first experiment is simple, so all the approaches can identify the MFS with precise 1 and recall 1. This experiment will give us a initial view of the cost of each algorithm on identifying the MFS in a failing configuration.

2)The second experiment is similar to the first one, except that we will inject double different MFSs with degree 2 in each SUT. It is easily compute their are $\binom{28}{2} = 278$ possible SUTs in this experiment. For each algorithm, we also feed them with the same failing configuration contain the two MFSs and then let them identify them. Another difference from the experiment 1 is that in this experiment we will record the "recall" and "precise" of each algorithm, as these metric is not equal to 1 for all the algorithms like experiment 1. This experiment mainly focus on observing the performance of each algorithms in facing multiple MFSs in a configuration.

3)The third experiment aims at learning the capability that each algorithm handling the introducing newly MFSs. To accomplish this goal, We firstly inject two MFSs in a SUT, and then feed each algorithm a failing configuration which contain only one MFS, remaining another MFS not in it. To increase the possibility of introducing newly MFSs for each algorithm, we set the second MFS with degree 1, which may be easily introduced by just changing one factor of the original failing configuration. The first MFS in the original configuration has degree 2. So there are $\binom{8}{2} \times \binom{8}{1} = 224$ possible SUTs in this experiment. The same as experiment 2, we will record the "recall" and "precise" of each algorithm.

4)For the fourth experiment, we will vary the number of parameters of SUT. In specific we will take 8, 9, 10, 12, 15, 20, 30, 40, 60, 80, 100, 120 respectively as the number of parameters of SUT. For each number of parameters, say n, we will generate $\binom{n}{2}$ SUTs with different MFSs injected. And for these SUT, we will do the same process as experiment 1. At last we will record the average number of additional configurations each algorithm needed for each number of parameters. The goral of this experiment is to see the influence of the number of parameters on each approach.

5)In the fifth experiment we will fix the number of parameters of SUT to be 8, but vary the number of degrees of the MFS we inject in the SUT. That is, we will inject MFS in SUT with degrees 1,2,3,4,5,6,7,8 respectively, for a degree of m (m = 1,2,3,4,5,6,7,8), we will generate $\binom{8}{m}$ SUTs with different m-degree MFS injected. And then we will do the same thing as experiment 4 , i.e., record the average number of additional configurations each algorithm needed for each number of degree. This experiment is order to inspect whether these algorithm can handle different degree of MFS in a failing configuration.

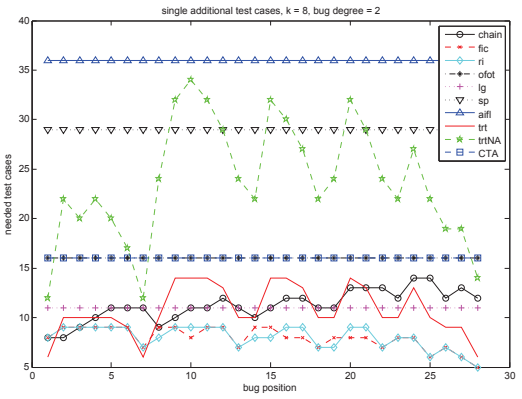

Fig. 4. k = 8, t = 2,single, additional test suites

5.4 results and discuss

We will discuss the results into five groups according to the experiment set up.

1)The result of the experiment 1 is depicted in figure 4. The y-axis report the number of configurations and the x-axis represent a SUT. Algorithms differs from each other in the signal of point and the line linking these points. For a particular algorithm, a point indicate the number of configurations corresponding to the y-axis needed for identifying the MFS in the SUT which corresponding to the x-axis of this point.

There are some intuitive information we can learn from this figure. First, the configurations needed of some algorithms didn't change along with the change of SUT. These algorithms are AIFL, CTA, OFOT,LG, while others need different configurations according to the SUT they are facing. Second, some algorithms(sp, aifl, trtNA, OFOT, CTA) shows a obvious disparity in this figure, we can easily rank them according to the number of configurations, i.e., $ofot = cta < trtNA < sp < aifl$. Besides, These algorithms needs more configurations than the remaining algorithms.

For the remaining algorithms, their results are tangled with each other. So in order to rank them, we give an average configurations each algorithms needed : Chain 9.25, FeedBack 11.25, fic 7.96, ri 8 ofot 16 lg 11 , sp .29, aifl 36, trt 10.75 trt-NA 23.75 CTA 16. Overall, we can conclude an initial rank in the number of configurations each algorithm needed as: $FIC{-}BS < RI < LG < TRT < CHAIN < OFOT = CTA < TRT - NA < AIFL < SP$.

2)Figure 5 list the result of experiment 2. This figure doesn't show the number of configurations of each algorithms,instead it just show the average precise and recall for each algorithms as not all these algorithms can both get recall 1 and precise 1 as experiment 1. It

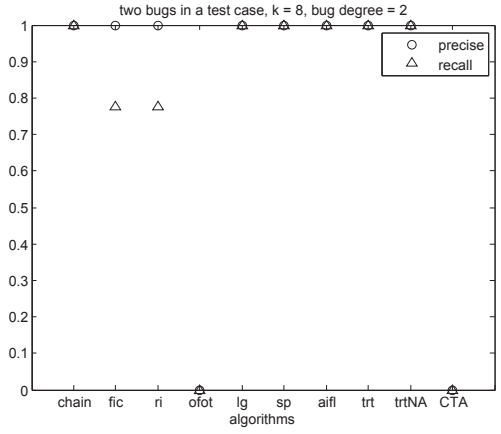

Fig. 5. k = 8, t = 2,double, additional test suites

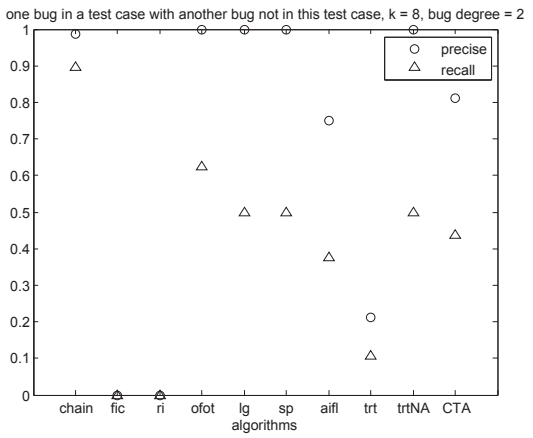

Fig. 6. k = 8, t = 2,import, additional test suites

is obvious meaningless if we compare two algorithms that one algorithm can reach recall 1 and precise 1 while the other can't. So we will only list the average number configurations of these algorithms that can reach recall 1 and precise 1 later.

From this figure, we can find the following algorithms: chain, lg, sp, aifl, trt and trt-na can both get the precise and recall 1, which means that they can perfectly deal with the multiple MFSs in a failing configuration. As the opposite side, algorithms OFOT and CTA both get the recall 0 and precise 0. This is a signal that the two algorithms can't handle the condition that a failing configuration contain multiple MFSs. So for these two algorithms, they may need other techniques to assist them in facing such conditions. As for the remaining algorithms RI and FIC, they got precise 1 , but didn't reach recall 1, this is because they can't handle the condition if two MFSs have overlapped part. When they facing such condition, they can only identify one MFS of them.

The average number of configurations of these algorithms can both get recall and precise 1 is as follows:

we can find that our approach can also get a no-bad result among them

3)The third experiment's result is shown in fig 6. Similar to experiment 2, this figure just record the recall and precise of each algorithm. We can learn several facts of this result.

First, no algorithm can both get recall 1 and precise 1 in this condition. Thus shows that no algorithm can perfectly handle the importing problem.

Second, even though, there are some algorithm(OFOT, LG, SP, TRT-NA, our approach is almost get precise 1) can get precise 1, this is a signal that the importing problem has a little impact on them

Third, our approach get the highest score in recall than others, the specific rank are $Chain > OFOT > LG = SP = TRT - NA > CTA > AIFL > TRT > FIC = RI$. It shows that our approach can find more MFSs than others when just feeding a failing configuration.

Fourth, algorithms fic and ri really suffers from this condition, as they get both recall and precise 0.

4)The data depicted in figure 7 shows that the average configurations needed of each algorithm for experiment 4. In this figure, the y-axis still reports the number of configurations while the x-axis represent a set of SUTs with the same number of configurations. These number of parameters are 8, 9, 10, 12, 15, 20, 30, 40, 60, 80, 100 and 120 respectively. And a point in this figure indicate the average number of configurations needed for the corresponding algorithm to identifying the MFS in these SUT with the same number of the parameters. For example, for the point whose x-axis is 8 corresponding to the algorithm "AIFL", we record the number of configurations needed to identify each SUT from the SUT with 8 parameters (totally $\binom{8}{2} = 28$ SUTs) and compute the average value of them, i.e., "".

We can easily learn that AIFL needs the largest number of configurations to identify MFS. Worse more, it need such big computing resource along with increase of k so that we even can't make it to identify the MFS in the SUT with number of parameters bigger than 12. The second largest is the algorithm "SP". Although it needs much smaller computing resource than "AIFL", we yet can't tolerate the time it need to identify the MFS with number of parameter bigger than 40.

As the average number of configurations of "SP" increase quickly with the increase of k. It make the results of the remaining algorithms show unclearly in this figure. To get a better view of the remaining algorithms, we enlarge the perspective for these algorithms in figure 8.

From this perspective, we first find that our previous approach TRT and TRT-NA also suffer from the large k. We can't let them identify the MFS with k bigger than 12. Apart form this two algorithms, all the remaining algorithms: chain, fic, lg, ri can complete this experiment. second, it also give us a

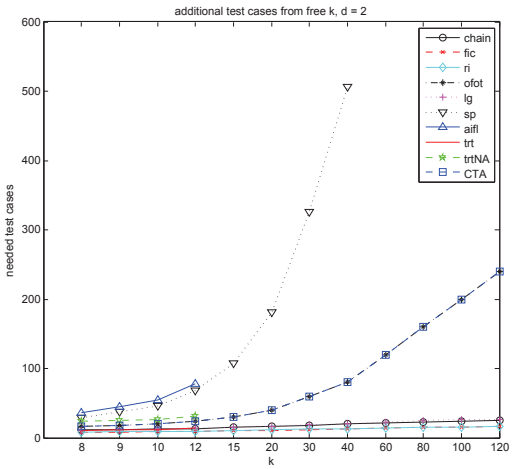

Fig. 7. k is free, t = 2,single, additional test suites

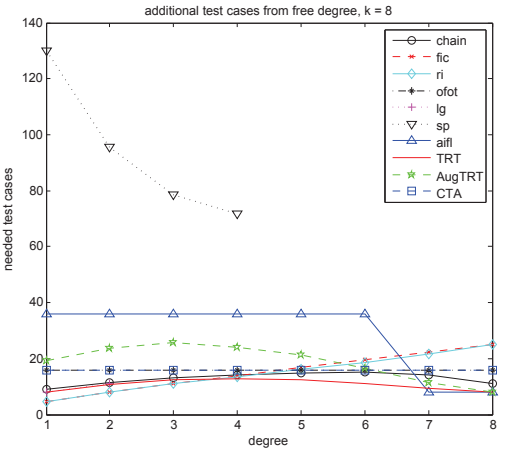

Fig. 9. k = 8, t is free,single, additional test suites

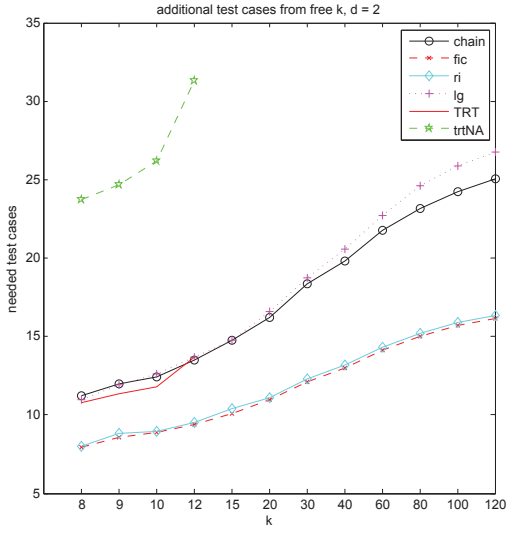

Fig. 8. k is free, t = 2,single, additional test suites for some algorithms

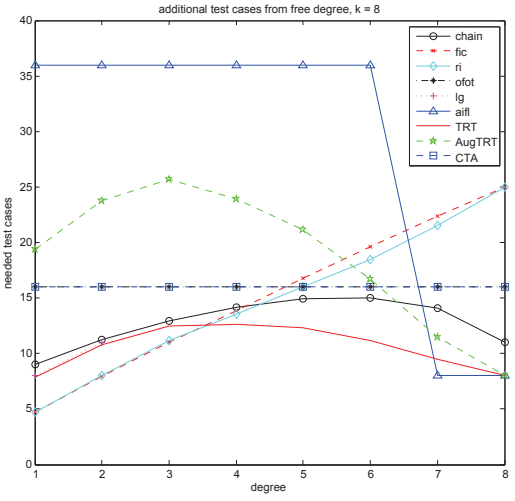

Fig. 10. k = 8, t is free,single, additional test suites for some algorithms

intuitive rank of each algorithms according to the cost of configurations generated, i.e., $TRT - NA > LG > CHAIN > TRT > RI > FIC$. Combine With the two algorithms "AIFL" and "SP" we mentioned before, the final rank is $AIFL > SP > TRT - NA > LG > CHAIN > TRT > RI > FIC$.

5)Figure 9 gives us the result of the last experiment. This figure is organised similarly as figure 7, except the x-axis represents a set of SUTs have the same degree of MFS we inject in it. For a particular algorithm, a point in this figure shows the average number of configurations needed to identify the MFS in these SUTs with same degree MFS injected(there are $\binom{8}{m}$ SUTs for the degree m). Note that int this experiment we set the degree that SP can be identified 4 to distinguish with algorithm LG which can just identify the MFS with degree not bigger than 2.

From this figure, we can find that as we set parameter of SP to be 4, the number of configurations SP needed to identify the MFS in a failing configuration increase a lot than before. And as we expected, SP can just identify the MFS with degree not than 4. Apart this algorithm we enlarge the view of the remaining algorithm in figure 10 as experiment 4. From this clearer view, we can find several useful information: First, the LG method can only indentify the MFS with degree not than 2 as expected. Second, we can find the number of configurations of ri and fic increase quickly along with the degree increase. Third, CTA and CTA need the same configurations 16 regardless of what the degree is. Fourth, though AIFL perform worst in most cases, when d is 7 and 8 it can do better than other techniques. Fifth, the number of configurations needed of TRT-NA, TRT and our chain techniques first increase then decrease along with increase of d.

we will enlarge some of them to figure 10

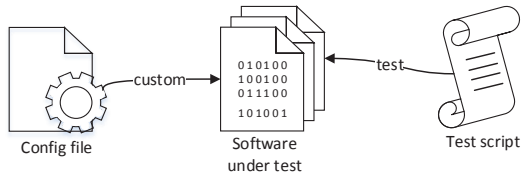

Fig. 11. proceeding

6 EMPIRICAL CASE STUDY

While we got know that our approach have a better performance than others in several scenarios such as a failing test case contains multiple MFSs, the generated extra test case introduces newly MFS and so on from previous simulated experiments, it did not give us strong confidence in that our approach will also perform well in real software systems. One important reason of this is that we don't know whether these competitive scenarios for our approach existed in real software systems. So to eliminate the doubt we conduct a series empirical studies in this section. These empirical studies aimed to answer the following questions:

Q1:Is there any test case that contain multiple MFSs, and if so, do they overlapped each other? Do any of these MFSs have high degree and how likely does it introduce newly MFS when generate extra test cases.

Q2:How well of these approaches mentioned in previous sections performed when identifying the MFS in the real softwares?

Q3:If we combine the result of one approach with another one, does it give us a better result than both of them?

The subject systems for these studies are HSQLDB(2.0rc8). HSQLDB is a database management system written in pure java.All of them have millions of uncommented lines of code. And they all share the same proceedings depicted in Fig.11 when tested in the following studies.

As see from the figure, there are three modules in this proceeding: configuration file, software under test and the test script. The software under test is a set of software components which supply some interfaces, with which one can invoke the serving functions of the software. The configuration file is the file that can custom the properties of the software, and the test script is a executable file which test some functions of the software. In specific, the proceeding is trying different configuration of the software by changing the content of the configuration file, and then executing the same test script to observe the result.

6.1 experimental setup

Before we use these programs for study, we will take the following steps for each program to obtain the basic information of them, 1)Build the input configuration model of the subject program 2)execute the test case under each possible configuration and record their executed result. 3)list all the types of fault during testing and judge their priority.

We will depict these processes and results for each subject program in detail.

TABLE 4
input model of HSQLDB

SQL properties(TRUE/FALSE)	
sql.enforce_strict_size, sql.enforce_names,sql.enforce_refs, sql.enforce_size, sql.enforce_types, sql.enforce_tdc_delete, sql.enforce_tdc_update	
table properties	**values**
hsqldb.default_table_type	CACHED, MEMORY
hsqldb.tx	LOCKS, MVLOCKS, MVCC
hsqldb.tx_level	read_commited, SERIALIZABLE
hsqldb.tx_level	read_commited, SERIALIZABLE
Server properties	**values**
Server Type	SERVER, WEBSERVER, INPROCESS
existed form	MEM, FILE
Result Set properties	**values**
resultSetTypes	TYPE_FORWARD_ONLY,TYPE_SCROLL_INSENS TYPE_SCROLL_SENSITIVE
resultSetConcurrencys	CONCUR_READ_ONLY,CONCUR_UPDATABLE
resultSetHoldabilitys	HOLD_CURSORS_OVER_COMMIT, CLOSE_CURSORS_AT_COMMIT
option in test script	**values**
StatementType	STATEMENT, PREPAREDSTATEMENT

6.1.1 HSQLDB

Through searching the developers' community of HSQLDB in sourceforge, we found a buggy version with its faulty description in [], and a test script is attached which can reproduce the bug. The original test script only considered one option which have two values, remaining the other options set default values. To see what will happen when executing the test script under more different configurations, we need add some other options which may influence the behaviour of the database management software. From numerous optioins of HSQLDB, We chose some of the options that control the properties of table, some control the sql, some control the server and some control the result set. Along with the one in the original test script, we derive a input configuration model of HSQLDBlisted in table 4.

We denoted this input model as $(2^{13}, 3^3)$. There are $2^{13} * 3^3 = 221184$ possible configurations in total. We executed the test script under each of the 221184 configurations and recorded their result. In specific, there are 147472 configurations triggered exceptions, while others passed this test. These exceptions can be classified to 4 types according to the exception traces. Through viewing the exception trace info and the source code of test script, we sorted out the priorities of these exceptions. Table 5 shows the detail of the

TABLE 5
faulty and Priority

exception ID	lower priority	num of configs
1	2	36855
2	1	110558
3	1 2	58
4	1 2	1

4 types of exceptions. The "exception ID" column shows the id of the exception. The "lower priority "column gives the IDs of exception which have lower priority than this exception. The "num of configs" means the number of test configurations under which the test script triggered this type of exception when executing.

6.2 Study 1: what the schemas like in practice

In the first study, we aimed to answer the Q1. We need to investigate the MFSs of the real softwares to see 1) Do there existed multiple MFSs in the same configuration? 2) if so, how many of them overlapped each other? 3) Do any of the MFSs have the degree larger than 2. 4) What the possibility of introducing newly MFSs when generating extra configurations. In particular we will inspect the MFSs in HSQLDB.

The first obstacle of case study 1 is that we don't know the MFSs in the real softwares. The original bug report page of these softwares[][] can give us hints of the MFSs of the software, but it is not enough for us to get accurate MFSs of the software. The reason is that we had added more options to test the script which resulted in more exceptions than reported and the original reported exception can also be related to more options that didn't be mentioned in the bug report.

So the only way to recognize the MFSs in the software is searching all the schemas in a configuration one by one, and then judging the state of the schema according to the definition and propositions. We will repeat the process in all the possible configurations of the software, this process is time-assuming which can be optimized by introducing hash table and adjusting the order when searching the schema of a configuration. We will omit these details as it is not point of this paper. Lastly after this time-assuming process we got the accurate MFSs for each software, we recorded them for later use.

6.2.1 Measure the MFSs in the softwares

As MFSs in the softwares are figured out, we can easily count the number of configurations contain multiples MFSs and MFSs that overlapped each other, as well as count the number of MFSs that have a degree larger than 2. But for the possibility of introducing newly MFSs when generating extra configuration (brief as possibility of introducing), we yet can't measure it for we didn't define how to compute the "possibility". We will give a formula to compute the possibility of introducing next, before then we will explain how is this formula derived.

First we should understand under which condition will the event of introducing newly MFSs harms our identifying algorithm. Consider the following scenario.

For a test case of which we want to identify the MFS, say test case A, if we change one factor of it to generate a new test case, say test Case B. Then assume in this step we introduced a new failure-inducing schema, but meanwhile we didn't break the original failure-inducing schema in the test case A when we generate B, in other words, the MFS in A is still in B.

At this situation it will not influence our identifying result for that if we did not introduce the schema, our result is the same—trigger the same failure.

Consider another scenario, still for test case A, and we also change one factor of it to generate a new test case, say test Case C. Similarly, we introduced a new MFS, but different from the first scenario, this time we break the original MFS in the test case A when we generate C, which means the MFS in A is not in C.

At this situation it will influence our identifying result for that our expected result is that C should passed the test as we have broken the MFS in A. But the result failed at last. And we could owe this failure to some schema in A that is not the real MFS if we did nothing to deal with the introducing problem.

So we just need consider the possibility of the situation of simultaneously broking one MFS and introducing another MFS. This metric is related to the cost of changing one schema to another schema. The followed formula define the changing cost of two schemas:

ChangeCost(A, B) = $|T(A,B)| + \sum_{i \in T(B,A)} (P_i/2) + \sum_{i \in S(A,B)} (|A_i - B_i|/2)$

In this formula, A and B represent two different schemas. the denotation of T(A,B) gives the parameters in A but not in B. And S(A,B) means the parameters in both A and B, but their value is different. P_i refers to the number of values in the ith parameter of SUT, and A_i is the value of one factor in Schema A , the factor is the ith parameter of SUT.

Then the introduce rate of a SUT is defined as:

$$\frac{\sum_{a,b \in MFSs, a \neq b} (ChangeCost(a,b))}{|MFSs| \times |MFSs - 1|}$$

6.2.2 result and analysis

The statistic info of MFSs of the real software is listed in Table 6.

In this table, "exp ID" is the id of exception which the MFSs will trigger, "MFSs" gives the number of MFSs that can trigger this type of Exception show in "exp ID" column. Column "degree than 2" list the number of MFSs that have a degree larger than 2.

TABLE 6
MFSs information of each exception

exp ID	MFSs	degree than 2	mutliple MFSs	overlapped MFSs	intro rate
1	9	8	8	8	0.1028
2	24	23	25	1	0.0145
3	56	56	1	1	0.0026
4	1	1	0	0	-

"multiple MFSs" means the number of configurations that contain multiple MFSs. The column "overlapped MFSs" gives the number of configurations that contain MFSs that overlapped each other. And the last column "intro rate" shows the introduce possibility of newly MFSs when generating extra test configuration.

From this result, we will answer the sub-question of Case study 1 one by one.

1)Do there existed multiple MFSs in the same configuration?

Answer: yes. Although it is rare among the failing configurations(for exception 1, there are 36855 configurations trigger the same exception, and only 8 configurations contain multiple MFSs), but it do exist in the configurations of real software, we can see except the 4th exception which just has one MFS, all the other exception have configurations have multiple MFSs, which are 8,25,1 respectively.

2)if so, how many of them overlapped each other?

Answer: most of them are overlapped each other. We learned configurations which have overlapped MFSs is 8, 1, 1 respectively. As we all know, the configurations have overlapped MFSs is just one part of the configurations have multiple configurations. Considering the configurations contain multiple MFSs is rare, which are 8 , 25, 1 respectively, then the We think it is a high possibility when we encounter the situation that configurations contain multiple MFSs and these MFSs overlapped with each other.

One possible explanation for this phenomenon may be real softwares may have many branches, and these branches may have iteration, so for some MFSs , they may share the same entrance of the branch.

3)Do any of the MFSs have the degree larger than 2.

Answer: yes. It is clearly that almost all the MFSs in the software have a degree than 2, except one MFS for exception 1 (8 among 9 MFSs have degrees larger than 2) and one MFS for exception 2(23 among 24 MFSs have degrees larger than 2).

4)What the possibility of introducing newly MFSs when generating extra configurations?

Answer: the possibility varies from one to another.

We can learn from the table that the intro rate of exception 1 is 0.1028, which are the biggest than others, and the smallest is the exception 3, which is 0.0026. They differ markedly, so the possibility of introducing newly MFSs depends on the specific exception and may have big difference among each other.

6.3 Study 2: how these algorithms behave when applied in real softwares

The second study aims to answer the Q2. We will evaluate the performance of each algorithm in identifying the MFSs of the real softwares.

6.3.1 Study setup

To conduct this case study, We will feed one failing configuration to each algorithm and use them to identify the MFSs. Then we will compare the results get by each algorithm to the real MFSs in that configuration given in the study 1 respectively. The comparison metrics is similar to the simulated experiment, which consists of the number of extra test configurations needed, the precise and the recall. To be fair, no other information is given to each algorithm except the feeded failing configuration. We will repeat this comparison for each failing configuration of a exception. At last we will report the average number of test configurations , precise and recall for each algorithm.

As we will take a large number of configurations to identify(147472 for HSQLDB), this is too big compact to some algorithms, such as TRT, IterAIFL, they can't even accomplish the task. Some other algorithm can complete this task but with a huge time cost, such as SP, it will take weeks or even months to complete. We will omit these algorithms in our comparison to make this study available in a relative short time. As a result, we just choose ChainFeedBack , FIC, RI, OFOT, LG, CTA as our comparison algorithms.

6.3.2 result and analysis

The result is shown in table 7. In this table, Column "exp ID" still means the ID for the specific exception. Column "algorithm" gives the specific algorithm measured in this row. Column "num of extra configs" shows the average number of extra configurations needed to generate to identify the MFSs. Column "recall" and "precise" respectively shows the average recall and precise which have been defined in the simulated experiment for each algorithm.

We will discuss the result by column:

1)the number of extra configurations: We may get two points in this column: a. no algorithms always

TABLE 7
comparison in real softwares

exp ID	algorithm	num of extra configs	recall	precise
1	ChainFeedBack	15.061	1.0	0.9995
	FIC	9.023	0.9991	0.9988
	RI	9.024	0.9991	0.9988
	OFOT	19.007	0.9997	0.9995
	LG	11.000	0.0	0.0
	CTA	19.0	0.9997	0.9995
2	ChainFeedBack	13.058	0.9999	0.9996
	FIC	6.0160	0.9999	0.9997
	RI	6.0161	0.9999	0.9997
	OFOT	19.0	0.9997	0.9997
	LG	11.0005	0.9998	1.0
	CTA	19.0	0.9997	-
3	ChainFeedBack	19.793	0.9827	0.9827
	FIC	61.6551	0.8879	0.89655
	RI	62.6206	0.9396	0.9482
	OFOT	19.0	0.9827	0.9827
	LG	5.3448	0	-
	CTA	19.0	0.9137	0.9137
4	ChainFeedBack	19.0	1.0	1.0
	FIC	65.0	1.0	1.0
	RI	65.0	1.0	1.0
	OFOT	19.0	1.0	1.0
	LG	5.0	0.0	-
	CTA	19.0	1.0	1.0

need the smallest configurations in all conditions. For example,FIC needs the smallest configurations for exception 1(9.023) and exception 2(6.016) but needs the largest number of configurations for exception 4(65.0) as FIC may need a high cost dealing high way degree schema. RI performs good at exception 2(6.016), but performs bad at exception 3(62.62) and 4 (65.0). LG needs the smallest number of configurations for exception 3(5.344) and exception 4(5.0), but for exception 1 and 2 it doesn't perform as good as FIC and RI. b. OFOT and CTA performs almost the same for all the exception exceptions(almost as 19.0, exception the OFOT for the exception 1(19.0007)), this is because they are not the adaptive method, they just change one factor one time for all the conditions. And the unique different is that OFOT add some more configurations to deal the introduce problem. c. Our algorithms performs moderately, better than OFOT for exception 1 and 2 but worse than others, and better than FIC and RI for exception 3 and 4 and worse than others. This is because our algorithm consider more conditions such as introduce, overlapped and high degree problem.

2)recall : our algorithms perform the best in all the circumstances for this metric, which means our algorithms can find more MFSs than others algorithms when identifying the MFSs in a failing configuration. The reason why our approach have advantages over others is that we consider more scenarios when fed a failing configurations, especially we took account of the case multiple , overlapped MFSs in a test configuration and the case when the MFSs have a high degree. Other approaches may consider some of them, but none of them consider all these scenarios. Another point that needed note is that for exception 1, 3 and 4, the recall of LG algorithm is 0, it is a signal that in this three circumstances, the MFSs in the SUT have degrees larger than 2.

3)precise : algorithm OFOT and our approach performs better than others (our approach is litter weaker than OFOT for the exception 2, which ours is 0.9996, OFOT is 0.9997). As we have discussed before, the condition that introducing newly configurations when generating extra test configurations can make the identifying result incorrectly, so it is a nature idea that if the algorithm deal with the introducing newly MFSs, the algorithm can get a more accurate result than those approaches don't consider this scenario. The result data show in this table is coincide with this idea. We can also learn that the machine learning approach– CTA can also get a well result (0.9995, - , 0.9137, 1.0, slightly weaker than OFOT and our approach) in this column. Although this approach didn't consider the introducing scenario, the nature of dealing noisy data in machine learning can give this approach a some relief from those bad affects.

In addition, we can find some regular among the exceptions, i.e., for all the algorithms , the recall and

precise is best in 4, second best in 2 , little worse in 1, the worst in 3. This is coincided with the intro rate, which means that the intro rate really have a influence for the performance in algorithms.

Overall, the answer to Q2 is: Our algorithms preforms good at recall and precise, and moderately in num of extra configs. This is coincided with the result in simulated experiment.

6.4 Study 3: Is combination useful?

We can learn from the result in the second case study that none of these algorithms can be the best among all the scenarios. So a nature question is that can we combine the result of multiple algorithms to find a better result. In this study we will carry out some experiments to answer this question, i.e., the Q3.

6.4.1 Study setup

To utilize the result of each algorithm, we build a voting system. Thus, choose some algorithms and take a voting schedule: if the a schema is identified more than one algorithms, we take it as MFS, otherwise, we will discard it. We will vary the number of algorithms and adopt different combination of algorithms to observe the result. In specific we will take 2, 3, 4, 5 and 6 algorithms as the voters respectively. In addition, we will take all the combination of algorithms for a specific number. Take 2 for example we will choose ChainFeedBack and OFOT as a combination, ChainFeedBack and RI as another combination and so on. The total number combination of algorithms for a specific number n is $\binom{6}{n}$.

For each combination of algorithms, we will record the metric ″recall″ and ″precise″ of voting result. To measure if the voting system is useful, we compare this voting ″recall″ to the highest ″recall″ of the algorithm among the combination. The same comparison is also made to the ″precise″ metric. Note that the highest ″recall″ and the highest ″precise″ is not necessarily belong to the same algorithm in a combination. As a example, for the combination ″OFOT″ and ″RI″ for the exception 2. The highest value of ″recall″ is , which belongs to ″OFOT″. and the highest value of ″precise″ is, which is belong to the ″RI″. And our voting result will compare to the recall ″″ and precise ″″ respectively.

6.4.2 result and analysis

The detail comparison result is listed is table 8. In this table, ″Column″ means. the ″+″ signal. means a promotion over the highest . and the ″-″ signal means a decreasing of the highest.

The answer to Q3 is that: yes, it has promotion, but very little.

6.5 threats to validate

how seeded bugs are representative auto oracle, if we don't have a correct version, or we can't determine a test case is faulty and wrong, then what can i do? timing result One worrying aspect of this research is that it seems to consider only the number of tests and number of faults uncovered. In the practice of testing, it is as important or more important to know testing times.

7 Related works and Discuss

Detail list the history. In the experiment section, we just give some brief introduction.

Nie's approach in [3] and [6] first separates the faulty-possible tuples and healthy-possible tuples into two sets. Subsequently, by changing a parameter value at a time of the original test configuration, this approach generates extra test configurations. After executing the configurations, the approach converges by reducing the number of tuples in the faulty-possible sets. Delta debugging [5] proposed by Zeller is an adaptive divide-and-conquer approach to locating interaction fault. It is very efficient and has been applied to real software environment. Zhang et al. [4] also proposed a similar approach that can identify the failure-inducing combinations that has no overlapped part efficiently, Colbourn and McClary [7] proposed a non-adaptive method. Their approach extends the covering array to the locating array to detect and locate interaction faults. C. Martiez [8-9] proposed two adaptive algorithms. The first one needs safe value as their assumption and the second one remove the assumption when the number of values of each parameter is equal to 2. Their algorithms focus on identifying the faulty tuples that have no more than 2 parameters. Ghandehari.etc [10] defines the suspiciousness of tuple and suspiciousness of the environment of a tuple. Based on this, they rank the possible tuples and generate the test cases. Although their approach imposes minimal assumption, it does not ensure that the tuples ranked in the top are the faulty tuples. Yilmaz [11] proposed a machine learning method to identify inducing combinations from a combinatorial testing set. They construct a classified tree to analyze the covering arrays and detect potential faulty combinations. Beside this, Fouch?[12] and Shakya [13] made some improvements in identifying failure-inducing combinations based on Yilmaz' work.

We list a comprehensive detail and comparison table 9as followed.

But even we get the failure-inducing schemas, it is still having a gap to fetch the failure causing root from the code. Such as int the TCAS, we get the failure-inducing schemas, and this is caused by a code mutation in the code such as followed:. So in the future, we will analysis the relationship between the failure-inducing schemas and the real code causing.

TABLE 8
the combination of algorithms

	ChainFeedBack	FIC	RI	OFOT	LG	CTA
ChainFeedBack	-	\	\	\	\	\
FIC	\	-	\	\	\	\
RI	\	\	-	\	\	\
OFOT	\	\	\	-	\	\
LG	\	\	\	\	-	\
CTA	\	\	\	\	\	-

TABLE 9
The comparison of each algorithms

algorithms	time complexity	space complexity	multiple schemas	strength of schema	safe value	result style
CMINFS	O(log(n))	O(n)	yes, no limit, can overlapped	1 n	can handle	precise
FIC	O(log(n))	O(n)	yes, no limit, can't overlapped	1 n	can't handle	precise
TRT	O(log(n))	$O(2^n)$	yes, no limit, can overlapped	1 n	can handle	precise
Spetrum	O(n*t)	$O(n*t)$	yes, no limit, can overlapped	1 n	can handle	a rank of possible

8 Conclusion

The conclusion goes here.

Appendix A
Proof of the First Zonklar Equation

Appendix one text goes here.

Appendix B

Appendix two text goes here.

The procedure :

$mutant_a$
$mutant_r$
$ispendingschema$
$makechain$

Acknowledgments

The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...The authors would like to thank...

References

[1] C. Song, A. Porter, and J. S. Foster, "itree: Efficiently discovering high-coverage configurations using interaction trees," in *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 2012, pp. 903–913.

[2] J. Bach and P. Schroeder, "Pairwise testing: A best practice that isnt," in *Proceedings of 22nd Pacific Northwest Software Quality Conference*. Citeseer, 2004, pp. 180–196.

[3] C. Nie and H. Leung, "The minimal failure-causing schema of combinatorial testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, p. 15, 2011.

[4] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *Software Engineering, IEEE Transactions on*, vol. 32, no. 1, pp. 20–34, 2006.

[5] Z. Zhang and J. Zhang, "Characterizing failure-causing parameter interactions by adaptive testing," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 331–341.

[6] L. S. G. Ghandehari, Y. Lei, T. Xie, R. Kuhn, and R. Kacker, "Identifying failure-inducing combinations in a combinatorial test set," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 370–379.

[7] C. Martínez, L. Moura, D. Panario, and B. Stevens, "Algorithms to locate errors using covering arrays," in *LATIN 2008: Theoretical Informatics*. Springer, 2008, pp. 504–519.

[8] C. J. Colbourn and D. W. McClary, "Locating and detecting arrays for interaction faults," *Journal of combinatorial optimization*, vol. 15, no. 1, pp. 17–48, 2008.

[9] X. Niu, C. Nie, Y. Lei, and A. T. Chan, "Identifying failure-inducing combinations using tuple relationship," in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 271–280.

[10] K. Shakya, T. Xie, N. Li, Y. Lei, R. Kacker, and R. Kuhn, "Isolating failure-inducing combinations in combinatorial testing using test augmentation and classification," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 620–623.

[11] Z. Wang, B. Xu, L. Chen, and L. Xu, "Adaptive interaction fault location based on combinatorial testing," in *Quality Software (QSIC), 2010 10th International Conference on*. IEEE, 2010, pp. 495–502.

[12] J. Li, C. Nie, and Y. Lei, "Improved delta debugging based on combinatorial testing," in *Quality Software (QSIC), 2012 12th International Conference on*. IEEE, 2012, pp. 102–105.

[13] C. Martínez, L. Moura, D. Panario, and B. Stevens, "Locating errors using elas, covering arrays, and adaptive testing algorithms," *SIAM Journal on Discrete Mathematics*, vol. 23, no. 4, pp. 1776–1799, 2009.

[14] S. Fouché, M. B. Cohen, and A. Porter, "Incremental covering array failure characterization in large configuration spaces," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 177–188.

PLACE
PHOTO
HERE

Michael Shell Biography text here.

John Doe Biography text here.

Jane Doe Biography text here.