Efficiently Identify the failure-inducing schemas

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

Abstract—It is common that the software under test(SUT) fails during testing under some specific test configurations(we called failing configurations) and passes under the others. For these failing configurations,in the most general case,not all the options in this configuration is related to the failures. It is desirable to identify the specific options responsible for the failures,i.e., failure-inducing schemas,as by doing this it can reduce the debugging effort.
Combinatorial testing(CT) can effectively detect the errors in SUT which are triggered by interactions of options.However, it provide weak support for identifying them.In this paper, we propose an efficient approach which can aids the CT to locate them.Through using the notions of CMINFS(represent for candidate minimal faulty schemas) and CMAXHS(represent for candidate maximal healthy schemas), our approach can get an efficient performance when identifying the failure-inducing schemas compared to our prior work.Also we make an improvement on the approach to better handle the case where additional failure-inducing schemas may be introduced by newly generated test configurations.
Moreover, we conduct empirical studies on both widely-used real highly-configurable software systems and simulated toy softwares.The results shows that the failure-inducing schemas do exist in software and our approach can identify them more effectively and efficiently than other works.

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

1 INTRODUCTION

MANY modern softwares are highly configurable and customizable, while this can increase the reusability of common code base and basic components, make the software easier to be extended, enable the software to run across different environments, however, it also make the testing task to be more challenging. This is because we need to test the software under every possible configurations(we called configuration space) to ensure the correctness of it, which is not feasible in practice. Combinatorial testing technique can handle this problem well. It design a relatively small test suite from the configuration spaces to test the SUT and can ensure to cover all the interactions of option values which not exceed t. When we find some configurations fail in this test suite, which means this suite detect some interaction fault, what we want to know next is to identify which specific interaction options and its values are the cause of these failures. To our best know, combinatorial testing provide week support to identify them.

As an example,consider the database MYSQL and some of its configuration options list in Table 1. Assume we generate a test suite consists of available configurations using combinatorial testing technique, and find some test configurations failed, say, (). Then which specific is the cause, is it *extra-charsets=NULL* and *assembler = Enabled* , or *extra-charsets = NULL* and *assembler=Enabled* and *innodb-flush-log = 2* or other possible interactions . Generally, for a failed test configurations consists of n options have $2^n - 1$ possibilities. To identify them is important, for they can help reduce the scope of source code need to be inspected.

TABLE 1
MySQL configuration options example

Compile Time Options	
Binary Options(Enabled/NULL)	
assembler, local-infile, thread-safe-client, archive-storage-engine, big-tables, blackhole-storage-engine,client-dflags, csv-storage-engine, example-storage-engine, fast-mutexex, federated-storage-engine, libedit, mysqld-ldflags, ndbcluster, pic, readline, config-ssl, zilb-dir	
Non-binary Options	**values**
extra-charsets	-with-extra-charsets=all, -with-extra-charsets=complex, NULL
innodb	-with-innodb, -without-innodb, NULL
Runtime options	**values**
transaction-isolation	READ-UNCOMMITTED, READ-COMMITTED, REPEATABLE-READ, SERIALIZABLE, NULL
innodb-flush-log	0, 1, 2, NULL
sql-mode	ANSI, TRADITIONAL, STRICT_ALL_TABLES

We call the faulty interactions of option values the failure-inducing schemas, in effect, this notion is not limited in configurable software testing. The input testing may want to find the failure causing input

- *M. Shell is with the Department of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, 30332. E-mail: see http://www.michaelshell.org/contact.html*
- *J. Doe and J. Doe are with Anonymous University.*

interaction. The GUT testing may want to find the events interaction which makes the software down. The regression testing may want to find which change trigger the new failure. The html testing may want to isolate the minimal faulty related html source code. The Compatibility testing may want to test which some software run simultaneously will crash the system.

How to identify them is not easy, in effect, there are two problems need to be solved: First, we just need to know the properties of the faulty schemas and healthy schemas, i.e., the differences between them. Figuring out that can help us to design approaches to identify a schema to be healthy or faulty. Second, we should find some strategy to check every possible schemas in a test configuration while using as small resources as possible. As the possible schemas in a test configurations can get to 2^n(n is the number of options in a configuration), then we want the resources needed is logarithmic related to the number of schemas, which will result in the complexity of algorithm with O(n)(n is the number of options in a configuration).

In this paper, 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 describe our approaches for identify failure-inducing schemas. Section 4 give the comparisons in theoretical metrics. Section 5 describe the experiments design. Section 6 list the result of the experiments. 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 (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 4 (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 5 (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 the case 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 6 (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 7 (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 8 (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 9 (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 THE APPROACH TO IDENTIFY THE MINIMAL FAILURE-INDUCING SCHEMAS

Based on these definitions and proportions, 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.

3.1 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). □

3.2 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.

3.2.1 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

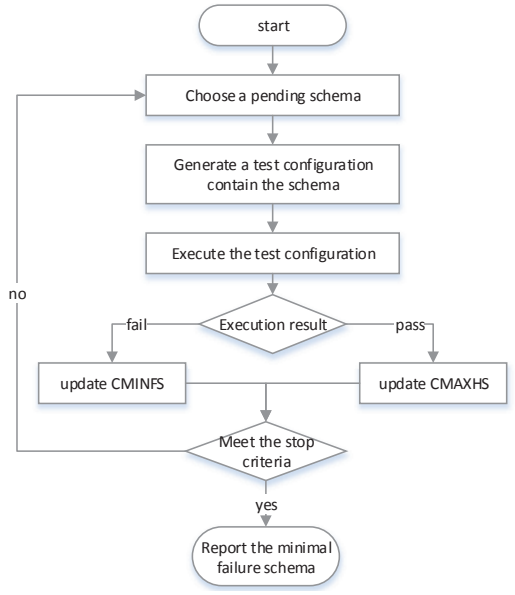

Fig. 1. overview of approach of identifying MFS

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 10 (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).

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: $\mathcal{S}_{\mathcal{UPS}} \leftarrow \{s | s \in lists \wedge s\ is\ pending\}$

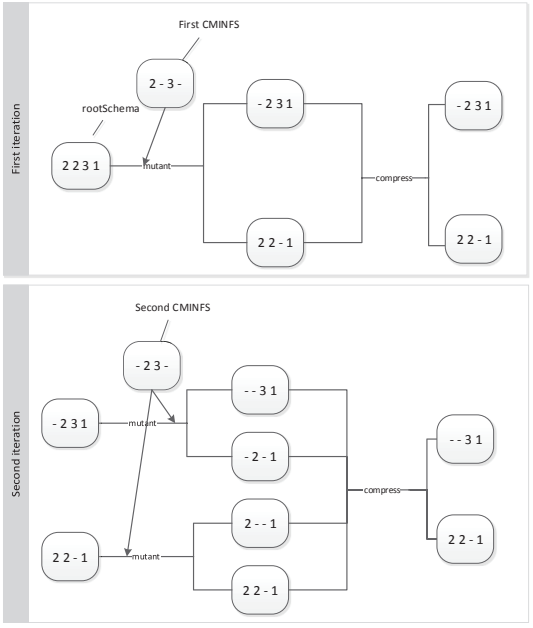

Fig. 2. example of getting up pending schemas

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

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:

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: $\mathcal{S}_{\mathcal{DOWNS}} \leftarrow \{s | s \in lists \wedge s \ is \ pending\}$

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

1: $\mathcal{S}_{\mathcal{UPS}} \leftarrow getUPS(\mathcal{T}, \mathcal{S}_{\mathcal{CMINFS}}, \mathcal{S}_{\mathcal{CMAXHS}})$
2: $\mathcal{S}_{\mathcal{DOWNS}} \leftarrow getDOWNS(\mathcal{T}, \mathcal{S}_{\mathcal{CMINFS}}, \mathcal{S}_{\mathcal{CMAXHS}})$
3: $max \leftarrow 0$
4: $head \leftarrow NULL$
5: $tail \leftarrow NULL$
6: $chain \leftarrow NULL$
7: **for each** up in S_{UPS} **do**
8: **for each** $down$ in S_{DOWNS} **do**
9: **if** $distance(up, down) \geq max$ **then**
10: $max \leftarrow distance(up, down)$
11: $head \leftarrow up$
12: $tail \leftarrow down$
13: **end if**
14: **end for**
15: **end for**
16: $\mathcal{CHAIN} \leftarrow makechain(head, tail)$

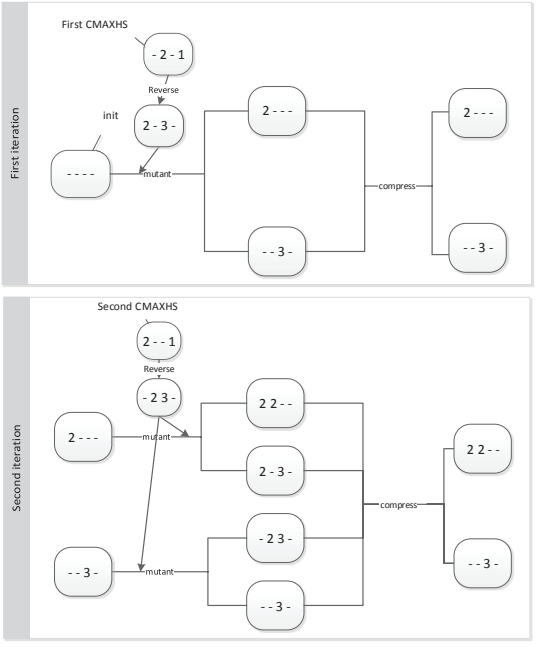

Fig. 3. example of getting down pending schemas

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

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 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.

3.2.2 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,

Algorithm 4 identify process

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

1: **while** *hasn't meet the end criteria* **do**
2: **if** *the beginning* or $headIndex > tailIndex$ **then**
3: $update(\mathcal{S}_{\mathcal{CMINFS}}, \mathcal{S}_{\mathcal{CMAXHS}})$
4: $longeset \leftarrow getLongest(\mathcal{T}, \mathcal{S}_{\mathcal{CMINFS}}, \mathcal{S}_{\mathcal{CMAXHS}})$
5: $headIndex \leftarrow 0$
6: $tailIndex \leftarrow length(longest) - 1$
7: $middleIndex \leftarrow 0$
8: **else**
9: $middleIndex \leftarrow \frac{1}{2} \times (tailIndex + headIndex)$
10: **end if**
11: $\mathcal{SCHEMA} \leftarrow longest[middleIndex]$
12: *generate a extra test configration* $\mathcal{T}'$ *contain* $\mathcal{SCHEMA}$
13: *execute SUT under* $\mathcal{T}'$
14: **if** *the test configration passed* **then**
15: $tailIndex \leftarrow middleIndex - 1$
16: **else**
17: $headIndex \leftarrow middleIndex + 1$
18: **end if**
19: **end while**
20: *report the minimal faulty schemas*

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

3.2.3 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.

3.2.4 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.

3.2.5 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.

3.2.6 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.

3.2.7 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.

3.3 example

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

3.4 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.

TABLE 2
An example of identifying

$\mathcal{S}_{\mathcal{CMINFS}}$	$\mathcal{S}_{\mathcal{CMAXHS}}$	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

$\mathcal{S}_{\mathcal{CMINFS}}$	$\mathcal{S}_{\mathcal{CMAXHS}}$	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

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

4 Evaluation using simulated model

In this section, we give an a series of studies of our two identify algorithms on simulated model. The goal of our experiments is to evaluate the efficiency and effectiveness of our two identify algorithms compared with other existed algorithms. The reason why we use simulated model is that we can control. We will validate in real softwares in the next section.

4.1 comparison algorithms

There are some 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 when given an executed test configurations while the second one do need. Our algorithm is belong to the second part. To make a clear comparison, we just compare the algorithms which all belong to this part.vThe more detail and discuss of all the algorithms is listed in the section 6. The compared algorithms is as follows:

4.1.1 OFOT

This method is proposed by Nie[]. It 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.

4.1.2 IterAIFL

This method proposed by Wang[] is a mutant based ont the Nie's[]. It define the "change strength" that not like the previous one just change one factor one time, it may change many factors one time.

4.1.3 FIC

Zhang [] propose this method which is combined the OFOT method and binary search. Different from the OFOT, it change half factor one time to quickly located in one factor firstly and then keep the factor and loop the process.

4.1.4 RI

Lee propose this algorithm[]. Similar to the FIC, it is also inspired by the delta debugging. The difference is the order that it change factors.

4.1.5 Spetrum based method

Different from the aforementioned methods, 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.

the parameter setting is as follows: the predefined number is 5, and the number of turns of finding the minimum pe is the parameter mutipled.

4.1.6 Martin safe value

4.1.7 classified tree

must add ”-M 1” ”-U” confidence factor 0.25

4.1.8 TRT

This algorithm is proposed in[].Different from this work. It should record all the schemas of a failed test configuration. It is very resource assume and this is reason we propose this work. A second difference is that the TRT algorithm when meet the situation the extra test configuration may introduce newly faulty schemas is using more test configuration to confirm a schema each time when find a schema is labeled as a faulty schema while this work just validate one time at the end of algorithm.

4.2 subject programs

This section will give an general illusion of the algorithms. We will give an test bench to measure the properties of all the algorithms. All the programs are toys and have the property that we can easily control its input parameters and faulty schemas, which makes clearly get the view of what we want to know from the algorithms. In addition, we will corroborate the results by observing each algorithm of identifying failure-inducing schemas in real software in section 6.

4.3 evaluation metrics

The metrics are additional generated test configurations, the recall and precise.The recall and precise is defined as:

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

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

4.4 experiment setups

We will do the three experiment: (1)single faulty schema. (2)multiple faulty schema in a failed test configuration. (3)multiple faulty schema, which give a fail test configuration contain one faulty schema , we do so that give an chance that the extra test configuration can introduce another faulty schema.

In detail, for every experiment in the toy experiment section, we first vary the number of input

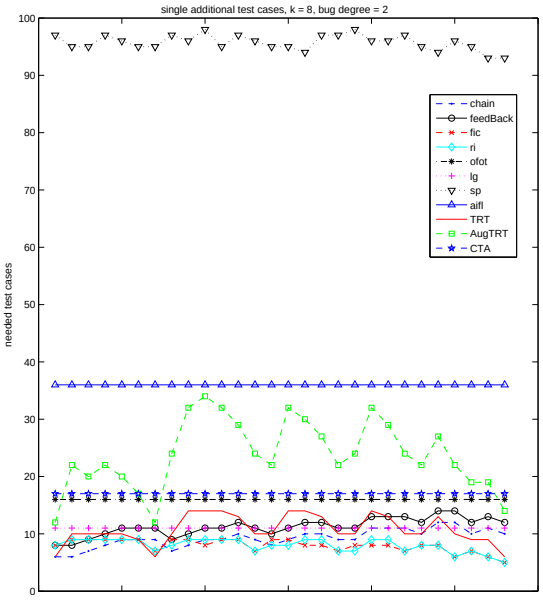

Fig. 4. k = 8, t = 2, additional test suites

parameters of SUT, i.e., n= 10, 40, 80, 120, 160, 200, 240, 280, 320, 360 to observe the behaviour of the algorithms executed among different size of SUT. Then to get an relative generally evaluation, we inject each possible t-value schema (reach to $\binom{n}{t}$,) one time for the single faulty schema experiment and each possible combination of two t-value schema one time for the multiple faulty schema experiment.(t = 2,3,4,5,6). For the third, we first fix one t-value faulty schema, and give an failed test configuration contain this schema, then for each algorithm aims to identify this faulty schema, the first extra test configuration each algorithm generate, we random select an t-value schema (do not contained in the original failing test configuration) as the introduced faulty schema.

For all the three experiment, we record the number of extra test configurations, correctly identified schemas and incorrectly identified schemas. At last we compute the recall and presise along with the number of extra test configurations.

4.5 results and discuss

4.6 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.

5 Empirical Case Study

While we got know that our approach have a better performance than others in several scenarios such

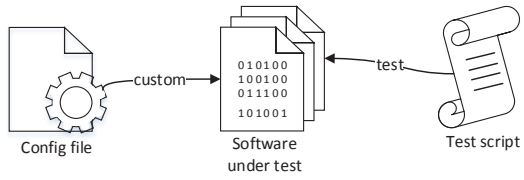

Fig. 5. proceeding

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.5 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.

5.1 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.

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

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

5.1.1 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 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

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

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.

5.2 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.

5.2.1 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:

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

5.2.2 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. "multiple MFSs" means the number of configurations that contain multiple MFSs. The column "overlapped

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	-

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.

5.3 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.

5.3.1 Study setup

To conduct this case study, for each algorithm. We will feed it with one failing configuration, for which we will use the algorithm to identify the MFSs. Then we will compare the result of this algorithm to the real MFSs in that configuration given in the study 1. 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 the computing is very time-consuming, we just chose some algorithms mentioned in simulated experiment, they are ChainFeedBack , FIC, RI, OFOT, LG, CTA.

5.3.2 result and analysis

The result is show in table 7. In this table, Column "exp ID" still means for the specific exception of the ID. 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" shows the recall and column "precise" shows the precise.

From the data listed in the table shows that our algorithm still perform better than others. So the answer to Q2 is: the result in real softwares coincided with the result in simulated experiment.

5.4 Study 3: Is combination useful?

5.4.1 Study setup

5.4.2 measurement

5.4.3 result and analysis

6 Related works and Discuss

Nie's approach in [3] and [6] first separates the faulty-possible tuples and healthy-possible tuples into two

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

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 dividend-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 8as 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.

7 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 :
$mutant_a$
$mutant_r$
$ispendingschema$
$makechain$

TABLE 8
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

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...

REFERENCES

[1] H. Kopka and P.W. Daly, *A Guide to LATEX*, third ed. Harlow, U.K.: Addison-Wesley, 1999.

PLACE PHOTO HERE

Michael Shell Biography text here.

John Doe Biography text here.

Jane Doe Biography text here.