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.

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

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

• *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.

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.

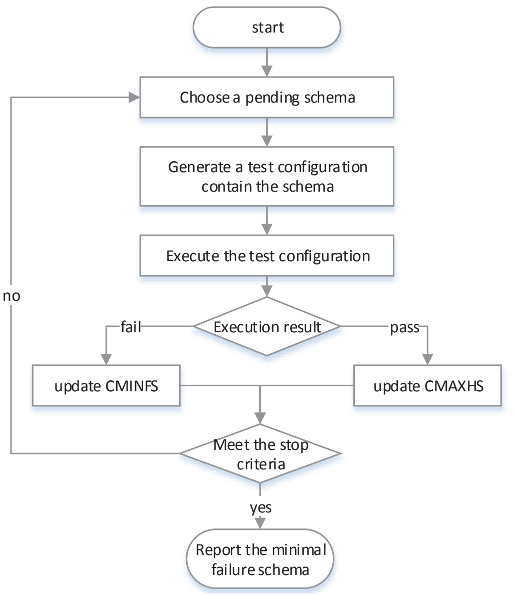
start
Choose a pending schema
Generate a test configuration contain the schema
Execute the test configuration
no
fail
Execution result
pass
update CMINFS
update CMAXHS
Meet the stop criteria
yes
Report the minimal failure schema

Fig. 1. overview of approach of identifying MFS

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 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).

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

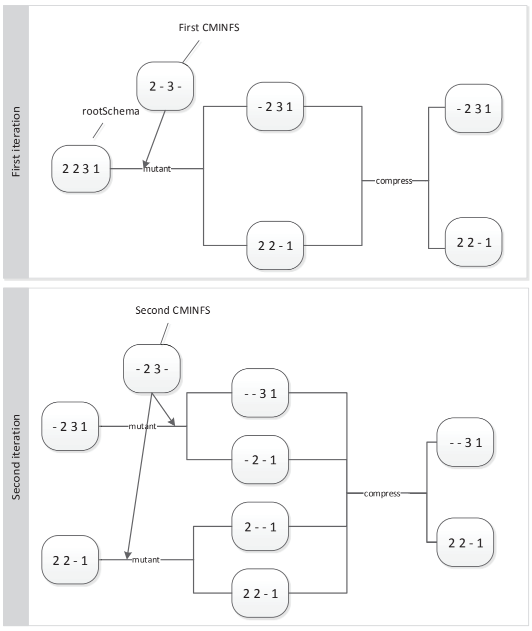

Fig. 2. example of getting up pending schemas

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

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: $\mathcal{CHAIN} \leftarrow makechain(head, tail)$
```

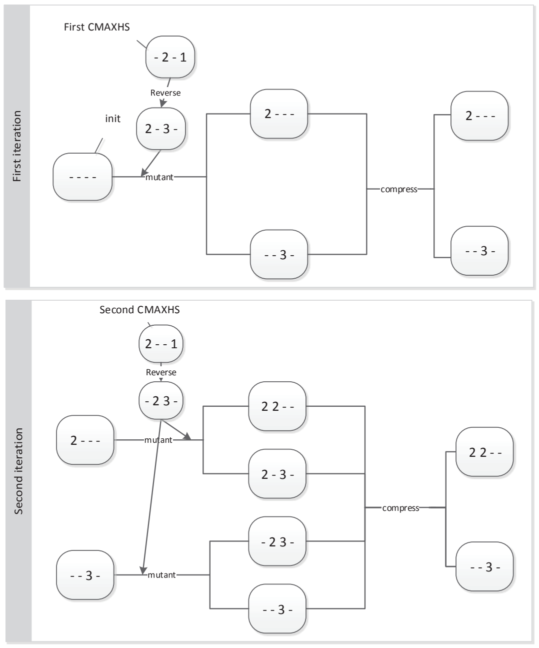

Fig. 3. example of getting down pending schemas

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, 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
 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: $report\ the\ minimal\ faulty\ schemas$
```

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.

3.3 example

We will give an complete example followed. Assume that a SUT is . constaints.

3.4 Without Safe Values Assumption

feedback machinery

4 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] 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.