IterAIFL:

Iterate 1: (change one factor)

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Iterate 2: (change two factor)

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |

Iterate 1: (change three factor)

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 |

……

Until the number of factor is equal to the length of test case

Then

Using the formula

*Schemas in failed - Schemas in passed*

Get the faulty schemas

And then reduce the parent schemas in the set to get the MFSs

Characteristics:

Iterate the SOFOT, then filtrate schemas which in passed test cases to get faulty possible schemas.

Can identify multiple schemas (but must know the number of non-overlapped MFSs , or the algorithms will ended in advance).

Can’t not identify the introduced schemas.

OFOT

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 1 | 1 | 1 |

|  |  |  |  |
| --- | --- | --- | --- |
| Col1 | Col2 | Col3 | Clo4 |
| |  |  |  |  | | --- | --- | --- | --- | | 0 | 1 | 1 | 1 | | 2 | 1 | 1 | 1 | | 3 | 1 | 1 | 1 | | 4 | 1 | 1 | 1 | | |  |  |  |  | | --- | --- | --- | --- | | 1 | 0 | 1 | 1 | | 1 | 2 | 1 | 1 | | 1 | 3 | 1 | 1 | | 1 | 4 | 1 | 1 | | |  |  |  |  | | --- | --- | --- | --- | | 1 | 1 | 0 | 1 | | 1 | 1 | 2 | 1 | | 1 | 1 | 3 | 1 | | 1 | 1 | 4 | 1 | | |  |  |  |  | | --- | --- | --- | --- | | 1 | 1 | 1 | 0 | | 1 | 1 | 1 | 2 | | 1 | 1 | 1 | 3 | | 1 | 1 | 1 | 4 | |
|  |  |  |  |

If a Col is all passed:

As example Col 1 is passed, then the schema must contain the first factor. I.e. (1 - - -) ≺ MFS

If a Col is all failed:

As example Col 2 failed, then

1. The corresponding factor (- 1 - -)It is not a part of MFS
2. Every test cases introduce new MFSs (too bad, and it can’t handle this problem, then the get result ≺ MFS but not = MFS)
3. (- 1 - -) ≺ one MFS , but it did not ϵ MFS a MFS b (the SUT has overlapped MFS, and this case it just can get the result of the overlapped part of the MFSs)

If a Col has passed test cases and failed tests:

As example Col 3, then the schema must contain the first factor. I.e. (- - 1 -) ≺ MFS

And the failed test cases must contain new MFSs. As example

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 1 | 2 | 1 |

Failed, and all the other passed, then we will iterate like this to find newly introduced MFSs. Tested test cases does not

|  |  |  |  |
| --- | --- | --- | --- |
| Col1 | Col2 | Col3 | Clo4 |
| |  |  |  |  | | --- | --- | --- | --- | | 2 | 1 | 2 | 1 | | 3 | 1 | 2 | 1 | | 4 | 1 | 2 | 1 | | 0 | 1 | 2 | 1 | | |  |  |  |  | | --- | --- | --- | --- | | 1 | 2 | 2 | 1 | | 1 | 3 | 2 | 1 | | 1 | 4 | 2 | 1 | | 1 | 0 | 2 | 1 | | |  |  |  |  | | --- | --- | --- | --- | | 1 | 1 | 3 | 1 | | 1 | 1 | 4 | 1 | | 1 | 1 | 0 | 1 | | 1 | 1 | 1 | 1 | | |  |  |  |  | | --- | --- | --- | --- | | 1 | 1 | 2 | 2 | | 1 | 1 | 2 | 3 | | 1 | 1 | 2 | 4 | | 1 | 1 | 2 | 0 | |
|  |  |  |  |

Characteristics:

Simple and intuition. Can identify one schema in a failed test case, and can detect and identify introduced schemas (but one schema in one iterate). Can detect but can’t identify multiple faulty schemas in a test cases. Can detect the overlapped part of multiple faulty schemas but can’t identify the multiple faulty schemas.

FIC

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Light blue: first part, change it . Low part (every time , change Low and U)

Yellow: second part, let it original be(high part)

Grey: unrelated to the failure || U (need to modify)

Red: relatedto the failure (interaction fixed parameter)

Pass candi 🡪 low , fail candi 🡪 high

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | Fail |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | Pass |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | Pass |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | pass |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | Fail |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | Fail |

multiple

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Pass |

RI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Light blue: first part, change it

Yellow: second part, let it original be

Grey: unrelated to the failure

Red: related to the failure

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | Fail |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | Pass |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | Pass |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | pass |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | Fail |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | Fail |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | Fail |

multiple

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Pass |

Suspicious Based

Suspicious of Factor o: 1/3 (u(o) + v(o) + w(o))

u(o) =

v(o) =

w(o) =

suspicious of combinations : average of suspicious factors in the combination.

Suspicious of environment: average of suspicious factors not in the combination. ( choose the test cases contain the combination, then let the minimal environment as its suspicious of environment)

π : the suspicious combinations set . Initial all the t-way combinations in the failed - passed

F: executed test suite, contain test cases. t: the strength of combinations.

While(true){

Filtrate the combinations in π that are in passed test cases in F.

If π is empty

Return empty ------------------ no t-way failure-inducing combinations

Else if π is not change compared to the last iteration

Return π. ------------------ suspicious t-way combinations

Else

Rank π based on the suspicious expression.

Generate a set F’ contain the top ranked combinations.

If(some combination can’t generate more test cases contain )

it which means all the test cases contain it are failed and we marked it as failure-inducing combiantions.

Break.

Else

Execute all the test cases in F’

F = F U F’

}

Reduce(π)

CTA (Classified tree analysis)

Train data set : instances and class

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Atrr1 | Atrr2 | Atrr3 | Atrr4 | Atrr5 | Atrr6 | Atrr7 | Atrr8 | Class |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | Fail |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | Pass |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | Pass |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | pass |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | Fail |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | Fail |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | Fail |

Then using C4.5 classified tree to analysis the attribute and the class. Using Weka J48 algorithm can fulfil this task.

The rationale is as follows:

1. For each option, partition the train set based on the settings of that option.

2. Evaluate the option based on how well it partitions configurations of different classes.

3. Select the best option and make it the root of the tree.

4. Add one edge to the root for every option setting.

5. Repeat the process for each new edge. The process stops when no further split is possible.

(a forecast: it may not work well under overlapping bugs. Cause the partition is not that clear)

A extent, when given a executed test suite, it uses OFOT first to generate additional train data set(to get more information and to make the algorithm more accurate). And then using CTA to get the failure-inducing combinations.

Martine Safe value algorithm and binary values algorithm

Safe value algorithm

Safe values:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Fail test case:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 1 | 1 | 1 |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 1 | 1 | 1 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

fail

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

fail

not contain a faulty schema not contain a faulty schema

|  |  |
| --- | --- |
| 1 | 1 |

|  |  |  |
| --- | --- | --- |
| 1 |  | 1 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |

Fail 🡪

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |

Fail 🡪 1---1---

Martine binary values algorithm

Important therome:

A factor can be in two faulty schema with different values can only be the case:

1 1 1 1 1 1 1 1

0 0 0 0 0 0 0 0

A pass test case: 0 0 0 0 0 0 0 0

Then failed test case: 1 0 1 0 1 1 0 0

Set:

A : 1-0 faulty schema the “1” factors 🡪

B: 1-1 faulty schema the “1” factors , and all its “1” factor are not in A.

C: the remained factors.

Then the faulty schema can be

1 – 0 schema inside AP

1 – 0 schema between AS and C

1 – 1 schema inside B

1 – 1 schema between AS and B

1 – 1 schema between AS and C