Orphan Defects: Chance Finding?

Dhanyamraju S U M Prasad, Satya Sai Prakash Kanakadandi

HCL Technologies, India

Email: prasad@hcl.com, satyasaiprakash.k@hcl.com

Abstract - At least 5% and at most 25% defects escape the rigorous to routine testing practices of various software products. Defects imply various holes in the test suite creation as well as selection. We implemented hybrid distance based algorithms to derive defect to test case mapping that yielded two metrics - Test Coverage Metric and Test Sufficiency Metric. In addition to these two metrics, we stumbled upon a set of defects that are not mapped to any test cases which were named as 'orphan defects (OD)'. Finding orphan defects contrasts the testing team's belief that they have created a comprehensive test suite with total coverage, we have coined the term 'chance finding' to emphasize that it is serendipitous finding while analyzing various defects, both internal and customer reported. These defects have been further analyzed for any associated features and presented to the user for validation and adding required test cases. Upon using this procedure, one of the industry partners has brought down the orphan defects by about 90% over three patch releases and test coverage and sufficiency have been improved considerably.

Keywords—Test Coverage, Test Case Sufficiency, Matrix Regression, Orphan defect, Field Defect, Internal Defect

I. INTRODUCTION

This exposition elucidates the ordeal of releasing bug free software products. During software development testing team develops lots of test cases to verify the functionality of the software developed/written in development (Henceforth called as test bank). These test cases are being developed by taking the Requirements document/application screen layout prototype as basis. The test bank grows along with the application/software released for each release. As the development of software progresses, the test team might need to add more number of test cases to verify the newly created features, test cases has to updated because of changes in the screen flow, or test cases might need to be deleted as they become obsolete. Normally, during the software development the requirements might change drastically as the stakeholders of the development product managers, development team, testing team and others might feel that some changes are required to the software to make it more useful to the end customers or some new requirements might be added. But the requirement document might not be updated with the new requirements. Because of this dynamic nature of requirement changes and time constraints testing team might not update the test bank with new/modified test cases, leaving test bank incomplete. For software testing phase, the test manager generates a plan with his previous experience with software product/experience considering requirements of release. Testers execute the test cases identified in the test plan and reports the defects in defect management system. As part of testing phase, Along with executing test cases identified in the test plan, testing team also performs adhoc testing(randomly execute the features not following test scripts). If there are any issues identified during this adhoc testing process, the test team is supposed to update the test bank with these test cases. But due to time/process constraints the test time might miss to update the test bank.

The defects identified as part of testing phase are referred as internal defects. Once the product is released, the users of application report the issues they are facing in the product. These will be due to insufficient steps in the test cases, or software is installed/executed in new execution environment or test cases are not covering the scenario completely. The defects reported by the end users are called as Customer reported defects (CRD). The test cases in test bank is required to be updated

- Requirements changes
- With the defects found in testing phase where the test case is in-sufficient
- With the scenarios mentioned in CRD

But due to the time/resource and other constraints the testing team might not updated the test bank. This will lead the test bank is in-sufficient to verify the product functionality and the test plan will cover only limited functionality which will give low test coverage. Here we propose an approach to improve the test case sufficiency, test coverage and reduce the Customer Reported Defects. In this approach we use numerous Natural processing techniques and machine learning techniques to map the test cases to defects. First, using natural processing techniques identify the features and sub-features of the product. Second, For each feature, identify the corresponding test cases. Third, for each feature identify the corresponding
defects. Fourth, map the defects to the test cases. Once feature to defect mapping is completed, test cases are extracted based on the features and mapped to the defects

- All the unmapped defects are labeled as orphan defects
- Mapped defects to the test cases imply the test coverage insufficiency
- Based on feature groups of the ODs, test case insufficiency is estimated.

Test case coverage and sufficiency metrics are defined as below:

A. Test Coverage Metric:

From the literature, typical formula used for test coverage metric is the ratio of test cases executed to total number of test cases. This is a single metric measured against the complete test suite.

Test Coverage = (Number of Test Cases Executed / Total number of Test Cases) * 100

Ex: For a release assume 200 test cases got executed from the total list of 600, then the test coverage is 33.3% \(\left(\frac{200}{600}\right) * 100\)

With our approach all defects are put into either orphan bucket or mapped bucket.

Percentage of orphan defects (Orphan Defect Ratio = (Number of orphan defects/Total number of defects) * 100) adds validation flag to the general test coverage metric. Continuing the above example, after the product release say 300 defects have been reported of which 75 turnout to be orphans (Orphan defect ratio = 25%), implying that the test coverage is poor. As orphans are labeled after processing all test cases for any feature match, it is stronger metric for test coverage than the other literature formulation.

B. Test case Sufficiency Metric:

Test case sufficiency is individual metric pertinent to each test case and its sufficiency to detect bugs as per the expected functionality/behavior. Literature has more information and metrics related to test case efficiency and effectiveness. Sufficiency is outcome of this proposed invention.

Test case sufficiency is derived from the mapped bin of defects. As these defects are mapped with some feature or the other that is corresponding to one more test cases, Test Case Sufficiency = \(\{1 - \text{(Number of mapped defects for a feature/total defects)}\} * 100\)

From the above example, out of 225 feature mapped defects, say 30 defects correspond to feature A related to test case T1, then the T1 sufficiency is 90% \(\{1 - \left(\frac{30}{300}\right)\} * 100\)

II. MOTIVATION AND BACKGROUND

The strong motivation to build this cross team knowledge optimizer that extracts and automates the processes across developer, tester and support engineer. Defects emerge either because a developer misunderstands a requirement or a tester misinterpret or miss a use case altogether. In order to help track software defects and build more reliable systems, bug tracking tools have been introduced. Bug tracking systems like Bugzilla enable many users to serve as “testers” and report their findings in a unified environment. These bug reports are then used to guide software corrective maintenance activities and result in more reliable software systems. Via the bug tracking systems, users are able to report new bugs, track statuses of bug reports, and comment on existing bug reports [15].

Despite the benefits of a bug reporting system, it does cause some challenges. As bug reporting process is often uncoordinated and ad-hoc, often the same bugs could be reported more than once by different users. Hence, there is often a need for manual inspection to detect whether the bug has been reported before. If the incoming bug report is not reported before then the bug should be assigned to a developer. However, if other users have reported the bug before then the bug would be classified as being a duplicate and attached to the original first-reported “master” bug report. This process referred to as triaging often takes much time. Bug triagers should not assign these reports to different developers; this would be a waste of effort and a potential of causing conflicting changes being made to a system [10]. According to Capers Jones [2], poor software quality is most expensive (> $150 billion in US and $500 billion per year worldwide). And 15% of projects cancelled due to poor quality. Defect Removal Efficiency (DRE) defines no of defects Fixed before release, the DRE rate will be 73% to 94% for different kind of applications. It shows that there is a 26% to 6% of the defects will be released to the end users. The cost of detecting and fixing an issue during software development phase is less expensive than fixing a defect after release. According to Caper Jones[2] there are significant number of defects are not identified due to insufficient test cases. Test case insufficiency will be raised due to

- Dynamic requirements, where requirements are changed during software review process and test cases are not updated
- Missing design constraints, performance measures
- Missing test case steps
It shows bug fixing is costly and time consuming as it requires multiple teams to come together and solve the same. Support, Development and Testing teams have to seamlessly work towards fixing and elimination of bugs, especially after hearing the same from customers. Sandu Popa et al [1] describe the method to identify duplicate using NLP techniques. This will eliminate logging a duplicate defect in to defect management system. There are numerous papers for selecting a test case for regression testing. They explained numerous methods to select a test plan and test cases for regression testing. Here we are suggesting a method which will provide the test case coverage and test case sufficiency and the defects that don’t have any test cases.

Test Coverage: Associate defects to a test case. A feature which has maximum number of issues, give priority to that feature in testing.

Test Sufficiency: A test case which has maximum no of defects associated. That shows that the test case needs to be updated because of some missing steps or environtment changes.

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>Testcase Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title of the testcase1</td>
<td>Testcase Steps</td>
<td>ID1</td>
</tr>
<tr>
<td>Title of the testcase2</td>
<td>Testcase Steps</td>
<td>ID2</td>
</tr>
</tbody>
</table>

Orphan Defects: These are the defects which are not mapped to any test case. This shows the test bank incompleteness.

III. FEATURE BASED REGRESSION APPROACH

Our approach has following steps
1. Using master seed list, Find out Features list of the product from all the testcases
2. Associate testcases, defects to features
3. For each feature, Associate test cases to defects

The test cases are text documents having sections Title, pre-requisites, description( steps ) and expected result. As a first step, extract the features/sub-features from test case title and description fields. To get this, we need list of master seed features, using these master seed features list, other features and sub-features will be extracted.

Various key word extraction techniques are considered to perform feature/sub-feature extraction.

KEA: Key Phrase extraction algorithm (KEA) is used to extract the features from test case documents. For this, some of the test cases will be labeled with features, generate a model. All the remaining documents will be feed in to the model. This algorithm will generate a list of features/sub-features for that test case. From all the extracted list identify the features/sub-features of the product.

NLP: Used Natural Language Processing (NLP) based TAG POS, TF (Term frequency), IDF (Inverse Document Frequency), and other customizations such as Adjacency and inference mechanisms.

Once complete features are identified for the product. Mark all test cases and defects available in test bank and defect management with feature/sub-feature. Here there is a case, where a test case/defect is not associated with a feature. Mark all these test cases/defect by introducing a new feature “Unknown”.

Now we have test cases and defects associated with a feature. The next step is associate these test cases with defects. To perform this operation, text classification algorithm matrix regression[1] is used with different distance metrics.

As mentioned in the Text Categorization for Multi-label Documents and Many Categories [1], all the test cases available in the test bank are considered for training phase.

Matrix Regression – Training Phase: The input set of test case for training phase will be

$$C = \{ ID_1, ID_2, \ldots, ID_m \}$$

We need term vector to generate the Weighted matrix required to Matrix Regression. Each test case in the above table will be treated as a document associated with the concept. The terms available in the whole corpus will be generated by using text mining techniques like punctuation removal, number removal, special character removal, stemming and lemitization will be executed before generating the tokens from test case title and description fields. The T vector will be

$$T = \text{Total no of terms in the corpus} = \{ t_1, t_2, \ldots, t_n \}$$

By using the above data generate weighted matrix

$$W = \begin{bmatrix} W_{11} & \cdots & W_{1m} \\ \vdots & \ddots & \vdots \\ W_{n1} & \cdots & W_{nm} \end{bmatrix}$$

Figure 1. Term Document Matrix

Matix Regression – Scoring Phase: In this phase, get all the defects from defect management system to associate them to corresponding test case.
TABLE II. TEST CASE DB WITH FEATURE MAPPING

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>Defect ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defect title</td>
<td>Defect</td>
<td>DID1</td>
</tr>
<tr>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Defect title</td>
<td>Defect</td>
<td>DID2</td>
</tr>
<tr>
<td>Description</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE III. DEFECT DB STRUCTURE

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>Testcase Id</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title of the testcase1</td>
<td>Testcase</td>
<td>ID1</td>
<td>F1</td>
</tr>
<tr>
<td>Steps</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Title of the testcase2</td>
<td>Testcase</td>
<td>ID2</td>
<td>F2</td>
</tr>
<tr>
<td>Steps</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here each defect record will be treated as a document, classify the defect document to a concept(test case). For each defect document generate term vector \( T_0 \). Text mining techniques like punctuation removal, number removal, special character removal, stemming and lemitization will be executed before generating the tokens from defect title and description fields. The term vector

\[ T_0 = \{ td_1, td_2, \ldots, td_n \} \]

With the above term vectors, build a term vector

\[ F = T \cap T_0 \]

Replace each term \( T \) in vector \( F \) with value 1 and multiply the \( F \) and \( W \), \( W_1 = F \times W \), which gives a vector with weighted values. The highest value in the vector \( W_1 \) gets assigned to the corresponding document. To filter out false positives, taken a threshold value of 0.5. If there is a value less than 0.5 in the weighted vector \( W_1 \), the defect will not be assigned to any test case. This approach resulted many false positives as all the test cases and defects of the product are taken to calculate Test Coverage, Test Sufficiency. A variation of the above approach is described below.

In this approach, Matrix Regression algorithm is applied at feature/sub-feature level test cases. Two phases of Matrix Regression algorithm will be executed for each feature/sub-feature of the product.

Matrix Regression – Training Phase: As part of Feature extraction step, all test cases and defects are associated with a feature. Given the same concept vector as previous approach i.e.

\[ C = \{ ID1, ID2, \ldots, IDn \} \]

But instead of creating the term matrix with all the test cases available in the test bank, create the term matrix with the test cases associated to a feature.

Now the Term vector for a feature would be

\[ T_f = \{ tf_1, tf_2, \ldots, tf_m \} \]

With these vectors calculate the weighted matrix using tf-idf.

\[ W = \begin{bmatrix} w_{11} & \ldots & w_{1m} \\ \vdots & \ddots & \vdots \\ w_{p1} & \ldots & w_{pm} \end{bmatrix} \]

Figure 2. Feature Document Matrix

Matix Regression – Scoring Phase: In this phase, get all the defects belonging to feature and associate them with the testcases belonging to the same feature. In this way, we can reduce the number of false positives.

TABLE IV. DEFECT FEATURE MAPPING

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>Defect Id</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defect Title</td>
<td>Defect</td>
<td>DID1</td>
<td>F1</td>
</tr>
<tr>
<td>Description</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Defect Title</td>
<td>Defect</td>
<td>DID2</td>
<td>F2</td>
</tr>
<tr>
<td>Description</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here each defect record will be treated as a document, classify the defect document to a concept(test case). For each defect document generate term vector \( T_0 \). Text mining techniques like punctuation removal, number removal, special character removal, stemming and lemitization will be executed before generating the tokens from defect title and description fields.

The term vector, \( T_0 = \{ td_1, td_2, \ldots, td_n \} \)

With the above term vectors, build a term vector

\[ F = T \cap T_0 \]

Replace each term \( T \) in vector \( F \) with value 1.

Now multiply the \( F \) and \( W \), \( W_1 = F \times W \), which will give a vector with weighted values. The highest value in the vector \( W_1 \) will assign to which concept the document that belongs. To filter out false positives, applied a threshold of 0.5. If there is a value less than 0.5 in the weighted vector \( W_1 \), the defect will not be assigned to any test case. This refinement has given significant improvement in the results. The experimentation is further carried to study the behaviour approach with two different distance measuring techniques. The following distance measuring techniques are used.
Manhattan Distance: The Manhattan distance is advised in situations where for example a difference of 1 in the first variable, and of 3 in the second variable is the same as a difference of 2 in the first variable and of 2 in the second [16].

Euclidian Distance is standard distance metric in cartesian coordinate system.

Cosine Similarity: Cosine similarity gives the measure of how similar two documents are. The euclidean distance gives the magnitude of difference between the two documents. There could be a number of ways to combine the two and determine the similarity measure. There is a significant improvement in the results with Cosine Similarity.

IV. EXPERIMENTAL EVALUATION

We experiment this approach with an enterprise customer data having 849 test cases, 10,345 defects. In the defects there are 8,300 defects are internal defects i.e. found as part of testing phase and remaining are the defects raised by customers after software release. 53 feature/sub-features are taken as master seed list.

A. Feature Extraction and identifying new features and sub-features

Using master seedlist of 53 feature/sub-feature names Feature extraction step is executed against 849 test cases. This gave 60 potential feature/sub-feature names, where 31 are actual real feature/sub-feature names. With this new feature/sub-feature list, total feature/sub-feature list become 84 features and 29 sub features totalling to 113.

B. Associating test cases to feature

Once the feature list is comprehensive, the test cases are processed to assign a feature/sub-feature to the test case. Among the 849 test cases, only 30 test cases are not assigned to any feature. This shows that there are 30 generic test cases exists in the system.

C. Associating defect to feature:

Now the defects are processed to assign a feature to the defect. Among which 1500 defects do not belong to any feature and remaining are assigned to the feature. Now we have sufficient data points to find out test coverage and test case sufficiency. Over three patch releases the orphan defect numbers have come down to 150 thus showing the proposed system efficiency by multifold. At the beginning with close to 1500 orphan defects, sufficiency was about 85.5%. That has been improved to 98.55% over three releases.

V. SIGNIFICANCE AND IMPACT

‘Orphan defect’ is a significant find in the field of software testing as it exposes the insufficiency and incompleteness of test bank. Connecting test cases and defects is a unique proposition we bring to the table that cuts across three different teams (development team who develops the code based on requirements and design; testing team that test the product independently based on the shared requirements; and support team that collects and shares the customer reported defects) and saves significant time if they work together. This work is based on translating the report and reference documents of these teams into inferable features and associate test cases and defects based on thus extracted features. When this process has been executed across three patch releases of a product, impact is seen under three heads

• Test coverage improved significantly based on assigned features thus improved DRE
• Test case sufficiency is improved by adding additional steps covering the related sub features, thus again improving DRE
• Orphan defects have reduced by 75%. This increased the team confidence that the test bank they are developing is highly relevant and comprehensive.

ACKNOWLEDGEMENTS

We thank senior management of HCL for encouraging and allowing us to publish this work. We acknowledge the efforts of our development team without which this report would not have been a reality.

REFERENCES


