Test Case Comparison and Clustering using Program Profiles and Static Execution

Vipindeep Vangala
Microsoft Corporation
vipinv@microsoft.com

Jacek Czerwonka
Microsoft Corporation
jacekcz@microsoft.com

Phani Talluri
Microsoft Corporation
phanikt@microsoft.com

ABSTRACT

Selection of diverse test cases and elimination of duplicates are two major problems in product testing life cycle, especially in sustained engineering environment. In order to solve these, we introduce a framework of test case comparison metrics which will quantitatively describe the distance between any arbitrary test case pair of an existing test suite, allowing various test case analysis applications. We combine program profiles from test execution, static analysis and statistical techniques to capture various aspects of test execution and compute a specialized test case distance measurement. Using these distance metrics, we drive a customized hierarchical test suite clustering algorithm that groups similar test cases together. We present an industrial strength framework called SPIRiT that works at binary level, implementing different metrics in the form of coverage, control, data, def-use, temporal variances and does test case clustering. This is step towards integrating runtime analysis, static analysis, statistical techniques and machine learning to drive new generation of test suite analysis algorithms.

Keywords

Testing, static analysis, sustained engineering, machine learning

1. INTRODUCTION

Test suites for large software systems like Microsoft Windows contains millions of test cases that execute various scenarios to identify defects in the code or verify code compliance. Test suite development of such big products happen in a distributed environment. There are various problems that test teams encounter during maintenance phases which include: test selection, redundancy elimination and test prioritization. There are various tools and frameworks designed to perform these tasks [1,2]. All the industrial strength tools work on obtaining a minimal test suite that optimizes a specific criterion. And this criterion has been mostly code coverage which is not good enough to capture the functionality and deeper runtime aspects exercised on the tested binaries.

In this work, we have developed a framework which provides us a mechanism to quantitatively compare any arbitrary pair of test cases. To achieve this we have created six quantitative test case comparison metrics, which capture the deeper runtime behavior of what these tests are executing. We use program profiles, static analysis and statistical methods to compute these. We started this work with a single metric of code coverage to measure distance between two test cases. We learnt that such a simple metric cannot adequately represent all aspects of test case execution through the software under test and needed to add better measures of distance using several available information sources. Each measure allowed us to become more accurate in assessing the distance and show better functional diversity between tests.

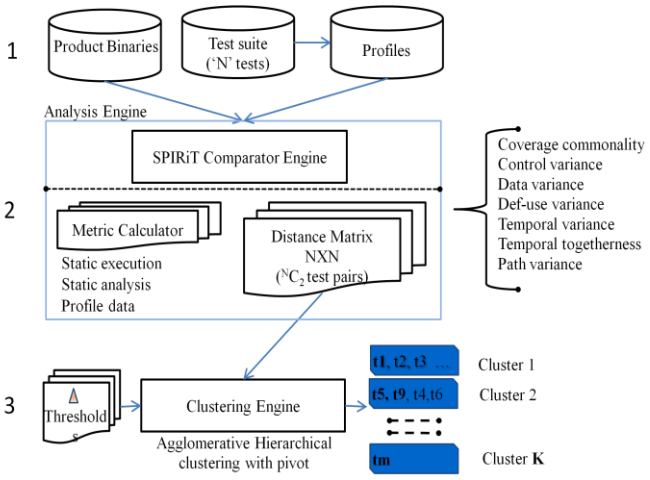

Figure 1: Architecture

We use program profiles that capture: code coverage information, number of times blocks/arcs were executed and temporal information in the form of time slices of execution (record which blocks are executed in a time interval). Increasing the amount of data collected at runtime increases the collection overhead and create timing related problems, hangs, blue screens etc. On the other hand, static analysis of executables or source code can help add information over execution profiles without incurring runtime penalty. We combine static analysis and runtime profiles in our framework which is called as: SPIRiT.

Our contributions include:

1. Quantitative test case comparison metrics.
2. Use of light weight program profiling, static analysis to capture the behavior of program being tested.
3. Use of clustering to group similar tests together.
4. Application of test case distance measurement for various test suite analysis problems.

2. OVERVIEW OF SPIRIT SYSTEM

SPIRiT framework contains 3 phases [Figure 1]. In first phase, we execute the test cases on instrumented builds of product binaries to collect program profiles. In the second phase, we simulate static execution using these test case profiles and calculate distance between every pair of test cases in test suite. This phase uses profiles, applies static and statistical analysis to compare all possible pairs of test cases and forms NxN distance matrix based on our metric computations. In the last phase, we apply customized clustering algorithm on the test cases to group similar test cases together. Clustering can be driven by specific thresholds to drive various applications like redundancy elimination, test selection, effectiveness of new test cases etc.

3. TEST CASE COMPARISON - METRICS

The following relationship scenarios are inferred for any test case pair T_i and T_j in a test suite apart from quantification in distance:

1. T_i is doing same testing as T_j : $T_i == T_j$
2. T_i is doing testing, which is subset of T_j : $T_i \leq T_j$
3. T_j is doing subset of testing of Ti : $T_j \leq T_i$
4. T_i and T_j are completely diverse: $T_i \neq T_j$

3.1 Quantifiable metrics

Quantitative test case comparison metrics suggest the amount of similarity between any test case pair, capturing what is being tested on the target binaries. We capture key aspects of program execution including: code coverage, counts of execution, data values, def-use of variables, and execution time among others. These are the aspects or features which drive our comparison metrics. We divide the program/binary into parts '*P'*, depending on the metric (Ex: control variance has points at control statements, def-uses on the chains etc). Signatures are calculated for each part '*P*' based on test execution and these *P-signatures* are used for comparing pairs of tests. Identifying program points where *P-signatures* can be gathered is independent of test case and depends on metric and binaries alone. *P-signatures* are calculated using standard deviation, variance, and maximal sub graph techniques that are well known in statistics and algorithms. Corresponding *P-signatures* are compared for each test case pair to compute statistical variance between test pairs over these, and do a normalized summation for overall metric value. We use static execution to simulate actual execution from profiles containing temporal data. Entire execution is divided and captured in the form of time slices which is cheaper on runtime. We start the simulation of two tests simultaneously from collected profiles and compute metrics.

3.2 Metrics: Test case pair comparison

Here we show how two test cases are compared to calculate distance between them and mention about few of our metrics.

3.2.1 Coverage Commonality

This metric is based on code coverage data, focusing on difference between any two test cases – T1 and T2.

$$\textit{Coverage Commonality} = \frac{\textit{Common blocks /arcs for T1 and T2}}{\textit{Total blocks covered in T1 and T2}} * 100$$

3.2.2 Control variance

This metrics computes the distance between test cases based on the difference in control statement executions. We record counts of block/arc execution in the binaries tested. Computation of this metric is illustrated in Figure 2.

3.2.3 Temporal variance

This metric computes the quantitative similarity in the time space for the test cases using available temporal information. We form a *P-signature* from temporal togetherness for every pair of blocks in the binary for every test case. We then derive a variance on these *P-signatures* across the two test cases being compared similar to Figure 2. In another variant we try to find out the variance between the number blocks that are temporally occurring in a time interval, across the two tests.

3.2.4 Def-use commonality

This metric captures the difference in the two test cases w.r.to the way they exercise different def-use pairs (*d-u*) of variables.

```
For each test case 't'
  For each program point 'b' IN
                        {branches}
    If ('b' has follower and target)
      mean = (taken + not_taken) / 2
      X = taken – mean;
      Y = not_taken – mean
      Deviation = (|X| + |Y|) / (taken + not_taken)

      P-Signature(t,b) = Deviation;
    End If
  End For
End For
```

Calculation of P-Signatures

```
For each test case pair (ti, tj)
  For each program point 'p' in
        {Program points=branches}
    Mean={P-Signature(ti,p)+
          P-Signature(tj,p)} /2
    Vaiance+={Square(Mean-
      P-Signature(ti,p))+
      Square(Mean-P-Signatur(tj,p))}/2
  End For
  Return variance;
End For
```

Calculation of Metric value

Figure 2: Control variance

$$\textit{D-U Commonality} = \frac{\textit{Common d-u pairs executed in T1 and T2}}{\textit{Total d-u pairs executed in T1 and T2}} * 100$$

3.2.5 Other metrics

We compute other metrics like- data variance: which captures the data values as proportional to the number of times the control statements are executed, execution time, type of test cases etc. Qualitative values are used to identify better test case when compared to another. This can be used for selecting good test when clustering is applied.

3.3 Combining metrics

From individual metrics for each test case pair, we form an aggregate quantifiable metric that shows comprehensive distance between test cases:

$$\textit{Dist (Ti, Tj)} = \sum_{k=1}^{m} \frac{W_k * \textit{Dist}_k(T_i, T_j)}{m}$$

Where $Dist_k(T_i, T_j)$ represents the value of k^{th} metric

W_k represents the weight for the k^{th} metric that can be varied for practical purposes or learned as system evolves.

4. RESULTS AND CONCLUSION

We used our test comparison metrics to drive clustering algorithm with aggressive thresholds for redundancy detection on some legacy test suites of Microsoft Windows and some open source software. Sizes of test suites we used for evaluation varied from hundreds of tests to thousands for diverse products. On an average, we identified 10-20% of redundant test cases with high accuracy of around 70%. We were able to use SPIRiT to select variable subset of diverse test cases that would suit in time constrained scenarios and trained optimizations. Our metrics gives around half the false positives compared to code coverage based methods in redundancy detection. Our experimental data shows that we can indeed combine various methodologies to create a powerful classification system for test suite to find redundancy, help prioritization among various other applications. This approach is indeed better than traditional code coverage methods giving lesser false positives, preserve the failure inducing cases and guaranteeing the diversity in subset selection. Our approach can be applied to find effectiveness of new test, find failure variants, change analysis and other test suite analysis problems.

5. REFERENCES

[1] A. Srivastava, J. Thiagarajan, " Effectively Prioritizing Tests in Development Environment", ISSTA 2002.

[2] M. J. Harrold, "Testing Evolving Software", Journal of Systems and Software, vol. 47, , pp. 173-181, Jul.1999.

[3] A. Srivastava, A. Edwards, and H. Vo, "Vulcan: Binary transformation in a Distributed Environment", Microsoft Research Technical Report, MSR-TR-2001