International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868
Foundation of Computer Science FCS, New York, USA
Volume 2– No.2, May 2012 – www.ijais.org

A Productive Method for Improving Test Effectiveness

Saran Prasad
Cadence Design Systems
India Pvt. Ltd.

Mona Jain
Cadence Design Systems
India Pvt. Ltd.

Shradha Singh
Cadence Design Systems
India Pvt. Ltd.

C.Patvardhan
Department of Electrical Engineering, Dayalbagh Educational Institute
Dayalbagh, Agra

ABSTRACT

Automated testing of software products has greatly expanded over the past few years. Ever increasing test suites have been developed, along with the computing infrastructure to support them. While the capacity for testing has grown, the environment is not infinitely scalable - eventually capital spending is capped. Methodologies need to be explored that improve the overall effectiveness of the test cases that are run. Furthermore, these methodologies need to be as independent from the test suites as possible: the size of the test suites render solutions that are tightly bound to them ineffective for widespread utilization. Problem associated with these huge numbers of test cases is that whenever the code is changed, the entire suites of test cases need to be run.
One idea is to run fewer tests on an ongoing basis, reserving full regression test runs for key milestones in the development lifecycle. This is workable if the limited tests produce a similar result in the short term.

In this paper, we present a new approach for test suite selection that focuses on improving test effectiveness. The methodology described produces a pruned list of test cases required to test an application. The method has three components, the predictive component which makes use of statistical data, coverage based method digs the delta from the code to produce a pruned list of test cases, and decision based technique that prioritizes important test cases. Our experiments show that our approach results in a better utilization of compute resources and also decreases validation cycle thus reducing time to market.

General Terms

Testcase prioritization and selection

Keywords

Software testing, regression testing, test case prioritization, test-case selection..

1. INTRODUCTION

Testing is the dominating verification technique used in industry today, and many man-hours and resources are invested in the testing of software products. Research has shown that at least 50% of the total software cost is comprised of testing activities. Companies are often faced with lack of resources, which limits their ability to effectively complete testing efforts. To cut down the cost of testing, automated test execution becomes more and more popular. However, which test cases to be run first i.e. the selection of which tests to be executed is still mainly a manual process that is error prone, and often without sufficient guarantee that the system will be systematically tested. In many a cases, instead of test case selection whole regression test suite is run whenever a new piece of functionality needs to be tested. This is computationally expensive task as well as wastage of resource time and effort. There is no point in running all testcases to test a small enhanced feature or functionality. Similarly to perform systematic regression testing is to ensure that the tests satisfy a required criterion. The criteria could be coverage criteria or it could be the fulfillment of some requirement or it may be the capturing of bugs. The stop criterion used is either available testing time or a sufficiently stable product. Whatever criterion is taken, the test process should be as efficient as possible to remove as many defects as possible i.e. methodologies need to be explored that improve the overall effectiveness of the regression testing.

One solution could be to run fewer tests on an ongoing basis, reserving full regression test runs for key milestones in the development lifecycle. This is workable if the limited tests produce a similar result in the short term.

Alternatively a Product Validation Engineer's experience can be used to identify test cases that need to be run to test specific functionality. This methodology however, is heavily dependent on the experience of the Product Validation engineer and his familiarity with the code and is more prone towards introducing serious bugs.

The key solution is test case selection and prioritization mechanisms. Therefore, there is a requirement of a test case selection mechanism that can be used to select a list of test cases from the given ones.

This paper presents a new tool for test suite selection and prioritization that focuses on improving test effectiveness. The tool functions in such a way that it produces a pruned list of test cases required to test an application from the given regression test suite. The tool operates in three modes:

- Predictive method which makes use of statistical data
- Coverage method digs the delta from the code to produce a pruned list of test cases
- Decision based method that prioritizes important test cases.

Our experiments show that the usage of this tool results in a better utilization of compute resources and also decreases validation cycle thus reduces time to market. Results indicate that test case selection and prioritization can significantly improve the rate of fault detection of test suites.

2. PROBLEM STATEMENT

Test engineers usually write new test cases to test new functionality or feature in the software, and add them to the existing test suite. As a result, these test suites grow in size with the constant addition of test cases. In many practical scenarios there is the absence of traceability matrix which maintains testcase to requirement mapping. Ideally a tester should review this matrix before adding the testcase into main test suite, but in the absence of this mapping, testers simply write and add test cases to the suite. Old test cases are not reviewed before adding a new test case which may lead to redundancy. Multiple test cases may exist in a test suite which may satisfy the same requirements. There may be multiple set of two or more testcases which may collectively satisfy same requirement. This again leads to redundancy and due to existence of redundant testcases; size of the test suite grows tremendously. And this is the point where problem arises. Large test suite size is the pain area because test suite execution can be very expensive both in terms of compute resources as well as human resource time. More human resources are needed to evaluate the failures and do root cause analysis. There is wastage of resources, time, effort and money by running too many test cases every time without gaining any code coverage or capturing bugs. Large test suite size is the pain area because test suites are run on servers; they utilize compute resources. Test cases may also need human intervention to check the output and set up other machinery. So having too many test cases to run can be very expensive. If the same problem is viewed from a developer's perspective then practically, in any software company multiple developers work on a project. These developers work on the code base having millions lines of code which are maintained using version control mechanism. It is also a common practice to check in updated code into repository after regression test suites is run. For a code base having million LOC, test suite also contains million test cases. For a larger test suite, execution time is longer and developers are required to wait for a longer time for the code to be checked in.

Execution of whole regression test suite is thus, major problem during regression testing and there is a requirement of techniques or mechanisms which can pick test cases selectively from the regression test suite as well as prioritize test cases in the given test suite. Those which are more prone to failure should be run first rather than running them all every time.

3. LITERATURE REVIEW

As per literature there are methodologies that are related to regression testing. There are four methodologies that are available for regression testing. These methods are [2, 4, 5]

- Retest all
- Regression Test Selection
- Test Suite Reduction
- Test Case Prioritization

[10] Test case prioritization is a method to prioritize and schedule test cases. The technique is developed in order to run test cases of higher priority in order to minimize time, cost and effort during software testing phase.
Rothermel [11], [12] gave an interesting example as follows: “one of the industrial collaborators reports that for one of its products that contains approximately 20,000 lines of code, running the entire test suite requires seven weeks. In such cases, testers may want to order their test cases so that those test cases with the highest priority, according to some criterion, are run first”. This has proven that prioritizing and scheduling test cases are one of the most important tasks during regression testing process.
Additionally, Rothermel [11], [13] mentioned that the test case prioritization process is required for software testing because: (a) the regression testing phase consumes a lot of time and cost to run, and (b) there is not enough time or resources to run the entire test suite, therefore (c) there is a need to decide which test cases to run first.
The literature review shows that many researchers propose many methods to prioritize and reduce the effort, time and cost in the software testing phase, such as test case prioritization methods, regression selection techniques and test case reduction approaches. As per [10] there are many research challenges and gaps in the test case prioritization area. Those challenges and gaps can give the research direction in this field. However, the research issues that motivated this study are:

- No existing prioritization techniques address the problem of multiple cases with same weight values. The existing test case prioritization techniques use a random approach to prioritize those cases to resolve that problem. The problem may lead to a poor performance of an ability to prioritize and schedule test cases.
- Existing test case prioritization techniques assume explicitly that there is only a single test suite. The test suite is a collection of a set of test cases. There are no prioritization techniques to resolve the problem of multiple test suites.

Siripong Roongruangsuwan, Jirapun Daengdej in [10] proposes two methods to resolve the above research issues. The first method aims to improve the ability to prioritize a set of test cases in case that there are multiple cases with the same priority weight values. The second method is developed to prioritize multiple test suites, which they contains a set of test cases.
Rothermel at el. [2, 3] defines the test case prioritization problem as follows:

Given: T, a test suite; PT, the set of permutations of T; f, a function from PT to the real numbers.

Problem: Find T’ belongs to PT such that (for all T”) (T” belongs to PT) (T” $\neq$ T’) [f (T’) $\geq$ f (T”)].

Here, PT represents the set of all possible prioritizations (orderings) of T and f is a function that, applied to any such ordering, yields an award value for that ordering [2,7].

An automatic strategy to test case selection was presented by Emanuela G. Cartaxo, Francisco G. O. Neto, Patrıcia D. L. Machado in [6]. The strategy is based on similarity between test cases. The main goal of this strategy is, by observing the similarity between test cases, to minimize redundancy and assure adequate transition coverage.
A research work by C. Jard and T. Jeron [7] explains a Test Generation with Verification technology (TGV) tool which is a conformance test generator. This tool selects test cases from model. The test cases are selected from a test purpose, that is a specific objective that a tester would like to test, and can be seen as a specification of a test case. Even though a test purpose targets the test at a particular functionality, reducing the final test suite size, the result can still be a huge exhaustive test suite.
Another research work by F. Basanieri, A. Bertolino, and E. Marchetti describes “The Cow Suite” tool which derive test

cases from UML sequence diagrams and use case diagrams [8]. Their test generation algorithm is exhaustive. For each diagram a weight function indicating the functional importance is attributed. This way the test case selection strategy chooses the most important set of test cases.
SPACES [9] describe another research work. It is a tool for functional component testing. In this tool, weights are associated to the model's transitions. According to the weights the most important set of test cases are selected.
In literature there is a description for Cost effective-based techniques. [10] These are methods to prioritize test cases based on costs, such as cost of analysis and cost of prioritization. Many researchers have researched this area, for instance, Malishevsky [14], Alexey [15], and Elbaum [16].
The objective of this research is to develop a test case prioritization technique that prioritizes test cases on the basis of execution result history i.e. pass/fail history of test cases which is predictive model. Another model is coverage model which generates a list of affected test cases due to change in source code file. Third one is weight model which generates a list of test cases on the basis of weight which is assigned to each testcase. Details are given in next sections.

4. NEED FOR TESTCASE SELECTION AND PRIORITIZATION

Testcase selection and test case prioritization problem is also a computationally expensive problem. In order to overcome the computational complexity of the problem in hand, we need to identify a test suite selection and prioritization technique which can select and prioritize test cases from a given regression test suite within a reasonable amount of time with an acceptable degree of accuracy. Test Suite selection and prioritization should happen in such a way that the selected list of testcases should capture same number of faults as the original test suite. The technique should be capable to operate on real testing environment with acceptable performance.

5. OVERVIEW OF OUR APPROACH

Software testing and retesting occurs continuously during the software development lifecycle. As software grows and evolves, new test cases are generated and added to a test suite to exercise the latest modifications to the software. Since the number of test cases is huge one requires high compute power and longer cycles for these regressions to finish. This paper presents a productive method for improving test effectiveness.

Test cases either produce a positive (pass) or negative (fail) result. While both results are important in that key information is gained, test failures are more useful because defects in the software and/or test are positively identified. Further action can be pinpointed by the test failure to improve the software and/or the test. Failures yield an actionable item.

When resources are constrained and only a limited set of test cases can be invoked, then those that are more likely to produce failure results should be selected.

The productive methodology on which our tool is based has three main components that work independently or in combination to produce a list of testcases which are more likely to fail.

- Predictive mode: It is Statistical data based technique makes sure that test cases that failed most are more likely to run.
- Coverage mode: It is Coverage based technique which works on capturing code coverage tool data and using it to map parts of code with test cases.
- Weight mode: It is Decision based technique that works on an optimal algorithm which prioritizes most important test cases to run first.

It is capable of operating in live testing environment in reasonable amount of time. Tool will not delete the test cases from the test suite; rather it will present the new pruned list of test cases to user for execution.

6. DETAILED DESCRIPTION OF OUR APPROACH

6.1 Predictive Model (Statistical Analysis Methodology for Test Case Selection)

The The fundamental premise of this method is that the historical failure behavior of individual test cases can be used to govern the frequency of test invocation. Those tests that fail more frequently are run more frequently, while those that generally pass are run less frequently. This increases the effectiveness of test cases by reducing the resources needed to yield a statistically similar test outcome.

This method does not suggest that only limited test runs be invoked - in fact, periodic re-sampling is an important part of this method.

This method has four general parts:

- The characterization of failure behavior.
- Accounting for change.
- The mapping of failure behavior to action.
- The correlation of usage of this method to results..

6.1.1 Charaterization of Failure Behavior

For The characterization of the failure behavior of any one test can be stated as:

$C = p/d$.

Where p is the probability of failure of the test, and d is the failure distance of the test. p is calculated by the test's previous track record:

$p = f/t$

f represents the number of failures.

t represents the total number of test runs.

For example, if a test x has failed 1 time out of 30 attempts; its probability of failure is 1/30, or roughly 3.3%.

Normally, this would be a sufficient characterization of the failure behavior. However, regression tests are running in an environment where the software and the tests are constantly changing. Therefore, the failure distance needs to be factored into the characterization.

Failure distance is the notion that more recent failures for a test indicate a higher need to exercise the test. Recent failures could be the start of a trend of failures for the test. Taking the previous example of 1 failure out of 30 attempts, it is a much different characterization if the failure occurred the first time the test was run, than if it failed during the most recent test run.

Therefore, more recent failures (those have a near distance) should induce test invocation more than those that are further in the past (those having a far distance).

While a simple temporal calculation appears sufficient to measure the failure distance, it is not. Determining the delta between the present and the point in time when the test last failed does not account for the possibility that the test hasn't been run in the intervening time period. Rather, the measurement of failure distance is the number of consecutive successes of the test, regardless of when the test was run.

Continuing the previous example, the case where the failure occurred the first time the test was run would have a failure distance d of 29. The second case would have a failure distance d of 0.

The failure characterization C is represented by a number where larger numbers signify a greater need for invoking a test. A failure distance of 0 is treated as special case, meaning always run the test.

This implements the notion of "When in doubt, run the test."

6.1.2 Accounting for change

When regression testing software, there are three areas which affect the results:

- The software itself
- The test
- The environment

When a test fails, the cause may be in one (or more) of these areas. This variability of causality is further compounded since these three areas are not static - they are constantly changing.

The predictive method reduces the number of tests run by focusing on the failure characteristic of the test. However, it needs to account for changes in the software, test suite and environment that may cause new failures. This is done via re-sampling.

Re-sampling is the act of invoking the test regardless of its failure characteristic. This refreshes the calculation of the failure characteristic, insuring that failures due to change are not undetected.

There are a number of ways to govern the re-sampling of tests. The simplest is an interval based mechanism for example, re-sample all tests every nth test invocation, this means run all the tests that were not run for last 'n' times.

For example, assume 3 is the re-sampling interval of test invocation. A test x was not run for 3 time out of 10 attempts. Based on Interval based re-sampling mechanism, the test will be selected to run in the next cycle.

In essence, the re-sampling control mechanism is the binding function between the predictive method, and dependency models.

6.1.3 Mapping to Action

The process for using the failure characterization is a two step process for each test suite:

- Calculate the current failure characterization (C) for each test in the test suite.
- At the point of running each test, compare its failure characterization (C) with an input threshold.

The threshold value itself is quite simple: it is an arbitrary number that C must be greater than if the test is to be run. The greater power lies in the flexibility.
Threshold values can be calculated from the user defined maximum and minimum failure distance values for a test suite.
Th=100/d(d+1)
Th represents threshold
d represents failure distance.

Maximum threshold value is calculated from minimum failure distance and minimum threshold value is calculated from maximum failure distance. Above formula is derived from failure characterization formula only.

6.1.4 Correlation to Results

This method attempts to look at the history of any test, and predict whether or not the test should be run. This method is successful if it can limit the run tests to those that fail. This is difficult to measure since the outcome of the tests not run isn’t known.
Still, there are ways to determine the viability of this predictive method. The data can be surveyed, and predictions made at particular points in the test history. Then, since the outcome is known, the method can be measured as to its effectiveness.

6.2 Coverage Model (Delta Methodology for test case selection)

In real testing world, coverage analyzers are normally used to make sure that the program source code is completely covered by the test cases in the regression test suite and no part is left. Coverage could be statement coverage, function coverage, branch coverage and call stack coverage. Mostly statement and function coverage is captured by real world testing teams by making use of coverage analyzer. One such coverage analyzer is Rational Purecov. Our tool currently supports Purecov. Purecov generates output in the form of .pcv file. It contains source file name, line number and number of times the line got executed during test case execution. Our tool’s coverage model makes use of Purecov output files for individual test cases and utilizes them for functionality driven coverage.
It reads each coverage data file and captures all source file names and their lines with hit count >0. It then captures relevant information like total lines, used lines, file names and stores this information into a database against the test case which got executed. In a way it maintains test case to file mapping. It also maintains filename to coverage string mapping. One coverage string is created for each file. It is

actually the concatenation of coverage string for every function within the file.

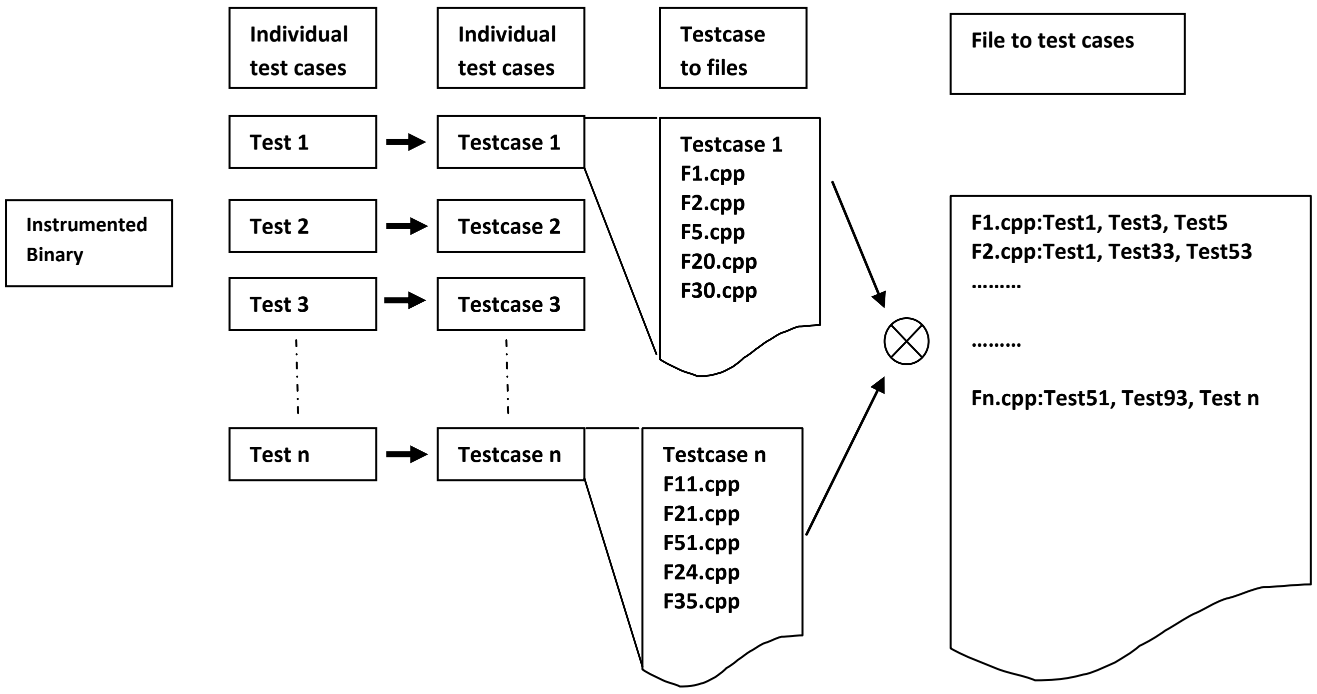

Fig 1: PMT Coverage

6.2.1 *Coverage String*

Coverage string for a function is created as line number followed by "," followed by total number of lines which got hit followed by#. Coverage strings of two functions are concatenated by "##". Thus, coverage string for a file is the concatenated coverage string of various functions within the file. For example: 23,2#37,4##53,1#82,5##
Thus if a file file1.cpp has two functions f1 and f2, then this coverage string shows:
In function f1 line numbers 23, 24, 25 got hit. In function f2 line numbers 53, 54 got hit and in function f3, line numbers 82, 83, 84,85,86,87 got hit during test case execution.

Whenever users want to extract testcases from the database corresponding to touched .c/.cpp files, they will just have to run a simple command which in terms of PMT is known as "Query" to the system. This query extracts testcases from database based on testcase to file mapping and corresponding coverage string. Thus query generates a list of affected test cases due to change in source code file.

6.3 Weight Model (Decision Based Approach for Test Case Prioritization)

Our tool uses "Decision based method" for test case prioritization. Test case weight is considered as prioritizing attribute for this method. The algorithm used, first calculates weight for each test case. In our case, "Weight" is derived on the basis of number of bugs captured by the test case and test case execution time. Our tool makes use of a Change Management System which is used to file Change Requests against a product. Each Change Request is given a CCR Id and it corresponds to a bug filed against a product. Tool makes use of this mapping between test cases and CCR Ids or bugs captured. Testcases are categorized into 3 different test suites.

6.3.1 *R&D suite contains "High" weight testcases*

Testcases whose weight lies in high range are added to R&D suite.

6.3.2 *Daily Yellow suite contains "Middle" weight testcases*

Testcases whose weight lies in middle range are added to Daily Yellow suite.

6.3.3 *QA test suite contains "Low" weight testcases*

Testcases whose weight lies in middle range are added to QA test suite.
Each test suite has a pre-defined maximum run time. User can request the type of the test suite they would like PMT system to generate. Once weight calculation is done and based on the user's request, run time of each test case is compared with the suite's maximum run time. Formula used is:
Tr < Tmax
Or
Tmax<Tr<1.5Tmax
Where:
Tr is test case run time.
Tmax is test suite maximum run time.

All those testcases satisfying either of the above 2 equations are selected for the next cycle. A weight sorted list of the selected test cases is returned to the user.

7. IMPLEMENTATION OVERVIEW

The Productive Method for Testing System (PMT) has the following architectural aspects:

- API based independent system
- Client/server architecture
- Portability features
- GUI frontend for administration and reporting

7.1 API Based Independent System

A key concept of the predictive method is that it is decoupled from the tests: the only linkage with the tests is the pass/fail history. By providing an independent system, many benefits may be realized, including:

- Wider deployment opportunities
- Ease of integration with existing test systems
- Ease of enhancement
- Better performance management

The PMT system is broken into two parts: a common/generic processing engine, and an API. The processing engine contains a database, holding all the pertinent data concerning a test's pass/fail history, cpu run time, CCRs filed, mapping with souce code file along with the line numbers. The processing engine performs all of the calculations necessary for tests based on the technique used.
A web interface to the processing engine is also provided. This allows an interface for performance tuning, as well as centralized reports.
PMT has a modular architecture based on plug-in model. Test systems access PMT via shell and/or plug-in wrapper. In essence, the test systems only view the PMT system through the shell/plug-in wrapper abstraction: no knowledge of the underlying system is necessary. A Unix shell wrapper over java API can easily be called by any test system. Predictive method API is directly invoked by unix shell wrapper. Plug-in wrappers are written for Coverage based and Decision based methods. These wrappers first extract test case data from their regression hierarchy and then calls unix shell wrapper to invoke PMT API.

7.2 Client Server Architecture

The PMT system uses client/server architecture. A centralized machine hosts the processing engine. Centralized hosts can be deployed in a site or group specific basis.
Client/server architecture allows for better management of the PMT system. It keeps PMT processing separate from the test systems, reserving processing capacity on the test systems for testing.
PMT system has a server and various clients. Server runs on the PMT host machine as a background process, which is setup for a group or site or can be a centralized machine. This process will run always and listen to client requests. Clients can be installed for various test setups & they can send requests to the server. Client server architecture here uses XML-RPC with HTTP as the transport protocol and XML as the encoding.

7.2.1 Portability Aspects

Since the PMT system must be able to service a wide variety of test systems, portability is a key concern. The PMT system provides portability through its independent, UNIX shell wrapper based architecture. The implementation details (i.e. language and OS) in no way affect the test systems using it.
The UNIX shell wrapper, calling PMT API is Java based, employing an XML-RPC /HTTP communication protocol with the processing engine. Given the API's abstraction layer, it can be re-implemented in different languages and for different OS's as necessary.

7.3 Extensions to Predictive Concept

There are certain extensions to basic predictive concept. They are:

- One Port Logic
- Priority Based Sorting
- Run Always/Run Next

7.3.1 One Port Logic

The Predictive Method is not perfect in the sense it cannot guarantee every failing test will always be recommended to user for review. There is a fair probability for some testing teams that PMT system may not identify all failing tests.
The "One Port Logic" mechanism ensures that all tests are run on at least one platform.

7.3.2 Priority Based Sorting

The Predictive Method is based on the notion that failure of a test is valuable, actionable information. A natural extension is to consider that there's an ordering of failure information – the failure of some tests is more important than the failure of others.

To handle this, a sorting scheme was devised. The PMT can sort test cases based on their C value. Test systems can then choose to run tests having a higher probability of failure first. In a constrained test system this allows engineers to get failure results sooner, so remedial action can start.

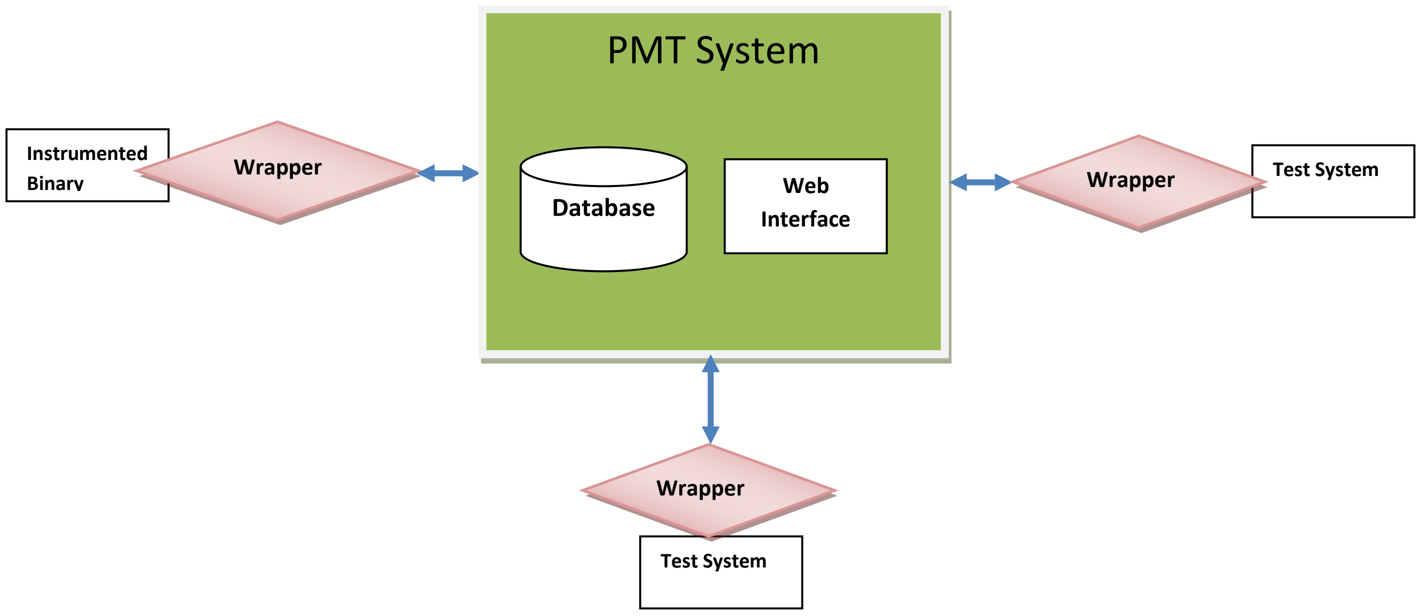

Fig 2: PMT System: Top Level Architecture

7.4 Run Always/Run Next

Sometimes it is advantageous to run a test regardless of its failure characterization. The Predictive Method provides the "Run Always" and "Run Next" features to handle this situation.
The "Run Always" attribute if set for a test case disables the operation of PMT on that test case and that test case will always be recommended by the PMT system.

The "Run Next" attribute if set for a test case ensures that the testcase will be recommended by the PMT system, regardless of the failure characterization calculated by the PMT – but this will happen only for the next singular invocation of the test. This is effectively a temporary "Run Always" marking.
The "Run Next" feature is useful for key test runs in the development lifecycle, such as prior to a release or major milestone.

8. DESCRIPTION IN DETAIL

8.1 Data Elements

The primary data element in the PMT system is the test object or test case. The test object is an abstraction of the notion of a test. By providing an abstraction layer, the PMT system allows for a great deal of flexibility in considering what might be a "test": test objects can be used to represent individual tests, test suites (banks or groups), or sub-divisions of individual tests.
Test cases are uniquely identified by a set of attributes. This set is flexible, and can vary depending upon the requirement of the user:
Test case information can be attributes such as:

- Full storage path (It can be path to the test case directory or it can be path of a file having absolute path of test cases).
- Product
- Release
- OS/Platform (Linux, Solaris, AIX)
- OS Bit (32/64)

A number of data attributes are associated with each test case.

8.1.1 Test Case Attributes for Predictive approach:

- Failure distance (Threshold) levels
- Skip levels
- Test result history
  - Number of test invocations
  - Number of failures

Threshold levels are the trigger value for determining whether or not a test object should be considered for recommendation to user.
Skip levels are the number of times the PMT system is permitted to not run a test object consecutively.
The test result history is an ordered set of pass/fail results for the test object. This includes the number of times the test object was invoked, the number of times it failed and the number of times it did not run i.e. it did not come for registration. From this information, the failure characterization of the test object is calculated, including the failure distance.

8.1.2 Test Case Attributes for Coverage approach:

- Source code file (.c/.cpp file)
- Line coverage
- Pass/fail status
- Run time

8.1.3 Test Case Attributes for Weight approach:

- CCR Ids (Change Request Ids generated by the Change Management System. Change Requests are used to show an enhancement in the system under test as a result of bug captured by the test case). CCR Ids are used to maintain a mapping between the test case and the number of bugs captured by the test case.
- Run time: This is test case execution time.

8.1.4 Processing

The following are the main elements of the PMT system:

- The API
- Calculations & Logic

The API
PMT API is very simple. PMT system interacts with the test system at two points:

- PMT API Query
- PMT API Registration

Although test systems may vary, the query interface is used whenever user requires a PMT recommended list of test cases for execution.

In order to update the PMT system database with the test case information, the registration interface is used.

Calculations & Logic

The Predictive system calculates the failure characterization (C) for each test object using the basic mathematical calculation. This is based on the pass/fail/did not run data.

In order to determine whether or not to run test objects the Predictive system employs threshold levels. There are two levels – an upper and lower threshold. The logic used is:

C > th1 –> run the test
th2 < C < th1–> run every Sk2 time
C < th2 –> run every Sk1 time

Where:

C represents the failure characterization
th1 is the upper threshold
th2 threshold is the lower threshold
Sk1 is the upper skip limit
Sk2 is the lower skip limit

9. PREREQUISITES TO USE OUR APPROACH

In order to use the PMT system in an already exiting real world testing environment, the test system must be able to do the following:

- In order to use the Coverage model, code coverage data for individual test cases is required. PMT system currently supports Rational's Purecov but system is configurable and can be enhanced to support any other Coverage analyzer.
- Test cases should be defined based on some key attributes for example:
  - Testcase name
  - Unique key (Any value that can identify uniqueness of a test case)
  - OS/OS Bit
  - Pass/Fail Status
  - Product Name/Release
  - Test Banks: This attribute is used to maintain the hierarchy within test cases (if any).
- PMT system requires all these attributes for each test case in the form of an input file.
- PMT system provides result on a per test object basis and writes the recommended list into a output file as per the requirement of test system.

10. FEATURES

10.1 Load balancing

In order to balance the load on server farms i.e. for equal distribution of test cases on server farm for execution, a load balancing feature exists in PMT system. By enabling this feature the system internally does some processing and divides the recommended list of test cases into different chunks. In this way the system returns a chunk, having approximately same number of tests to the user for some consecutive days.

10.2 Deregister Last Consolidated/Registered Run

Using "Deregister" mechanism user can deregister the last consolidated or registered run from the system.

10.3 Inputs for max and min Failure Distance

User can provide these values as input which is used to calculate threshold range.

11. EXPERIMENTAL STUDY

This section discusses an evaluation result of our approach. Our predictive model is capable to operate in real world testing environment. It gives best results in more stable software products. For the software code base which changes frequently we would recommend to use coverage model.
Savings by Priority sorting only (Time saved) : 1.5 hours saved per run, out of 7 hours required to collect & analyze all failures.
Predictive model Savings (i.e. – percent fewer tests run): 61% to 73% while finding all bugs.

12. CONCLUSION

Reducing cost and duration of a test phase is of utmost importance to stakeholders. The capabilities of the testing team can greatly affect the success, or failure, of the testing effort. An effective testing team not only includes a mixture of technical and domain expertise but also efficient testing techniques and tools necessary to perform the actual tests. The capabilities of the testing teams can be greatly enhanced by the usage of PMT System. By performing test case selection and test case prioritization, the costs of executing, validating, and maintaining test suites over future releases of the software can be greatly reduced. Our approach results in better utilization of compute resources and also decreases validation cycle thus reducing time to market. The statistical approach of our PMT system proves better for testing of stable products and their test suites, however coverage based approach proves better for frequently changing test suites.

13. REFERENCES

[1] Praveen Ranjan Srivastava, TEST CASE PRIORITIZATION. Computer Science and Information System Group, BITS Pilani, India-333031. Journal of Theoretical and Applied Information Technology, ©2005-2008 JATIT. All rights reserved. Link: www.jatit.org

[2] S. Elbaum, A. Malishevsky, and G.Rothermel Test case prioritization: A family of empirical studies. IEEE Transactions on Software Engineering, February 2002.

[3] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold, "Prioritizing Test Cases for Regression Testing," IEEE

Trans. Software Eng., vol. 27, no. 10, pp. 929-948, Oct. 2001

[4] Aditya P.Mathur, Foundation of software testing, Pearson Education 1st edition.

[5] Maruan Khoury, Cost-Effective Regression Testing, 2006.

[6] Emanuela G. Cartaxo, Francisco G. O. Neto, Patrıcia D. L. Machado Automated Test Case Selection Based on a Similarity Function,{emanuela,netojin,patricia}@dsc.ufcg.edu.br.

[7] C. Jard and T. J´eron. TGV: theory, principles and algorithms, A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. Software Tools for Technology Transfer (STTT), 6, 2004.

[8] F. Basanieri, A. Bertolino, and E. Marchetti. The Cow Suite Approach to Planning and Deriving Test Suites in UML Projects. In UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. LNCS. Springer, 2002.

[9] D. L. Barbosa, H. S. Lima, P. D. L. Machado, J. C. A. Figueiredo, M. A. Juca, and W. L. Andrade. Automating Functional Testing of Components from UML Specifications. Int. Journal of Software Eng. and Knowledge Engineering, 2007.

[10] SIRIPONG ROONGRUANGSUWAN, JIRAPUN DAENGDEJ. TEST CASE PRIORITIZATION TECHNIQUES. Journal of Theoretical and Applied Information Technology, © 2005 - 2010 JATIT & LLS. All rights reserved. www.jatit.org.

[11] B. Korel and J. Laski, "Algorithmic software fault localization", Annual Hawaii International Conference on System Sciences, pages 246–252, 1991.

[12] Cem Kaner, "Exploratory Testing", Florida Institute of Technology, Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, FL, 2006.

[13] Dennis Jeffrey and Neelam Gupta, "Test Case Prioritization Using Relevant Slices", In *Proceedings of the 30th Annual International Computer Software and Applications Conference*,Volume 01, 2006, pages 411-420, 2006.

[14] A. Srivastava, A. Edwards, and H. Vo., "Vulcan: Binary Transformation in a Distributed Environment", Microsoft Research Technical Report, MSR-TR-2001-50, 2001.

[15] Alexey G. Malishevsky, Gregg Rothermel and Sebastian Elbaum, "Modeling the Cost-Benefits Tradeoffs for Regression Testing Techniques", *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, 2002.

[16] James A. Jones and Mary Jean Harrold, "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage", In *Proceedings of the International Conference on Software Maintenance*, 2001.