

**Automatically Finding Abstractions for Input Space Partitioning For
Software Performance Testing**

BY

ASWATHY NAIR
B.Tech., Anna University, 2008

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Masters in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2011

Chicago, Illinois

Defense Committee:

Dr Mark Grechanik, Chair and Advisor
Dr Ugo Buy
Dr V.N Venkatakrishnan

Copyright by

Aswathy Nair

2011

This thesis is dedicated to my mother for her unwavering support and motivation and to my father who made me believe that all our dreams can come true.

ACKNOWLEDGMENTS

First and foremost, I want to express my sincere gratitude to my advisor Dr. Mark Grechanik, without whose patience, motivation and support this thesis would not have been possible. He taught me how the basic principles of software engineering are applied in practice. I thank him for all the contributions of ideas, funding and time which helped me make my graduate school experience productive. Being able to work so closely with Dr. Grechanik and creating a very good professional relationship with him has been one of the key takeaways from my graduate school experience.

Besides my advisor, I would also like to thank members of my thesis committee, Dr. Ugo Buy and Dr. Venkatakrishnan for their encouragement and support.

I am greatly indebted to Dr. Qing Xie from Accenture Technology Labs who assisted me with the automated test script generation using JMeter. I would like to thank Dr. Chen Fu from Accenture Technology Labs who helped me set-up the profile repository to collect the execution profiles. It was a pleasure working with such good researchers whose enthusiasm for research was contagious and motivational for me. I also thank Kalyani Balakumar who worked with me on the test script generation part.

Lastly, I would like to thank my family for all their love and support. I thank my parents, who supported me in all my pursuits. I thank my sister, Revathy, for all the love and encouragement. And most of all, I thank my loving, supportive husband, Abilash for all the love, patience and encouragement.

AN

TABLE OF CONTENTS

<u>CHAPTER</u>	<u>PAGE</u>
1 INTRODUCTION	1
1.1 Background	1
2 PROBLEM STATEMENT	5
2.1 Problem	5
2.1.1 Input Space Partitioning	5
2.1.2 Abstractions For Performance Testing	7
2.1.3 An Illustrative Example	9
2.1.4 The Problem Statement	10
3 APPROACH	13
3.1 The ASSIST Approach	13
3.1.1 The AUT Model	13
3.1.2 An Overview of ASSIST	14
3.1.3 Blind Source Separation	16
3.1.4 Independent Component Analysis	18
3.1.5 ASSIST Architecture And Workflow	19
4 IMPLEMENTATION	23
4.1 Introduction	23
4.2 System Under Test - Web Application JPetstore	26
4.2.1 JPetstore Package details	28
4.3 Test Scripts	32
4.4 Profiling	35
4.4.1 ProbeKit	35
4.4.2 Specifying Targets in Probe	36
4.4.3 Import Directives in ProbeKit	37
4.4.4 Fragments in ProbeKit	37
4.5 Profile Repository	40
4.6 How are rules formed?	41
4.7 Profile Analyzer	43
4.8 Rule Generation	46
5 RESULTS	51
5.1 Results	51
5.2 Random Testing	52
5.3 ASSIST Tool	52
6 RELATED WORK	58
6.1 Related Work	58

TABLE OF CONTENTS (Continued)

<u>CHAPTER</u>		<u>PAGE</u>
7	CONCLUSION	60
7.1	Conclusion	60
7.2	Future Work	60
	CITED LITERATURE	62
	VITA	67

LIST OF FIGURES

<u>FIGURE</u>		<u>PAGE</u>
1	The process of input space partitioning.	7
2	The model of the AUT for ASSIST.	13
3	A speech model of blind source separation.	16
4	Schematics of the ICA matrix decomposition.	17
5	The architecture and workflow of ASSIST.	20
6	ASSIST Application Workflow	24
7	JPetstore Home Page	27
8	JDepend metrics result for JPetstore packages	28
9	Metrics for com.ibatis.jpetstore.domain	29
10	Metrics for com.ibatis.jpetstore.persistence	29
11	Metrics for com.ibatis.jpetstore.persistence.iface	30
12	Metrics for com.ibatis.jpetstore.persistence.sqlmapdao	30
13	Metrics for com.ibatis.jpetstore.presentation	31
14	Metrics for com.ibatis.jpetstore.service	31
15	Metrics for com.ibatis.struts and com.ibatis.struts.httpmap	32
16	Probekit Editor	36
17	Specifying Targets in Probekit Editor	37
18	Import Directive in Probekit Editor	38
19	Creating the ARFF file as input to the Rule Learner	42
20	Input Arff file	44
21	Snapshot of the Profile Analyzer	45
22	Menu Options to control Profile Analyzer	46
23	Rules Generated by the System	47
24	Menu Options for generating rules	48
25	Comparison of the time taken to execute 25000 transactions	55
26	Comparison of the time taken to execute 50000 transactions	55
27	Comparison of the time taken to execute 75000 transactions	56
28	Comparison of the time taken to execute 100000 transactions	56
29	Comparison of the time taken to execute 125000 transactions	57
30	ASSIST steers the load testing toward the slow performing regions of the application	57

LIST OF TABLES

<u>TABLE</u>		<u>PAGE</u>
I	Time Taken during Random Load Testing	52
II	Time Taken during ASSIST approach based Load Testing	53
III	Average time Taken by Both Approaches	54

SUMMARY

The goal of performance testing is to uncover problems where an application unexpectedly exhibits worsened characteristics for a specific workload. It is difficult to construct effective performance test cases that can find performance problems in a short period of time, since it requires test engineers to test a large number of combinations of actions and data for large-scale applications. A fundamental question of performance testing is how to find "key abstractions" that allow testers to select a manageable subset of the input data for test cases without compromising the effectiveness of testing.

We offer a novel solution for Abstraction Search for Input partitioning for Software performance Testing (ASSIST) for finding key abstractions for input space partitioning for performance testing automatically. ASSIST is an adaptive, feedback-directed learning testing system that starts with a small subset of test cases to enable testers to steer towards challenging tests automatically to find more performance problems in applications in a shorter period of testing time. We have implemented ASSIST and have applied it to a dummy web application called JPetstore which has all the functionality found in any e commerce application.

CHAPTER 1

INTRODUCTION

1.1 Background

Quality assurance is an important process in any industry. In organizations that develop software applications, various tests are performed on the system to ensure that the product meets customer's expectations. To evaluate the performance of the software product, test engineers adopt various techniques with the intent of finding defects in the software. Some defects are caused due to programming errors by developers. These defects are found by running functional tests on the system till the software fails. But not all defects can be found using functional testing. Many defects are caused due to requirements gap. The defects may be hidden in such a way that the software seems to work fine in one environment but starts failing when the environment changes. Here the failure is not related to any specific function but the overall quality of the product. For example, scalability, security or performance are non-functional properties of a software that needs to be tested. In order to uncover these defects test engineers perform various non-functional tests on the software.

Goal of performance testing is to uncover problems where an application unexpectedly exhibits worsened characteristics for a specific workload (42; 56). Performance degradation is a situation when a system has very low throughput and high response time caused by unexpectedly large transactions or concurrent users. Performance testing or load testing is performed to uncover areas of performance degradation. Load testing should ensure that the web application is able to maintain the response times for the requests during many simultaneous transactions or users. In other words, this ensure that the application is scalable. Scalability of an application

is important because of the nature of web linking of pages. As web applications may generate enormous volume of workload in a short period of time, it is important that it is scalable.

Consider an online store ecommerce application. The major business functionalities handled by the application are the following

- product catalogue which lists all the products in their store, their specifications and price
- creating user accounts for the customers and giving them access to a virtual cart to which they add their items
- allowing customers to select products and add it to their cart

Functional testing will be used to ensure that the application can successfully perform the above functionalities. If there is a requirement which says that this application should support concurrent access by hundreds of users, load testing is performed to ensure that the application performs well even when under stress.

Performance testing is important for web applications because there is a direct correlation between fast and stable web applications and the revenue generated from them. With the increase in use of web applications, the users of web applications will not tolerate high response time or errors.

Effective test cases for load testing, which is a variant of performance testing, find situations where an application suffers from unexpectedly high response time or low throughput (34; 8). Test engineers construct performance test cases, and these cases include actions (e.g., interactions with GUI objects or method calls of exposed interfaces) as well as data that accompany these actions (33). It is difficult to construct effective performance test cases that can find performance problems in a short period of time, since it requires test engineers to test all combinations of actions and data for large-scale applications.

Consider *Renters Insurance Program* (or simply *Renters*) designed and built by a major insurance company. A goal of this program is to compute quotes for insurance premiums for rental condominiums. *Renters* is written in Java and it contains over 10,000 methods that are invoked more than three million times over the course of a single end-to-end pass through the application. Its database contains approximately 78Mil customer profiles. Since it takes on average ten minutes to compute a quote for a single profile, it would take over 1,500 years to test *Renters* on test cases that cover all profile inputs. A fundamental question of testing is how to select a manageable subset of the input data for performance test cases without compromising the effectiveness of testing.

This problem is partially addressed by partitioning input data space into disjoint blocks and constructing test cases by selecting one value from each block (5, page 150)(55). A common way to partition input space is to model the domain of each program input, partition each domain's values into blocks, and then combines values from each blocks into test inputs. A classic example of input space partitioning is a calculator program where the domain of input values is partitioned into three blocks: negative numbers, zero, and positive numbers. Doing so involves introducing the “**sign**” abstraction – concrete numerical values are abstracted away and only signs are considered. A benefit of using this abstraction is that tests can be created from as few as three concrete values chosen from these partitions.

Naturally, finding proper abstractions is a highly creative process that involves deep understanding of input domains (5, page 152). Ideally, test engineers should spend more time to find abstractions that will enable them to concentrate on more challenging tests for applications rather than blindly forcing all tests, which is unfortunately a common practice now (43). Currently, a prevalent method for performance testing is *intuitive testing*, which is a method

for testers to exercise the product based on their intuition and experience, surmising probable errors (44)(17). Intuitive testing was first introduced in 1970s as an approach to use experience of test engineers to focus on error-prone and relevant system functions without writing time-consuming test specifications thus lowering pre-investment and procedural overhead costs (17). When running many different test cases and observing application’s behavior, testers intuitively sense that there are certain properties of test cases that steer applications toward more interesting behavior that is likely to reveal bugs. Distilling these properties into key abstractions automatically is a goal of our approach.

We offer a novel solution for *AbStraction Search for Input partitioning for Software performance Testing (ASSIST)* for finding key abstractions automatically. ASSIST is an adaptive, feedback-directed learning testing system that starts with a small subset of test cases to enable testers to steer towards challenging tests automatically to find more bugs in applications in a shorter period of testing time. ASSIST uses limited runtime monitoring together with machine learning techniques and automated test scripts to reduce large amounts of performance-related information collected during application runs to a small number of factors that provide insights into abstractions that guide input space partitioning. We have implemented ASSIST and applied it to JPetStore - an Online Petstore from the Open Source Community. JPetStore application has catalogues for various pets like fish, dog from which we can select and buy pets.

CHAPTER 2

PROBLEM STATEMENT

2.1 Problem

In this section we give background on the process of input space partitioning, discuss abstractions for testing, show an illustrative example of how abstractions are learned for performance testing, and formulate the problem statement.

2.1.1 Input Space Partitioning

The process of input space partitioning is shown in a diagram in Figure 1, where steps of the process are shown as rectangles, arrows specify the directions in which these steps are applied, and these arrows are labelled with numbers in circles to indicate the sequence of steps. The beginning of the process is shown with the rectangle in the upper left corner that specifies modelling of domain and application.

Modelling the domain and the application involves identifying testable functions and all parameters that can affect the behavior of each identified testable function. These parameters form the input domains of the *application under test (AUT)*. Once the input domain is defined, test engineers (1) abstract away nonessential properties of this domain leaving a set of essential input parameters that affects the performance of the system the most, in test engineer’s opinion. Using this set of essential parameters, the test engineer (2) creates domain and operational abstractions that can be described as rules on input characteristics. Recall the example of the “**sign**” abstraction for a calculator where inputs are represented as positive and negative regions of values and zero. Defining these abstractions is a highly creative and intellectually

intensive process that requires deep understanding of input domains and applications. Finding good abstraction leads to effective or “good” tests that are created using these abstractions.

A main goal of testing is to create “good” test cases with which different testing objectives can be achieved (37). These objectives include finding functional and nonfunctional defects, finding safe and non-safe scenarios for using products, and checking for conformance to regulations. Good test cases are more likely to expose bugs and to produce results that yield additional insight into behavior of application under test (i.e., they are more informative and more useful for troubleshooting). Constructing good test cases requires significant insight into an AUT and its requirements and useful abstractions for testing.

Once abstractions are found, (3) input domains are partitioned into blocks, each of which comprises a range of values for each input parameter. Then (4) tests are created by taking one value from each block and combining these values. After running these tests on AUT, (5) results are analyzed to determine if tests are effective. This analysis involves statistical evaluation of different measurements collected during execution. If this analysis shows that AUT shows the same behavior for most of all tests, then these tests are deemed ineffective. In this case, using information learned from running ineffective tests, engineers go back (6) to the initial step of modelling domains and AUT and the process repeats until proper abstractions are found and effective tests are created.

According to Beizer, five different testing levels have one thing in common – it is assumed that testers acquire certain knowledge of the application that enables them to create abstractions and models that would guide testing process. Level 0 is the most primitive – the model is the code and debugging is testing, and level 4 is when testing becomes a mental discipline that increases quality (11). That is, the current state of the art is that testers apply a certain level of

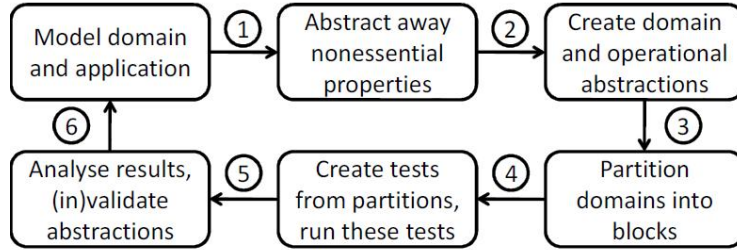


Figure 1. The process of input space partitioning.

mental exercise to create abstractions and models, and then build test cases using these models. This process is intellectually intensive, and intuitive testing is a way to address this problem by relying upon experience and intuition of testers who test applications for a long time. Clearly, automating this process is a fundamental problem of software testing.

2.1.2 Abstractions For Performance Testing

Abstraction is a fundamental technique in computer science to approximate entities or objects in order to infer useful information about programs that use these entities or objects (19). Abstract interpretation, type static checking, and predicate abstraction are examples of mathematical frameworks that allow scientists to design abstractions over mathematical structures in order to build models and prove properties of programs. Software model checking is one of largest beneficiary fields of computer science where abstractions enable engineers to deal with the problem of state space explosion.

User-defined abstractions are most effective in the solution domain, i.e., the domain in which engineers use their ingenuity to solve problems (31, pages 87,109). In the problem domain, mathematical abstractions are used to express semantics of requirements. Conversely,

in the solution domain, engineers go into implementation details. To realize requirements in the solution domain, engineers look for user-defined abstractions that are often implemented using ad-hoc techniques (e.g., mock objects that abstract missing components (24)). Thus user-defined abstractions are most effective when they reflect the reality of the solution domain (3).

Abstractions play a significant role in software testing (10). Useful abstractions for testing approximate the functionality of an AUT. For example, a useful abstraction for the Renters is that some insurance customers will pose a high insurance risk if these customers have one or more prior insurance fraud convictions and deadbolt locks are not installed on their premises. Computing insurance premium for these customers requires invoking additional procedures that retrieve and process additional data from the back-end database thereby significantly increasing the workload and degrading performance of the AUT, i.e., Renters. Using this abstraction, testers can model the system as having two main performance components: one the computes insurance premium for customers with clean history and the other for renters with fraud convictions who do not use deadbolts. With this model, testers can partition test inputs in two regions that correspond to the functionalities of these main components. Even though real-world systems exhibit much more complex behavior, useful abstractions enable testers to build effective test cases.

Abstractions for testing are notoriously difficult to capture. Test engineers must intimately know the functionality of the subject application under test, understand how programmers designed and implemented their abstractions, and hypothesize on how application behavior matches key abstractions that were extracted from requirements for this application. Without

having useful abstractions it is difficult to define objectives that lead to selecting good test cases (37).

2.1.3 An Illustrative Example

Currently, the state-of-the-art of finding useful abstractions is to use experience and intuition of performance test engineers who spend time observing the behavior of AUTs when running manually constructed test cases. There is little automated support for discovering problems with performance testing, where a recent work by Hassan et. al. is the first that can automatically detect performance problems in the load testing results by analyzing performance logs (34). Experience and intuition are main tools that performance test engineers use to surmise probable errors (44)(17).

In psychology, intuition means a faculty that enables people to acquire knowledge by linking relevant but spatially and temporally distributed facts and by recognizing and discarding irrelevant facts (54). What makes intuitive acquisition of knowledge difficult is how relevancy of facts is perceived. In software testing, facts describe properties of systems under test, and many properties may be partially relevant to an observed phenomenon. Intuition helps testers to form abstractions with correctly assigned relevancy rankings to different facts, form hypotheses based on these abstractions, and test these hypotheses without going through a formal process.

Consider an illustrative example of how intuitive testing works for Renters. A performance test engineer notices at some point that it takes more CPU and hardware resources (fact 1) to compute quotes for residents of the states California and Texas (fact 2). Independently, the database administrator casually mentions to the tester that a bigger number of transactions are executed by the database when this tester runs test cases in the afternoon (fact 3). Trying to find an answer to explain this phenomenon, the tester makes a mental note that test cases

with northeastern states are usually completed by noon and new test cases with southwestern states are executed afterwards (fact 4). A few days later the tester sees a bonfire (fact 5) and remembers that someone’s property was destroyed in wildfires in Oklahoma (fact 6). All of a sudden the tester experiences an epiphany – it takes more resources for Renters to execute tests for the states California and Texas because these states have the high probability of having wildfires.

Once this crucial logical relevance is established between facts 1 and 2, facts 3-6 are not needed any more – they simply helped to establish the rule that running tests on states with wildfires leads to bigger workload on the applications. When test cases are run for wildfire states, more data is retrieved from the database and more computations are performed. The tester then identifies other wildfire states (e.g., Oklahoma) and partition input space into wildfire and nonwildfire states. From these partitions the tester creates test cases thereby concentrating on fewer more challenging tests for Renters rather than blindly forcing all tests, which is unfortunately a common practice now (43).

2.1.4 The Problem Statement

Our goal is to automate finding useful abstractions for performance testing by reversing this process. That is, testers should be able to first run applications on a small set of test cases and then infer useful abstractions for testing with a high precision. Specifically, these abstractions should link inputs, methods, and models of back-end databases with which these methods exchange data into **if-then** rules that tell testers how to create good test cases. For example, a rule may say “if inputs `convictedFraud` is `true` and `deadboltInstalled` is `false` then the test case is good.”

In this thesis we accept a performance testing definition of what constitutes a good test case. One of the goals of performance testing is to find test cases that worsen response time or throughput of the AUT. It can be achieved by adding more users as well as finding inputs that make the AUT take more resources and time to compute results. Conversely, bad test cases are those that utilize very few resources and take much less time to execute compared to good test cases. A main goal is to produce rules that describe good and bad test cases automatically and then use these rules also automatically to partition input space from which more test cases are built and used. The system should also correct itself by testing learned rules on selecting test data from partitions that are based on these rules and verifying that these test data lead to predicted performance results.

Additional rules may provide insight into the behavior of the AUT. For example, a rule may specify that the method `checkFraud` is always invoked when test cases are good and the values of the attribute `SecurityDeposit` of the table `Finances` are frequently retrieved from the back-end database. This information helps testers to create a holistic view of testing, partition test inputs appropriately thereby reducing the number of tests, and thus these rules can be used to select better test cases automatically.

Finally, no performance testing is complete without providing sufficient clues to performance engineers where in the AUT problems can lurk. A main objective of performance analysis is to find *bottlenecks* (or *hot spots*), which are phenomena where the performance of an application is limited by one or few components (41), (52), (4), (6). A single method that drags down the performance of the entire application is easy to detect using profilers; however, it is a difficult problem to find bottlenecks when there are hundreds of methods whose execution time is approximately the same, which often is the case in large-scale applications (4),(6). The

problem that we solve in this thesis is that once the input space is partitioned into blocks that lead to good and bad test cases, we want to find methods that are specific to good performance test cases and that are most likely to contribute to bottlenecks, with high precision and in a short period of time.

CHAPTER 3

APPROACH

3.1 The ASSIST Approach

In this section we present the AUT model for ASSIST, explain the key ideas behind our approach, give an overview of ASSIST, describe the architecture of ASSIST, and provide an overview of how ASSIST is used.

3.1.1 The AUT Model

Database-centric applications (DCAs) are common in enterprise computing, and they use nontrivial databases (38). Database abstractions and models are complementary to abstractions and models that are used to create DCAs; it is often a case that they complement one another (30). Together, inputs to DCAs, method calls, and databases are fused in the logic of these DCAs. During execution of DCAs, interactions between inputs, method calls, and databases form different patterns that reflect underlying abstractions using which DCA logic is implemented. Recovering those abstractions, refining them, and making them available to testers is the main goal of this thesis.

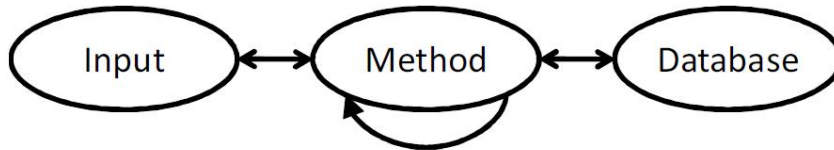


Figure 2. The model of the AUT for ASSIST.

Consider a model of the DCA that is shown in Figure 2. Inputs to the DCA lead to invocation of methods that may request additional inputs or call other methods. In addition, these methods retrieve data from databases, use this data to trigger more computations, request more input data, and insert or update data in the database. Good performance test cases are those where input data trigger more computations that update data in the databases and request more data from the databases that in turn trigger more computations. We do not differentiate between serialized and concurrent executions in this model.

3.1.2 An Overview of ASSIST

ASSIST is based on two key ideas. First, the instrumented AUT is run on a small number of test cases that can be selected randomly, its execution profiles are collected and clustered using machine learning (ML) techniques automatically into two groups that correspond to good and bad test cases. The values for AUT inputs for good and bad test cases are formed into the input to an ML classification algorithm. This input contains implications of the form $V_{I_1}, \dots, V_{I_k} \rightarrow T$, where V_{I_m} is the value of the input I_m and $T \in \{G, B\}$, G and B standing for good and bad test case correspondingly. The ML classification algorithm learns the model and outputs rules that have the form $I_p \odot V_{I_p} \bullet I_q \odot V_{I_q} \bullet \dots \bullet I_k \odot V_{I_k} \rightarrow T$, where \odot is one of the relational operators and \bullet stands for logical connectors **and** and **or**.

These learned rules are supplied back into the test script, which is a program that test engineers write to automate testing. This test script performs actions (invoking methods or mimicking user actions on GUI objects) on interfaces of the AUT using some underlying testing frameworks. Test engineers write code in test scripts that guide selection of test inputs; typically, it is done using exhaustive enumeration of all input values or by using algorithms of

combinatorial design interactions (27). In our case, selection of test inputs is guided by rules that are obtained using the ML classification algorithm.

Using newly learned rules, input test space is partitioned and the cycle repeats. The test script selects inputs from different partitions, the AUT is executed again, and new rules are learned. If no new rules are learned after some time of testing, it means that the partition of test inputs is stable with a high degree of probability. At this point instrumentation can be removed and testing can continue with using ASSIST.

Our goal is to help test engineers to identify bottlenecks automatically in a form of method calls, so that these engineers can debug the identified performance problems. Some methods are more important than others. There may be a method that is periodically executed by a thread to check to see if the content of some file is modified. While this method may be one of the bottlenecks, it is invoked in both good and bad test cases thus reflecting the fact that its presence does not lead to any insight that may resolve a problem that leads to good test cases.

Our second key idea is to consider the most significant methods that occur in good test cases and that are not invoked or have little to no significance in bad test cases. The significance of a method is a function of the number of times that this method is invoked, the total elapsed time of its invocations minus the elapsed time of all methods that are invoked from this method, the number of attributes that this method accesses in the databases, the amount of data it transfers between the AUT and the databases, and finally, the number of methods that are invoked from this method. Large applications implement multiple requirements, each of these requirements is implemented using different methods. Each AUT run involves thousands of its methods that are invoked millions of times. The resulting execution profile is a mixture of different method invocations, each of which address a part of some requirement. To identify most

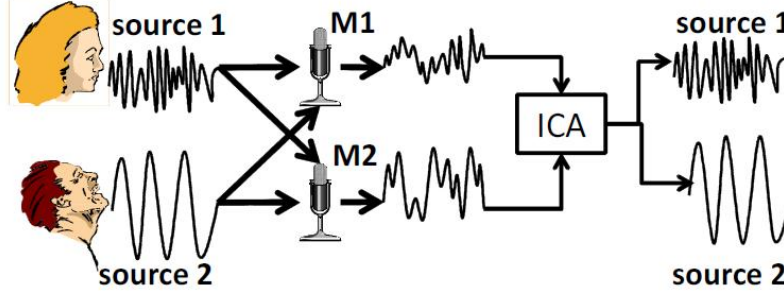


Figure 3. A speech model of blind source separation.

significant methods, we need a new approach with which we break these profiles automatically into components that match high-level requirements and then we identify methods with most significant contributions to these components.

3.1.3 Blind Source Separation

We draw an analogy between separating method invocations in execution profiles into components that represent high-level requirements and a well-known problem of separating signals that represent different sources from a signal that is a mixture of these separate signals. This problem is known as *blind source separation (BSS)*(47, pages 13-18).

The idea of BSS is illustrated in Figure 3. Two people speak at the same time in a room with two microphones M1 and M2. Their speech signals are designated as **source 1** and **source 2**. Each microphone captures the mixture of the signals **source 1** and **source 2** shown as the corresponding signal mixtures from M1 and M2 respectively. The original signals **source 1** and **source 2** are separated from the mixtures using a technique called *independent component analysis (ICA)*(32), which we describe in Section 3.1.4. Even though the idea of BSS is illustrated using the speech model, ICA is widely used in econometrics to find hidden fac-

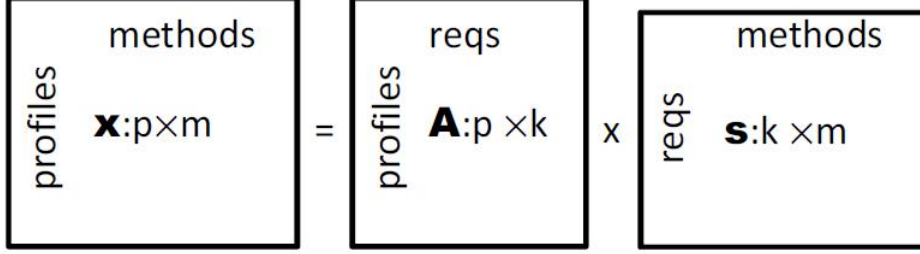


Figure 4. Schematics of the ICA matrix decomposition.

tors in financial data, image denoising and feature extraction, face recognition, compression, watermarking, topic extraction, and automated concept location in source code (26).

In this thesis we adjust the BSS model for breaking profiles automatically into components that match high-level requirements and then identifying methods with most significant contributions to these components. Nontrivial applications implement quite a few high-level requirements in different methods that are executed in different threads, often concurrently. We view each requirement as a source of a signal that consists of method calls. When an application is executed, multiple requirements are realized, and method invocations are mixed together in a mixed signal that is represented by the execution profile. Microphones are represented by instrumenters that capture program execution; multiple executions of the application with different input data is equivalent to different speakers talking at the same time – as a result multiple signal mixtures (i.e., execution profiles for different input data with mixed realized requirements) are produced. With ICA, not only it is possible to separate these signal mixtures into components, but also to define most significant constituents of these signals (i.e., method calls). We choose ICA because it works with non-Gaussian distributions of data, which is the case with ASSIST.

3.1.4 Independent Component Analysis

ICA is a recently developed mathematical technique for separating signal mixtures into statistically independent signals (32). It is based on the assumption that different signals from different physical processes are statistically independent. For example, different requirements are often considered independent since they implemented in applications as separate concerns (46; 53). When physical processes are realized (e.g., different people speak at the same time or stocks are traded or an application is run and its implementations of different requirements are executed in methods concurrently), these different signals are mixed and these signal mixtures are recorded by some sensors. Using ICA, independent signals can be extracted from these mixtures with a high degree of precision.

A schematics of ICA matrix decomposition is shown in Figure 4. The equation $\| \mathbf{x} \| = \| \mathbf{A} \| \cdot \| \mathbf{s} \|$ described the process, where $\| \mathbf{x} \|$ is the matrix that contains the observed signal mixtures and $\| \mathbf{A} \|$ is the transformation or mixing matrix that is applied to the signal matrix $\| \mathbf{s} \|$. In our case, the matrix $\| \mathbf{x} \|$ is shown in Figure 4 on the left hand side of the equal sign, and its rows correspond to application runs with different input data (or profiles) with its columns corresponding to method invocations that are observed for each profile. Each element of the matrix $\| \mathbf{x} \|$ is calculated as

$$x_i^j = \lambda_N \cdot N_i^j + \lambda_T \cdot T_i^j + \lambda_A \cdot A_i^j + \lambda_D \cdot D_i^j + \lambda_M \cdot M_i^j$$

where N_i^j is the number of times that the method j is invoked in the profile i , T_i^j is the total elapsed time of these invocations minus the elapsed time of all methods that are invoked from this method in this profile, A_i^j is the number of attributes that this method accesses in the databases, D_i^j is the amount of data that this method transfers between the AUT and the

databases, M_i^j is the number of methods that are invoked from this method, and finally, λ are normalization coefficients computed for the entire matrix $\| \mathbf{x} \|$ to ensure $0 \leq x_i^j \leq 1$. Naturally, $x_i^j = 0$ means that the method i is not invoked in the profile j , while $x_i^j = 1$ means that the given method makes the most significant contribution to the computation in the given profile.

Using ICA the matrix $\| \mathbf{x} \|$ is decomposed into a transformation and a signal matrices that are shown on the right hand side of the equal sign in Figure 4. The input to ICA is the matrix $\| \mathbf{x} \|$ and the number of source signals, that in our case is the number of requirements (reqs in the Figure 4) implemented in the application. Elements of the matrix $\| \mathbf{A} \|$, A_p^q specify weights that each profile p contributes to executing code that implements the requirement q , and elements of the matrix $\| \mathbf{s} \|$, s_q^k specify weights that each method k contributes to executing code that implements the requirement q . Methods that have the highest weights for given requirements are thought to be the most significant and interesting for troubleshooting performance problems. This is a hypothesis that we validate with our case study.

3.1.5 ASSIST Architecture And Workflow

The architecture of ASSIST is shown in Figure 5. Solid arrows show command and data flows between components, and numbers in circles indicate the sequence of operations in the workflow. The beginning of the workflow is shown with the fat arrow that indicates that the Test Script executes the application by simulating virtual users and invoking methods of the AUT interfaces. The Test Script is written (1) by the test engineer as part of automating application testing; it is practically impossible to performance test applications without automated test scripts since it is not feasible to engage hundreds of thousands of testers who will call multiple methods with high frequency manually (20)(21)(36)(42).

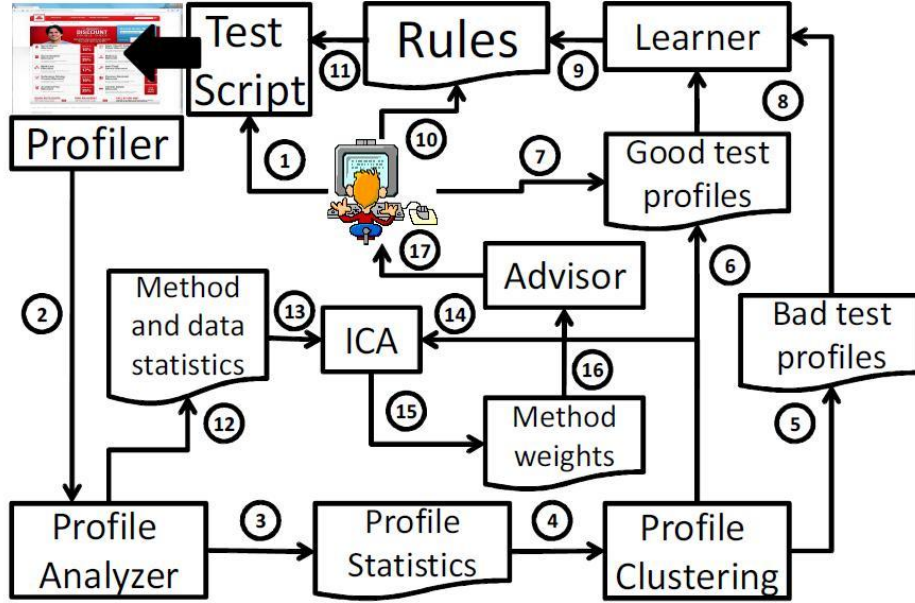


Figure 5. The architecture and workflow of ASSIST.

Once the test script starts executing the application, its execution profiles are collected (2) by the Profiler, and these profiles are forwarded to the Profile Analyzer, which produces (3) the Profile Statistics. This statistics contains information on each profile such as the number of invoked methods, the elapsed time it takes to complete the end-to-end application run, the number of threads, and the number of unique methods that were invoked in this profile. The profile statistics is supplied (4) to the module Profile Clustering, which uses an ML algorithm to perform unsupervised clustering of these profiles into two groups that correspond to (6) Good and (5) Bad test profiles. The user can review the results of clustering and (7) reassign clustered profiles if a need exists. These clustered profiles are supplied (8) to the Learner that uses them to learn the classification model and (9) output rules that we described in Section 3.1.2. The user can review (10) these rules and mark some of them as erroneous if the

user has sufficient evidence to do so. Then the rules are supplied (11) to the Test Script that keeps executing the application, but at the same time it listens at a connection for rules. Once the Test Script receives a new set of rules, it partitions the input space into blocks according to these rules and starts forming test inputs by selecting one input from each block. Thus the cycle repeats, with new rules that can be learned at each several passes and the input space is repartitioned adaptively to accomodate these rules.

Interestingly, the AUT does not have to execute instrumented with the Profiler for the entire duration of testing. Once ASSIST does not learn new rules, which means that the classification model performs well on newly added profiles, instrumentation can be removed and the AUT can be run on the partitioned input space using learned rules. This way the performance degradation that is introduced by instrumentation is temporary, which makes this approach usable in practice.

Finally, recall from Section 2.1.4 that once the input space is partitioned into blocks that lead to good and bad test cases, we want to find methods that are specific to good performance test cases and that are most likely to contribute to bottlenecks. This task is accomplished in parallel to computing rules, and it starts when the Profile Analyzer produces (12) the method and data statistics that is used to construct (13) two matrices $\| \mathbf{x}_B \|$ and $\| \mathbf{x}_G \|$ for (14) bad and good test cases correspondingly. Constructing these matrices is done as we described in Section 3.1.4. Once these matrices are constructed, ICA decomposes them (15) into the matrices $\| \mathbf{s}_B \|$ and $\| \mathbf{s}_G \|$ for bad and good test cases correspondingly. Recall that our key idea is to consider the most significant methods that occur in good test cases and that are not invoked or have little to no significance in bad test cases. Crossreferencing the matrices $\| \mathbf{s}_B \|$ and $\| \mathbf{s}_G \|$ which specifies method weights for different requirements, the Advisor (16) determines top methods

that performance testers should look at (17) to debug possible performance problems. This step completes the workflow of ASSIST.

CHAPTER 4

IMPLEMENTATION

4.1 Introduction

In this section we explain the implementation details of ASSIST. A detailed application work flow is illustrated in the Figure 6.

The JPetstore application is statically instrumented with user defined probes. We use JMeter to create automated test scripts for load testing. Automated test scripts are used because load testing requires to simulate many concurrent users and transactions which is not feasible to perform using manual testing. Our application has JMXLoadScript module which creates JMeter test scripts iteratively. Initially, the test scripts are generated by randomly selecting URLs of the JPetstore application. Once the test scripts start executing JPetstore on the Apache Tomcat web server, the execution profiles of the test runs are collected by the Profiler. The execution profile contains run time information about the JPetstore application written to the Profile Repository by the probes. These execution profiles are processed by the ProfileAnalyzer to produce Profile Statistics. The Profile Clustering module analyzes the profile statistics to cluster the profiles as Good or Bad Test profiles. The user can review the results of clustering and may reassign the class of Profiles. These clustered profiles are fed as input to the Learner which uses a classification model to learn rules. In our implementation we use WEKA to implement the machine learning algorithms. Once the rules are generated, JMXLoadScript module includes these rules in the new test scripts.

The application has following parts.

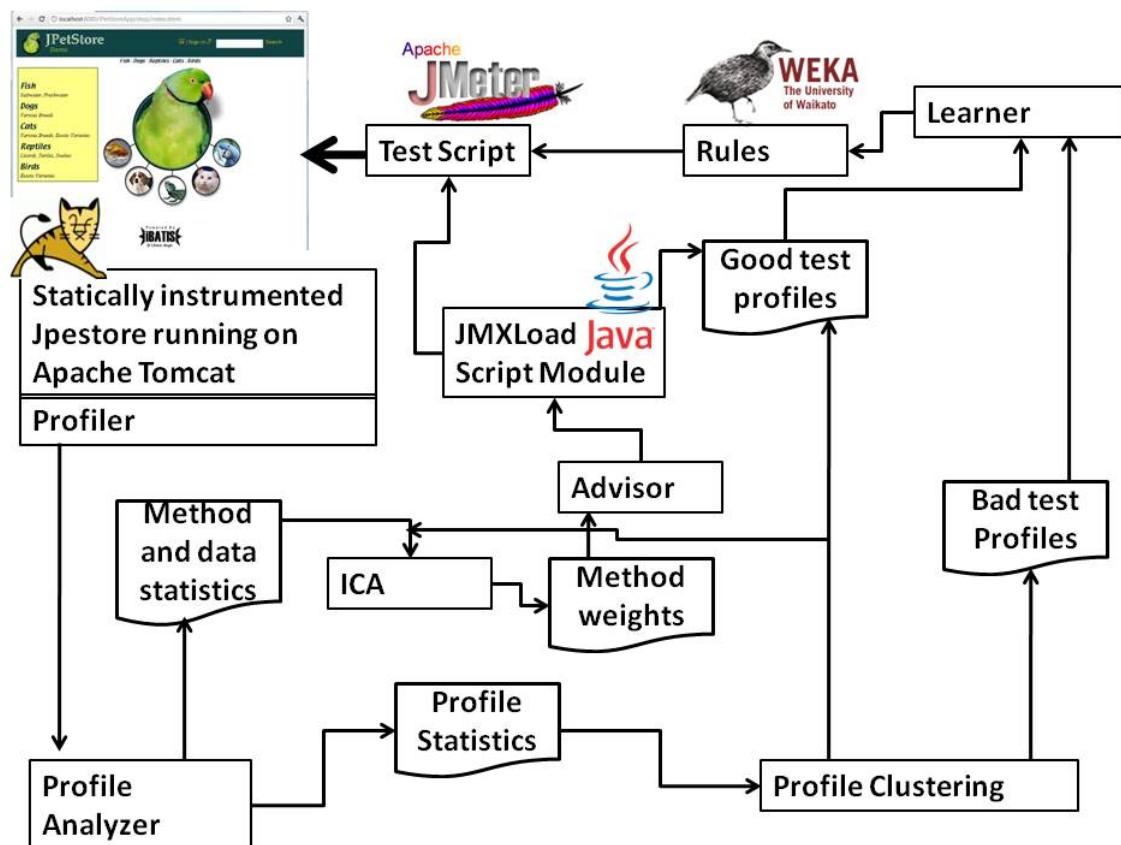


Figure 6. ASSIST Application Workflow

- User defined probes generated using ProbeKit. ProbeKit is a framework on Eclipse Test and Performance Tools Platform (TPTP)
- The JMXLoadScript module. It generates the JMeter Test Scripts. These scripts are composed of random HTTPRequests to the web server where JPetstore is deployed. As a preliminary setting, we start the Apache tomcat server at the beginning of the application run.
- The JPetstore web application which has been statically instrumented by ProbeKit and runs on Apache Tomcat web server. When the HTTPRequest hits the web server, the instrumented code writes data about method invocations, SQL queries and ServletRequests to a socket.
- ProfileRepository which opens a ServerSocket which listens on a port. Once a client socket writes to this port, it reads from the socket and generates profiles for each run of the test.
- ProfileAnalyzer which lets the user assign classes to each profile and analyse the profiles. It uses a machine learning algorithm to learn rules from these clustered profiles. Profiles can be clustered as Good or Bad Profile depending on the time it takes for all the method invocations. In our implementation, for a given set of clustered profiles, those profiles which have total elapsed time greater than the average elapsed time for the entire set is considered as good profile.

WEKA is used to perform this clustering.(7) WEKA is a collection of open source Machine Learning algorithms for data mining. Once the profiles are classified, rules are generated. The user can reassign the class of the clustered profiles if need be. These rules are fed back to the test scripts generation module.

4.2 System Under Test - Web Application JPetstore

For the implementation of ASSIST, we used JPetstore(2). It is an online petstore application (see Figure 7) which allows users to buy pets. User can select among many options to buy dogs, cats or fishes. Each user gets a virtual cart to which items can be added and the user can checkout the cart after shopping. Users have to enter their billing address, shipping address and credit card information in order to checkout from the store.

The performance critical scenarios in this application are

- Browsing the catalogue of various pets
- Adding,Removing,Updating items to the shopping cart
- Customer Log in,Log out
- Checkout

The JPetstore is based on a Model-View-Controller framework. It uses Apache Struts framework to implement the controller. Model (application business logic), view (html pages) and controller (ActionServlet) are bound to each other through the configuration file struts-config.xml. We have deployed iBatis JPetstore on Apache Tomcat web server and we use Apache Derby as the database backend.(1) Each test script simulates a load of 5 users executing 100 HttpRequests for 5 iterations on the web server. The next section talks in detail about the test scripts.

In order to have a clearer understanding of the System Under Test, we use JDepend to calculate the metrics of the JPetstore application. JDepend traverses through Java class files and generates design quality metrics for each Java Package.(14) Jdepend can calculate the following metrics of the packages:

1 Number of Concrete classes [CC]

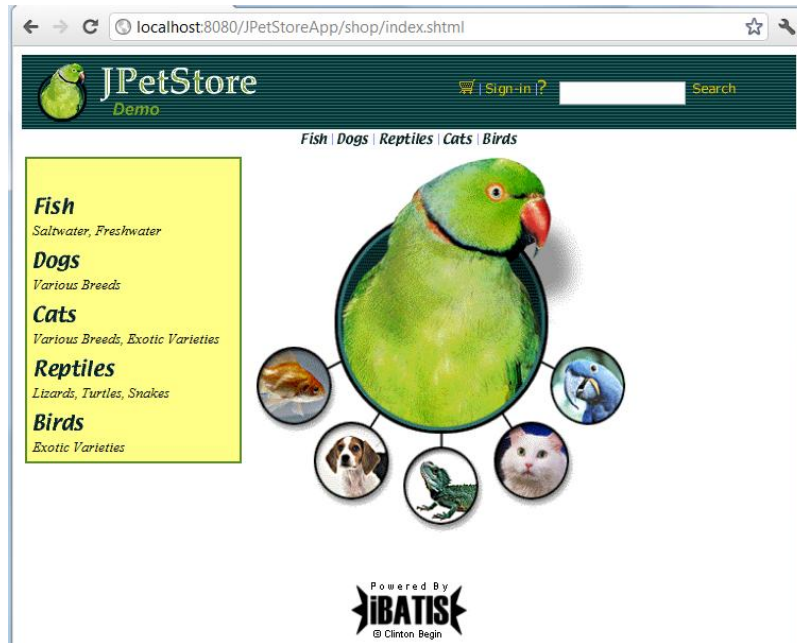


Figure 7. JPetstore Home Page

2 Number of Abstract classes [AC]

3 Afferent Coupling [Ca] - Number of packages which depend on this package, how responsible is this package.

4 Efferent Coupling [Ce] - Number of packages which this package depends on, how independent is this package.

5 Abstractness [A] - Ratio of number of abstract classes in the package to the total number of classes in the package.

$$A = \frac{AC}{AC + CC} \quad (4.1)$$

It has a value between 0 and 1. A value of 1 indicates completely abstract and 0 is completely concrete.

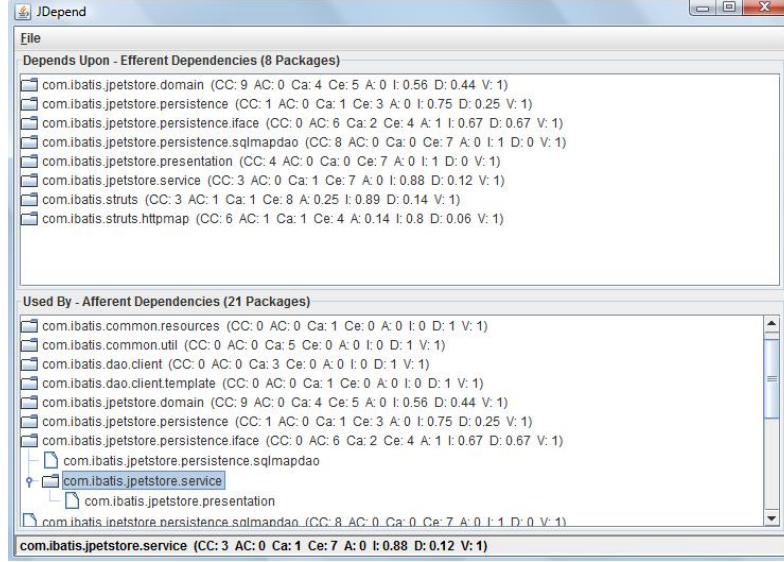


Figure 8. JDepend metrics result for JPetstore packages

6 Instability [I] - Ratio of Efferent coupling to total coupling.

$$I = \frac{Ce}{Ce + Ca} \quad (4.2)$$

7 Distance from the Main Sequence [D] - This represents the perpendicular distance from the imaginary line $A + I = 1$. The x coordinate will represent the abstractness and y coordinate will represent the instability.

Figure 8 is a snapshot of the results generated by JDepend (48). It lists all the packages and the values for each of the above mentioned metrics.

4.2.1 JPetstore Package details

com.ibatis.jpjpetstore.domain This package has a total of 9 classes. All of them are Concrete classes. As there are no abstract classes, the abstractness (A) is 0%. This package has

com.ibatis.jpjpetstore.domain				
Afferent Couplings	Efferent Couplings	Abstractness	Instability	Distance
4	5	0%	56%	44%
Abstract Classes	Concrete Classes	Used by Packages	Uses Packages	
None	Account Cart CartItem Category Item LineItem Order Product Sequence	com.ibatis.jpjpetstore.persistence.iface com.ibatis.jpjpetstore.persistence.sqlmapdao com.ibatis.jpjpetstore.presentation com.ibatis.jpjpetstore.service	com.ibatis.common.util java.io java.lang java.math java.util	

Figure 9. Metrics for com.ibatis.jpjpetstore.domain

com.ibatis.jpjpetstore.persistence				
Afferent Couplings	Efferent Couplings	Abstractness	Instability	Distance
1	5	0%	83%	17%
Abstract Classes	Concrete Classes	Used by Packages	Uses Packages	
None	DaoConfig LocalHsqldbConfigurator	com.ibatis.jpjpetstore.service	com.ibatis.common.resources com.ibatis.dao.client java.io java.lang javax.servlet	

Figure 10. Metrics for com.ibatis.jpjpetstore.persistence

a efferent coupling of 5 because it uses 5 external packages. Also, this package has a afferent coupling of 4 since it is being used by 4 other packages. Figure 9 illustrates these metrics clearly.

com.ibatis.jpjpetstore.persistence This package has a total of 2 classes. All of them are Concrete classes. As there are no abstract classes, the abstractness (A) is 0%. This package has a efferent coupling of 5 because it uses 5 external packages. Also, this package has a afferent coupling of 1 since it is being used by 1 other package. Figure 10 illustrates these metrics clearly.

com.ibatis.jpjpetstore.persistence.iface This package has a total of 6 classes. All of them are Abstract classes. As there are no concrete classes, the abstractness (A) is 100%. This package has a efferent coupling of 4 because it uses 4 external packages. Also, this package has a afferent coupling of 2 since it is being used by 2 other packages. Figure 11 illustrates these metrics clearly.

com.ibatis.jpjpetstore.persistence.iface				
Afferent Couplings	Efferent Couplings		Abstractness	Instability
2	4		100%	67%
Abstract Classes	Concrete Classes	Used by Packages	Uses Packages	
AccountDao CategoryDao ItemDao OrderDao ProductDao SequenceDao	None	com.ibatis.jpjpetstore.persistence.sqlmapdao com.ibatis.jpjpetstore.service	com.ibatis.common.util com.ibatis.jpjpetstore.domain java.lang java.util	

Figure 11. Metrics for com.ibatis.jpjpetstore.persistence.iface

com.ibatis.jpjpetstore.persistence.sqlmapdao				
Afferent Couplings	Efferent Couplings		Abstractness	Instability
0	7		0%	100%
Abstract Classes	Concrete Classes	Used by Packages	Uses Packages	
None	AccountSqlMapDao BaseSqlMapDao CategorySqlMapDao ItemSqlMapDao OrdersSqlMapDao ProductSqlMapDao ProductSqlMapDao\$ProductSearch SequenceSqlMapDao	None	com.ibatis.common.util com.ibatis.dao.client com.ibatis.dao.client.template com.ibatis.jpjpetstore.domain com.ibatis.jpjpetstore.persistence.iface java.lang java.util	

Figure 12. Metrics for com.ibatis.jpjpetstore.persistence.sqlmapdao

com.ibatis.jpjpetstore.persistence.sqlmapdao This package has a total of 8 classes. All of them are Concrete classes. As there are no abstract classes, the abstractness (A) is 0%. This package has a efferent coupling of 7 because it uses 7 external packages. Also, this package has a afferent coupling of 0 since it is not being used by any other packages. Figure 12 illustrates these metrics clearly.

com.ibatis.jpjpetstore.presentation This package has a total of 4 classes. All of them are Concrete classes. As there are no abstract classes, the abstractness (A) is 0%. This package has a efferent coupling of 7 because it uses 7 external packages. Also, this package has a afferent coupling of 0 since it is not being used by any other packages. Figure 13 illustrates these metrics clearly.

com.ibatis.jpjpetstore.presentation				
Afferent Couplings	Efferent Couplings	Abstractness	Instability	Distance
0	7	0%	100%	0%
Abstract Classes	Concrete Classes	Used by Packages	Uses Packages	
None	AccountBean CartBean CatalogBean OrderBean	None	com.ibatis.common.util com.ibatis.jpjpetstore.domain com.ibatis.jpjpetstore.service com.ibatis.struts java.lang java.util javax.servlet.http	

Figure 13. Metrics for com.ibatis.jpjpetstore.presentation

com.ibatis.jpjpetstore.service				
Afferent Couplings	Efferent Couplings	Abstractness	Instability	Distance
1	7	0%	88%	12%
Abstract Classes	Concrete Classes	Used by Packages	Uses Packages	
None	AccountService CatalogService OrderService	com.ibatis.jpjpetstore.presentation	com.ibatis.common.util com.ibatis.dao.client com.ibatis.jpjpetstore.domain com.ibatis.jpjpetstore.persistence com.ibatis.jpjpetstore.persistence.iface java.lang java.util	

Figure 14. Metrics for com.ibatis.jpjpetstore.service

com.ibatis.jpjpetstore.service This package has a total of 3 classes. All of them are Concrete classes. As there are no abstract classes, the abstractness (A) is 0%. This package has a efferent coupling of 7 because it uses 7 external packages. Also, this package has a afferent coupling of 1 since it is not being used by any other packages. Figure 15 illustrates these metrics clearly.

com.ibatis.struts This package has a total of 4 classes. 3 of them are Concrete classes and 1 Abstract class. The abstractness (A) is 25%. This package has a efferent coupling of 8 because it uses 8 external packages. Also, this package has a afferent coupling of 1 since it is not being used by any other packages. Figure 15 illustrates these metrics clearly.

com.ibatis.struts.httpmap This package has a total of 7 classes. 6 of them are Concrete classes and 1 Abstract class. The abstractness (A) is 14%. This package has a efferent coupling

com.ibatis.struts				
Afferent Couplings	Efferent Couplings		Abstractness	Instability
1	8		25%	89%
Abstract Classes	Concrete Classes	Used by Packages	Uses Packages	
BaseBean	ActionContext BeanAction BeanActionException	com.ibatis.jpstore.presentation	com.ibatis.common.exception com.ibatis.struts.httpmap java.lang java.lang.reflect java.util javax.servlet javax.servlet.http org.apache.struts.action	

com.ibatis.struts.httpmap				
Afferent Couplings	Efferent Couplings		Abstractness	Instability
1	4		14%	80%
Abstract Classes	Concrete Classes	Used by Packages	Uses Packages	
BaseHttpMap	ApplicationMap CookieMap CookieMap\$CookieEnumerator ParameterMap RequestMap SessionMap	com.ibatis.struts	java.lang java.util javax.servlet javax.servlet.http	

Figure 15. Metrics for com.ibatis.struts and com.ibatis.struts.httpmap

of 4 because it uses 4 external packages. Also, this package has a afferent coupling of 1 since it is not being used by any other packages. Figure 15 illustrates these metrics clearly.

4.3 Test Scripts

Load testing is performed to identify the bottlenecks in the web application as well as the maximum operating capacity of the application. The process followed to perform load testing involves various stages. The web application is thoroughly studied and the various performance-critical scenarios are identified. Once the performance objectives and metrics are clear, tests are designed to perform load testing. To execute these tests, tools are used to simulate the load on the application. In this implementation of ASSIST Apache Jmeter is used.

Apache JMeter (49) is a Java desktop application designed to load test functional behaviour and measure performance. This implementation mainly focuses on testing the Apache Tomcat Web Server which hosts the JPetstore application. JMeter performs tests on web server by sending the server HTTP/HTTPS requests. It runs in a fully multi threaded framework which allows to specify the number of concurrent users needed to simulate. This is done by specifying

the number of Threads in the Thread Group. JMeter provides both GUI and Non-GUI options to work with. We chose the Non-GUI option as the test scripts had to be generated iteratively and randomly from the JMXLoadscript module.

Once the number of users have been decided, the tasks assigned to each user need to be specified. We have to specify which HTTP Requests each user will fire. All the possible URLs of JPetStore were identified and stored in an array list. JMXLoadScript module was used to randomly select 20 URLs from the array list. The HTTP Requests could be GET or POST. After thoroughly studying the JPetstore web application, we found that there were a total of 120 possible URLs. 5 of these were HTTP POST requests and 115 of these were HTTP GET requests. Out of the 115 GET requests, 12 URLs did not pass any URL parameters and the rest did.

To construct the test scripts, we use the classes provided by JMeter. We start with the `org.apache.jmeter.testelement.TestPlan` which has a `org.apache.jmeter.threads.ThreadGroup` object and a `HashTree`. The `ThreadGroup` object implements the thread group of the test. Thread group controls the number of threads or the number of distinct users running the test. It allows us to specify the following parameters of the test

- The number of distinct users running the test
- The ramp up period of the test
- The number of times to execute the test

```
ThreadGroup threadGroup = new ThreadGroup();
LoopController lc = new LoopController();
lc.setLoops(5);
lc.setContinueForever(true);
threadGroup.setSamplerController(lc);
```

```
threadGroup.setNumThreads(5);
threadGroup.setRampUp(1);
```

The HashTree contains all the Samplers we need to have in the test. Samplers tell JMeter to send requests to a server and wait for a response. They are processed in the order they appear in the tree. We use `org.apache.jmeter.protocol.http.sampler.HTTPSampler` to implement the HTTP requests. To create the HTTP requests, we set the domain, port, path and the arguments of the request. Depending on whether we choose to create a HTTP GET or POST request, we set the method of the HTTPSampler object and the arguments. Arguments are implemented using `org.apache.jmeter.protocol.http.util.HTTPArgument` and specified as key-value pairs.

```
sampler.setDomain("localhost");
sampler.setPort(8080);
path = URLS[random.nextInt(19)]; //randomly selecting URL from the list of possible URLs
sampler.setPath(path);
if(URLGetMap.containsKey(path)) {
    sampler.setMethod("GET");
    sampler.setArguments(createArguments(path,true)); //createArguments() creates arguments based on the URL chosen
    ...
}
```

All the Samplers are added to the HashTree and it is passed from the JMXLoadScriptModule to the main thread which runs JMeter. JMeter thread parses this HashTree and executes the HTTPRequests sequentially. Another functionality of JMXLoadScript is to include rules in the test scripts, once they are generated. Once rules are generated by the Learner, the main JMeter thread gets a notification that new rules exist. It passes this information to JMXLoadScript module. In the beginning of every iteration, we check whether new rules exist. If they do, we parse the rules and build our Test Script to incorporate the rule.

4.4 Profiling

To classify a given run of test script as good or bad, we need to find exactly how much time it takes to execute it. Each test script consists of 20 random URLs initially. As we find more rules, the test script incorporates those rules. We classify test scripts as good or bad based on the total time taken for all the method invocations. Since we are performing load testing, we are interested to find those cases where the total time taken for the method invocation is high. These will let us focus on areas causing performance issues. In order to find the total time for the method invocations, we needed to collect the runtime information required for collecting the various statistics of each test script run. There were basically 3 things we focused on

- the method calls made, the parameters passed to the method and the objects returned by the method call
- the SQL queries made and the parameters passed to the queries
- the HTTP requests passed by the to the web server.

4.4.1 ProbeKit

For profiling, we used ProbeKit (16), which is available through Eclipse Test and Performance Tools Platform (TPTP). Probekit allows us to insert fragments of code that can be invoked from specific points in a Java Class (at entry, at exit). These fragments of code are called Probes. Eclipse TPTP provides the Probekit Editor which is used to define probes. Probes are contained in a probekit source file which has extension '.probe'. A single Probekit Source file may contain more than one probes. Probe specification consists of

- Target specifications which indicates the classes and methods on which the probe should be applied.

- Import Directives specify the Java packages and classes that are referenced by the probe.
- Framgments is the logic of the probe. Here we specify the fragment type, data items and the Java code.

Targets let us specify filter rules which lets us include or exclude certain methods or classes from instrumentation. The Probekit Editor performs pattern matching to decide which classes and methods must be instrumented. The pattern can also include wildcard character. The above figure applies the probe only to all the methods in the com.ibatis.jpestore package. All other classes and methods are excluded from instrumentation. If no value is entered, all the

Targets let us specify filter rules which lets us include or exclude certain methods or classes from instrumentation. The Probekit Editor performs pattern matching to decide which classes and methods must be instrumented. The pattern can also include wildcard character. The above figure applies the probe only to all the methods in the com.ibatis.jpestore package. All other classes and methods are excluded from instrumentation. If no value is entered, all the

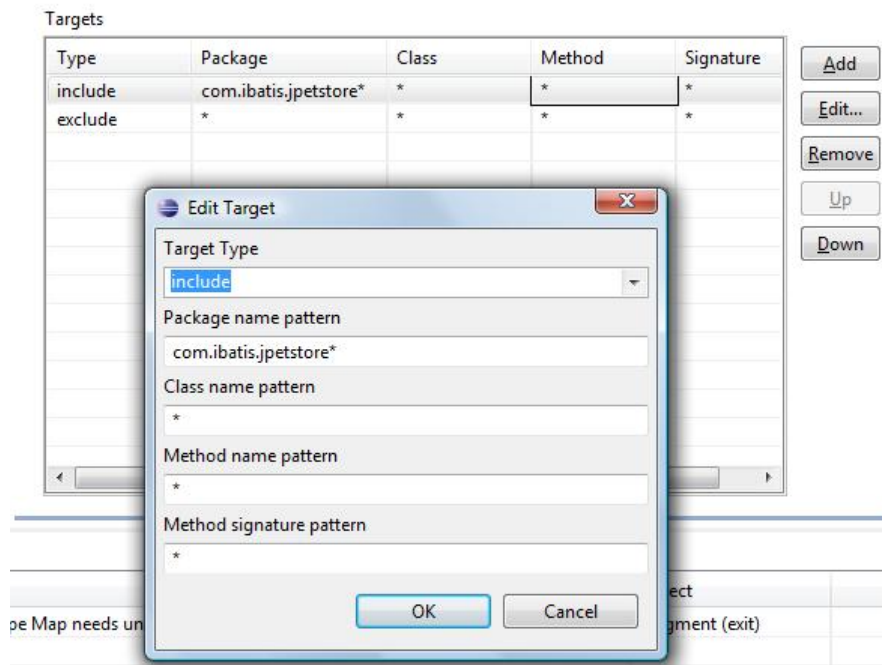


Figure 17. Specifying Targets in Probekit Editor

classes and methods will be instrumented. To apply probes to only specific classes or methods, a target specification `type=exclude,package=*,className=*,method=*,signature=*` is specified.

4.4.3 Import Directives in ProbeKit

An import directive is to import those classes which are being referenced by the probe. They are optional. A probe can contain more than one import directive.

4.4.4 Fragments in ProbeKit

Probe fragments defines the logic. There are different types of probe fragment. The type indicates when the fragment will run. A probe can contain more than one fragment, but cannot contain more than one fragment of a given type. For example, probe with fragment type as "entry" will run upon the entry of every method. Similarly, a probe with fragment type as

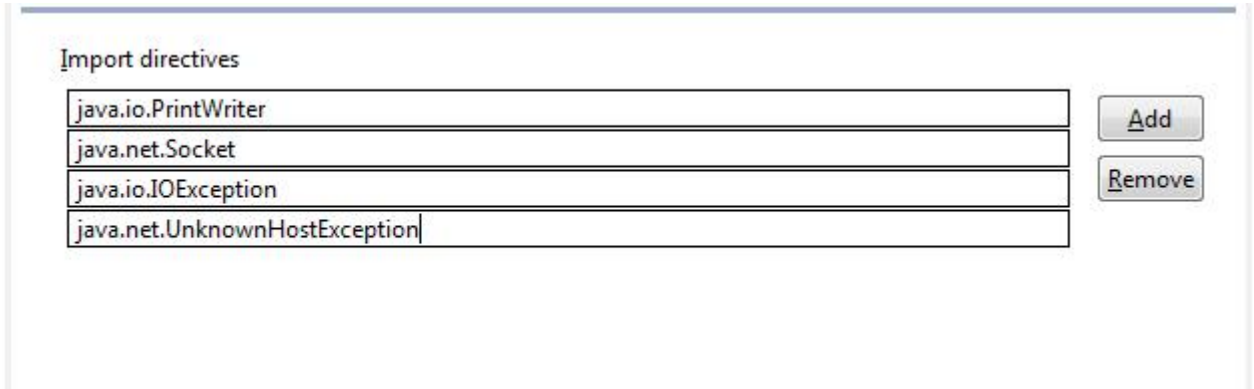


Figure 18. Import Directive in Probekit Editor

"exit" will run upon method exit. The different fragment types are entry, exit, afterCall which runs after the target method is called, beforeCall which runs before the target method is called, catch which runs at the beginning of a catch clause, staticInitializer which runs during the static class initializer of every probed class, executableUnit which runs before every executable unit of code in methods that match the probe's target and filter specification.

Fragments also contain data items. It specifies the name and type of data which will be used in the fragment code. The specification of data item is optional. While specifying a data item, we have to specify its data type and variable name. There is a small set of datatypes identified by ProbeKit. `className`, `methodName`, `methodSig`, `args` are some of the available datatypes. The variable name follows any Java variable name naming convention.

The final part of a fragment contains the Java source code which has to be executed while the probe runs. This source code accesses various run time information about the instrumented class files using the data items. For example, the name of the currently loaded class can be retrieved through `className` data item.

In our implementation, we created three probes. The first probe was for entry and exit of every method in all classes in the package `com.ibatis.jpetstore`. The fragment source code captures information about the timestamp at which the probe is called while method entry and exit. It also capture the method name, method signature and the classname. All this information is then written to a socket. The information in the socket will be later read by the profiler to create the profiles.

Another probe captures all the information associated with the database related methods. This probe is run after any database related method is invoked. It captures the method name and writes it to the same socket.

The third probe is written to capture the Http get and post requests. In this code fragment we capture the http requests URL and the parameters passed which is written to the socket. These values are later used as input ARFF data in WEKA to generate rules.

After writing the probes, we have to instrument the classes. There are two methods of instrumenting the class files using ProbeKit - dynamic and static. For our implementation we instrumented the class files statically. To enable profiling over tomcat, we had to enable standalone profiling. To do this, we need to set some Java options in tomcat. We add these options to `catalina.bat`.

```
-agentlib:JPIBootLoader=JPIAgent:server=standalone;
ProbekitAgent:ext-pk-BCILibraryName=BCIEngProbe,
ext-probescript=<dir where SQLProbe is downloaded>\bin\SQLProbe.probescript
```

To run the Java Profiler in stand alone mode, in Windows, we need to add certain locations to our Path before we can start profiling.

- TPTP Agent Controller Home - Location where the agent controller is installed

```
..\agtctrl.win_ia32-TPTP-4.7.1a\bin
```


- TPTP Agent Controller Plugins - Location where the Agent Controller plugins are installed

```
..\agntctrl.win_ia32-TPTP-4.7.1a\plugins\org.eclipse.hyades.probekit\lib
```

- TPTP JAVA Profiler Home - Location of tptp java profiler

```
..\agntctrl.win_ia32-TPTP-4.7.1a\plugins\org.eclipse.tptp.javaprofiler
```

4.5 Profile Repository

When the probes run, the information which is collected about the application run is written to a socket, dynamically. Profile Repository is the part of application which ensures that the Server Socket is ready for listening. Profile Repository reads the information written in the socket and creates profiles. Profiles are text files which have all the information retrieved by the probes for each run of the JMeter test. All these profiles are stored in the Repository.

Given below are few examples of part of a profile. These are captured from the three different probes which was mentioned previously. This line from the profile captures the information on the struts class to which the HTTP request is redirected, the URL and all the request parameters. It was captured by the probe which runs during the exit from every doGet and doPost methods.

```
REQENT_|_http-8080-1_|_30547667616764_|_org/apache/struts/action/
ActionServlet_|_doGet(Ljavax/servlet/http/HttpServletRequest;Ljavax/servlet/http/HttpServletResponse;)
V_|_URL_|_http://localhost:8080/JPetStoreApp/shop/viewProduct.shtml_|_productId={RP-SN-01+}|_|_
```

This line from the profile captures information on the database calls made. It lists the java.sql Class and method which was invoked. It also lists the SQL query which was used. It was captured by the probe which runs after the call of every database related method.

```
JDBC_|_http-8080-1_|_30546988983935_|_25987818_|_32778033_|_prepareStatement_|_
```

```
select ITEMID, LISTPRICE, UNITCOST, SUPPLIER, I.PRODUCTID, NAME, DESCN, CATEGORY,
STATUS, ATTR1, ATTR2, ATTR3, ATTR4, ATTR5
from ITEM I, PRODUCT P where P.PRODUCTID = I.PRODUCTID and I.PRODUCTID = ?_||_
```

This line from the profile captures information on the method calls which were made. It contains the name of the class and the method which was invoked. This was captured by the probe which runs on entry and exit of every method call made in the com.ibatis package.

```
MTDENT_|_http-8080-1_|_30542358122959_|_com.ibatis/common/resources/
Resources_|_getResourceAsStream(Ljava/lang/ClassLoader;Ljava/lang/String;)Ljava/io/InputStream;_||_
MTDRET_|_http-8080-1_|_30542403474044_|_com.ibatis/common/resources/
Resources_|_getResourceAsStream(Ljava/lang/ClassLoader;Ljava/lang/String;)Ljava/io/InputStream;_||_
```

4.6 How are rules formed?

For every iteration of test script that is executed by JMeter, a profile is created. Each profile has many URLs. These URLs are selected randomly. We cluster these profiles as Good or Bad depending on the total elapsed time taken by the profile. A good profile is one which has a longer total elapsed time. We use the `weka.classifiers.rules.JRip` class to implement the rule learner. It is a propositional rule learner. The input parameters to the rule learner are a count of the number of times each URL occurs in the profile. The input to the WEKA is an Attribute Relation File Format file. ARFF file has two section the **Header** section and **Data** section. The Header section contains the relation name and the attributes. The format for the relation declaration is

```
@relation <relation-name>
```

The attribute declaration part contains the names of all attributes and the datatype. In our implementation, each profile generates an instance in the ARFF file. The attributes are all the possible URLs in the JPetstore and the data contains the number of times each URL occurs in

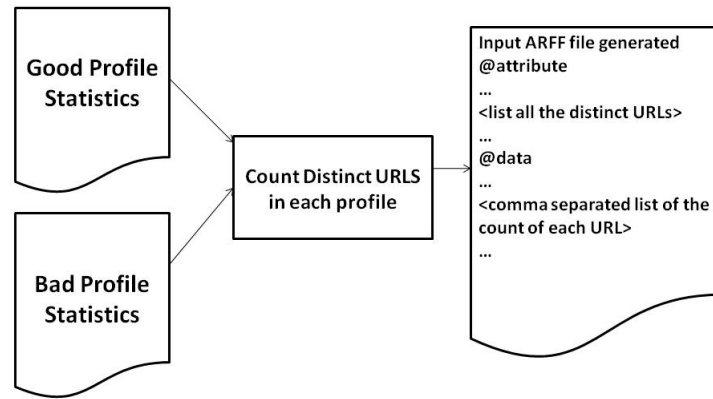


Figure 19. Creating the ARFF file as input to the Rule Learner

the profile. Figure 19 gives the illustration. The rule learner takes the count of URL as input parameters and therefore, the rules which are formed is of the form

`URL1_name > count1 & URL2_name < count2 --> Good`

Here `url1_name`, `url2_name` are readable forms of URL. To form rules in the above format, we map each URL to a readable form of the URL. We later map these readable forms of the URL with the count of number of occurrences of the URL in each profile. Let us consider an example. After running the application for a while, we collect 12 profiles. We cluster these profiles as Good or Bad. In our implementation, the profiles will be auto-clustered based on the total elapsed time of the profile. But the classes of these profiles can be reassigned if there is a need. After assigning classes (clustering) to these profiles, we learn the rules. We get a rule which says

```
(viewPrdct_F1-FW-01 <= 0) -> Bad
```

This rule will be passed to the JMXLoadScript module, which incorporates this rule in the next iteration of test scripts generation. If we split this rule, `viewPrdct_F1-FW-01` is the more readable form of the URL

```
http://localhost:8080/JPetStoreApp/shop/viewProduct.shtml?\\productId=FI-FW-01
```

and **0** is the count of occurrence of the URL in the profile. This rule implies that a profile where the URL

```
http://localhost:8080/JPetStoreApp/shop/ viewProduct.shtml?productId=FI-FW-01
```

does not occur at all, has lower chances of having a performance problem.

Figure 20 is a snapshot of the input ARFF file. The attribute declaration part lists all the readable URL names and the data type. The data type is numeric, since the data stores the count of the occurrences of the URL. The data declaration part lists the comma separated values of the count of occurrences of the URLs listed in the attributes. The columns in the data appear in the order of the attributes declared.

4.7 Profile Analyzer

The Profile Analyzer performs the analysis and clustering of Profiles. The rule learner is also a part of Analyzer. The clustered profiles are used to generate rules. When the application starts, it starts the Profile Repository, tomcat server, profile analyzer and jmeter scripts in that order. The Profile Repository creates a server socket that listens for clients on a port. When tomcat starts running, it runs the statically instrumented JPetstore. Once tomcat is up and running, profile analyzer starts. It has a GUI interface which lets the tester interact with the system. After the profile analyzer is started in an independent thread, JMeter test script

Profile No	Elapsed Time	Methods	Invocations	Attributes	Data in KB	Class	Exclude
1	3.2821484649E10	545	62320	0	0	Good	false
2	2.743257165E9	184	32398	0	0	Bad	false
3	6.28746146E9	196	49557	0	0	Bad	false
4	9.869082104E9	199	50682	0	0	Good	false
5	1.700336074E9	184	31151	0	0	Bad	false
6	3.862068783E9	185	42499	0	0	Bad	false
7	6.486096751E9	196	54664	0	0	Bad	false
8	1.8487741236E10	198	111317	0	0	Good	false
9	4.150207085E9	196	49468	0	0	Bad	false
10	1.2961663091E10	198	106448	0	0	Good	false
11	6.169299795E9	196	73700	0	0	Bad	false

Auto Cluster

Figure 21. Snapshot of the Profile Analyzer

generation is invoked. The JMeter tests are generated iteratively and the profiles generated are stored in the repository by the Profile Repository module. Once the iterations start, test engineer can start analyzing the profile by selecting **Control->Analyze->Start** from the drop down menu. This will initiate the system to read the profiles in the repository and extract various information from these files. During the profile analysis, the application collects the number of methods, number of invocations and the total elapsed time taken by each profile. We can stop the analyzer by selecting **Control->Analyzer->Stop**. To resume the analyzer we can select **Control->Analyzer->Resume**. Figure 21 is a snapshot of the UI screen of Profile Analyzer. Figure 22 shows the menu options for starting and stopping the profile analyzer.

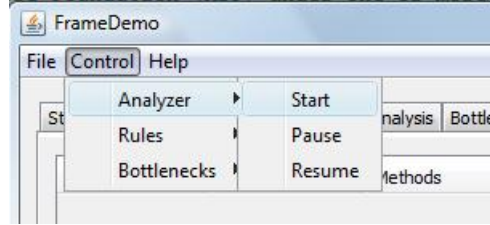
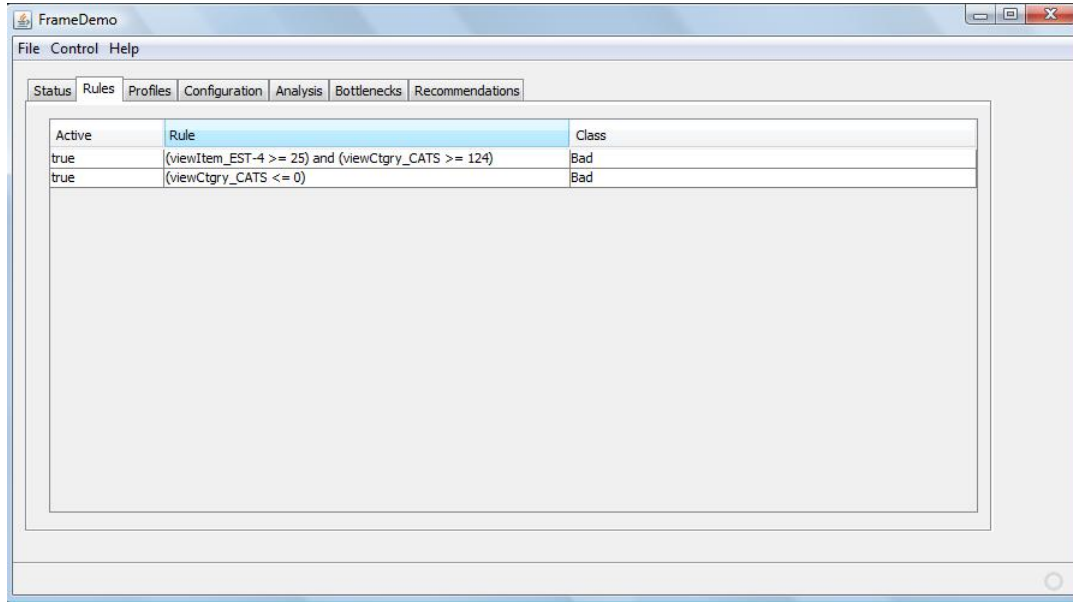


Figure 22. Menu Options to control Profile Analyzer

Once we have enough profiles to start learning rules, we can cluster the profiles. In our implementation, we need to have at least 10 profiles whose class have been assigned before we can start learning rules. Clustering the profiles can be done automatically by clicking the **AutoCluster** button or it can be done manually by the test engineer. The autocluster assigns the classes based on the total elapsed time. Consider 10 profiles have been analyzed. A profile would be assigned as Good by the auto cluster if the total elapsed time of the profile is greater than the average total elapsed times of the 10 profiles. It is our assumption that a profile which has longer elapsed time is prone to performance degradation issues during load testing. Hence we concentrate on areas which take more time. If the test engineer wishes to reassign the class of the profile, clicking on the assigned class will pull up a drop down menu. The menu contains all the possible values for the class.(Good, Bad, Unassigned) The test engineer can choose any of the value from the drop down to reassign the class of the profile.

4.8 Rule Generation

Profile Analyzer collects all the profile statistics and clusters the profiles as Good or Bad. The clustering can be done automatically or manually by the test engineer. Once we have clustered a minimum number of profiles (in our implementation, it is 10) rules can be generated



Active	Rule	Class
true	(viewItem_EST-4 >= 25) and (viewCtgr_CATS >= 124)	Bad
true	(viewCtgr_CATS <= 0)	Bad

Figure 23. Rules Generated by the System

from these clustered profiles. We use WEKA for rule generation. In our implementation, we use `weka.classifiers.rules.JRip` to implement the rule learning algorithm. JRip implements **Repeated Incremental Pruning to Produce Error Reduction (RIPPER)** (15). We pass the input ARFF file to the Classifier. The algorithm uses this training set to build classifiers and get rule set. The rule set is parsed and displayed on the Profile Analyzer GUI. Figure 23 is a snapshot of the rules which have been generated by the system. Figure 24 shows the menu option available to start learning the rules and to create a decision tree.

Once the rules are generated, they are fed-back into the input test script generation module. The input test script generation module, parses all the rules and generates a HashMap of the http requests and the count of its occurrence which have to be included in the next iteration of

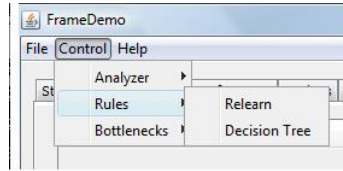


Figure 24. Menu Options for generating rules

input test scripts generation. The algorithm followed to implement this can be briefly described as follows :

- 1 Initialize RuleCountMap = {}, the hashmap which stores the http requests and the count of occurrence after parsing the rules. Repeat the following steps until all the Rules for the RuleSet have been iterated over
 - 1 If the rule is classified as Good rule, Get the URL name, count of occurrence of the URL and the relational operator of the current rule
 - 2 Check if the relational operator is GREATER or GREATER-EQUAL. If yes, check whether the RuleCountMap already contains the url name. If yes, check the count of occurrence stored in the map against the URL name. If the count of occurrence is lesser than the count in the current rule, store the current count of occurrence against the URL name in the RuleCountMap. If the hashmap does not contain the URL, store the current count of occurrence + 1 against the URL name in the RuleCountMap.
 - 3 Check if the relational operator is LESSER or LESSER-EQUAL. If yes, check whether the RuleCountMap already contains the url name. If yes, check the count of occur-

- rence stored in the map against the URL name. If the count of occurrence is greater than the count in the current rule, store the current count of occurrence against the URL name in the RuleCountMap. If the hashmap does not contain the URL, store the current count of occurrence - 1 against the URL name in the RuleCountMap.
- 4 Check if the relational operator is EQUAL. If yes, store the current count of occurrence against the URL name in the RuleCountMap.
- 1 If the rules is classified as Bad rule, Get the URL name, count of occurrence of the URL and the relational operator of the current rule
 - 2 Check if the relational operator is GREATER or GREATER-EQUAL. If yes, check whether the RuleCountMap already contains the url name. If yes, check the count of occurrence stored in the map against the URL name. If the count of occurrence is greater than the count in the current rule, store the current count of occurrence against the URL name in the RuleCountMap. If the hashmap does not contain the URL, store the current count of occurrence - 1 against the URL name in the RuleCountMap.
 - 3 Check if the relational operator is LESSER or LESSER-EQUAL. If yes, check whether the RuleCountMap already contains the url name. If yes, check the count of occurrence stored in the map against the URL name. If the count of occurrence is lesser than the count in the current rule, store the current count of occurrence against the URL name in the RuleCountMap. If the hashmap does not contain the URL, store the current count of occurrence + 1 against the URL name in the RuleCountMap.
 - 4 Check if the relational operator is EQUAL. If yes, store the current count of occurrence against the URL name in the RuleCountMap.

- 2 Iterate through the map and construct HTTP Requests to the TestPlan depending on the count of occurrence stored against each URL name. After a few iterations of relearning the rules, we observe that the rules will stabilize.

CHAPTER 5

RESULTS

5.1 Results

ASSIST tool is for the test engineers to locate those regions in the application which will require more resource allocation. Initially, the tool uses random load testing of the AUT. After some execution profiles of the application have been collected, the tool learns about the AUT and creates abstractions on the input space through rules. The rules are generated using machine learning algorithms.

The rules which are generated look like below

```
( url1_name > count1 ) and ( url2_name < count2)
```

Here, `url1_name` ,`url2_name` are more readable forms of the URLs and `count1`,`count2` are the number of occurrence of the URL in the profile. Rules are formed based on whether a profile is clustered as Good profile or Bad profile. We measure the method weights in each profile by collecting the method statistics. A profile is considered good profile if, for a set of profiles, the total time for all method invocations in that profile is greater than the average of the total time for all method invocations for all the profiles in the set. We ran our tool against the instrumented JPetstore application and collected execution profiles. To compare the effectiveness of our method with the normal random load testing, we had two sets of experiments. The first was completely random load testing which did not use any profile analysis or rule generation. Next, we performed the load testing with ASSIST tool.

TABLE I

Time Taken during Random Load Testing

Number of Transactions	Run 1 (in seconds)	Run 2 (in seconds)	Run 3 (in seconds)	Run 4 (in seconds)	Run 5 (in seconds)
25000	111.652	108.912	109.016	108.848	112.709
50000	225.981	210.781	211.53	218.106	220.463
75000	350.392	306.811	311.492	313.834	323.677
100000	525.858	406.25	402.275	413.45	469.856
125000	710.072	507.428	502.565	537.662	625.654

5.2 Random Testing

First we performed the completely random load testing. We choose random URLs and used JMeter Test Scripts to simulate those HTTP requests on the Tomcat web server where the instrumented web application is deployed. The execution profile of these HTTP requests were collected. In our implementation, we had the following values for the parameters :

- Number of Simulated Users : 5
- Number of iterations per User : 5
- Number of URLs chosen in each JMeter Test Script : 100

This experiment was repeated a couple of times and we calculated the time (in milliseconds) taken to complete the transactions. The table Table I shows the time taken to execute transactions for 5 different runs of load testing.

5.3 ASSIST Tool

Once we collected the results from the random load testing, in the next set of experiments, we use our tool to perform load testing. The parameters of the application run remain the same. We had the following values for the parameters :

TABLE II

Time Taken during ASSIST approach based Load Testing

Number of Transactions	Run 1 (in seconds)	Run 2 (in seconds)	Run 3 (in seconds)	Run 4 (in seconds)	Run 5 (in seconds)
25000	114.736	122.567	128.12	115.229	110.957
50000	395.952	401.289	417.813	511.923	217.027
75000	1347.582	2169.098	2219.494	1181.176	519.821
100000	3336.947	4699.773	4982.46	2319.03	2212.628
125000	6431.132	8611.381	6141.95	6351.225	4938.466

- Number of Simulated Users : 5
- Number of iterations per User : 5
- Number of URLs chosen in each JMeter Test Script : 100

Once the test scripts start executing the instrumented JPetstore application, execution profiles are formed. Each profile corresponds to one Test Script. We start the profile analyzer to analyze the execution profiles. The analyzer collects the profile statistics. The analyzer calculates the total number of methods called in the profile, total number of method invocations, total elapsed time for each test script and populates the datagrid on the Tool.

Once the profiles have been analyzed and the statistics have been calculated, it appears on the datagrid. Test engineer can cluster the profiles manually or use the AutoClustering button to cluster the profiles as Good or Bad. In our test runs, we collected 15 profiles, clustered them and generated rules. After the rule generation, we collected more profiles. These profiles were generated by test scripts which chose the URLs based on the rules which were generated. The table Table II shows the time taken to execute transactions for 5 different runs of load testing.

TABLE III

Average time Taken by Both Approaches

Number of Transactions	Random Load Testing (in seconds)	ASSIST Approach (in seconds)
25000	110.2274	118.3218
50000	217.3722	388.8008
75000	321.2412	1487.4342
100000	443.5378	3510.1676
125000	576.6762	6494.8308

We see that while performing random load testing it takes an average 576.67seconds to execute 125,000 transactions. But with the ASSIST approach, executing 125,000 transactions takes an average 6494.8308seconds. This implies that ASSIST picks up those regions of application for load testing which take longer time to execute. The requests which take longer time to execute have higher chances of turning out to create a performance bottleneck.

The plots show the comparison between the 2 approaches with respect to the time taken to execute certain number of transactions. The figures depicts how the ASSIST approach moves towards the slow region of the application. We also give a comparison of the average time taken by both the approaches. Figure 30, Table III. From the above results it is clearly visible that ASSIST approach steers the load testing towards those region which require more resource allocation. It helps the test engineer to find abstractions for input space partitioning for software performance testing.

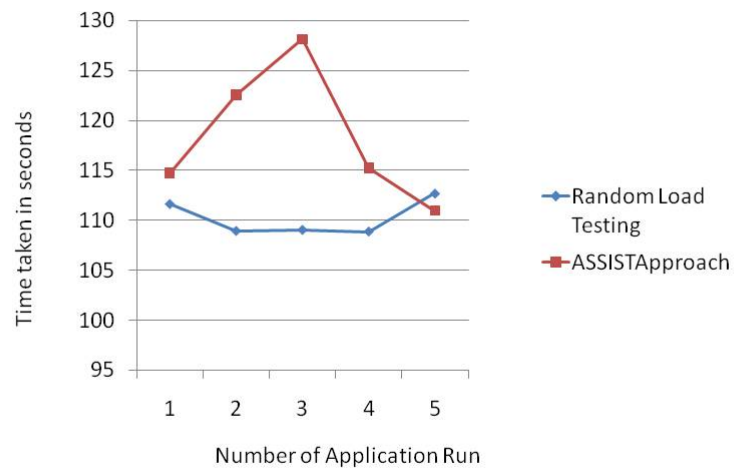


Figure 25. Comparison of the time taken to execute 25000 transactions

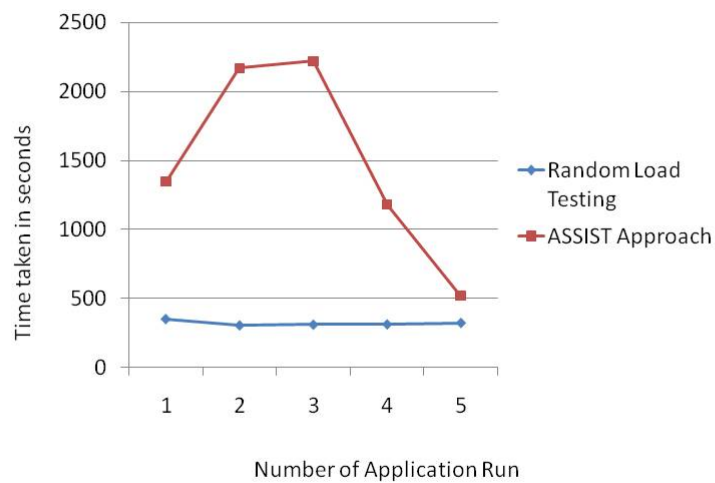


Figure 26. Comparison of the time taken to execute 50000 transactions

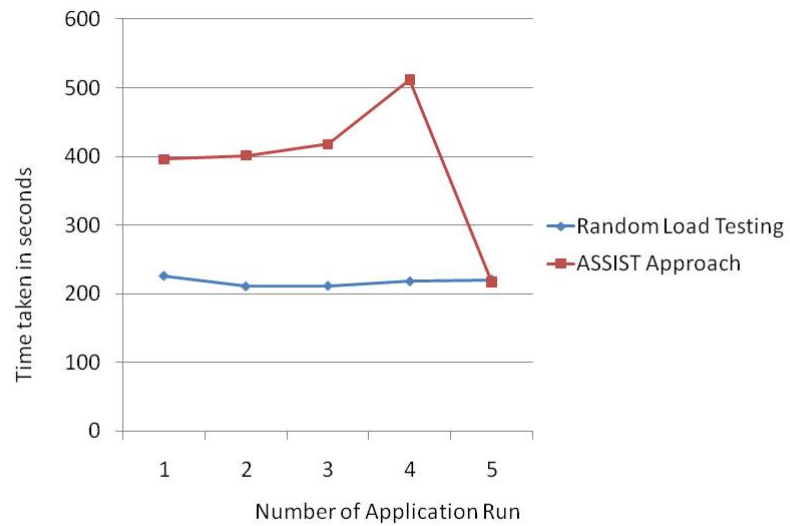


Figure 27. Comparison of the time taken to execute 75000 transactions

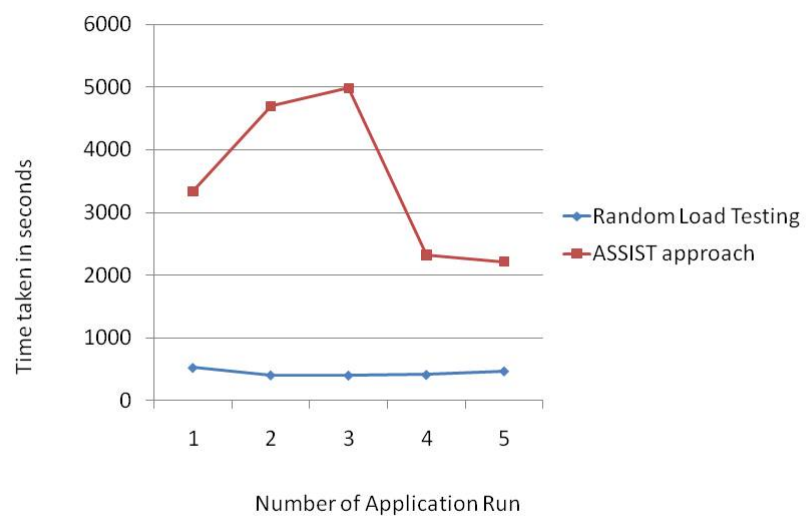


Figure 28. Comparison of the time taken to execute 100000 transactions

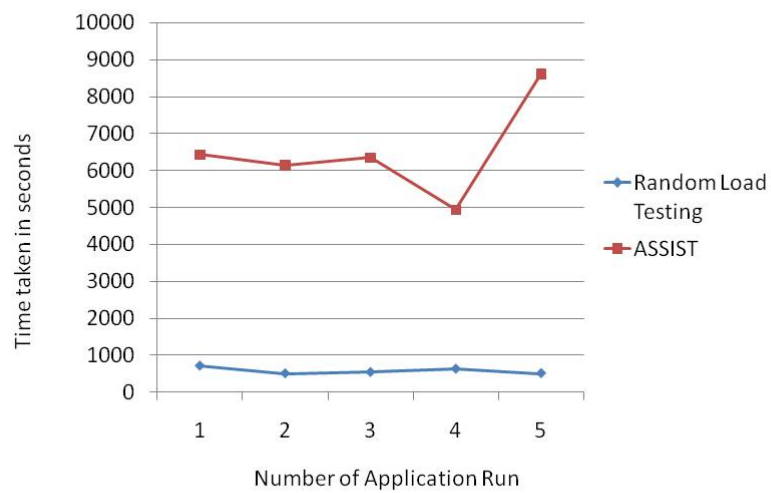


Figure 29. Comparison of the time taken to execute 125000 transactions

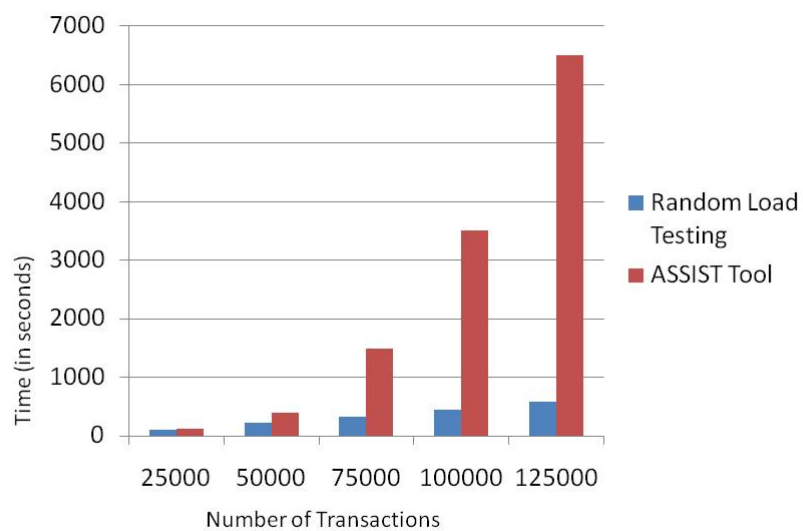


Figure 30. ASSIST steers the load testing toward the slow performing regions of the application

CHAPTER 6

RELATED WORK

6.1 Related Work

Numerous techniques have been proposed to automate regression testing. These techniques usually rely on information obtained from the modifications made to the source code. Some of the popular regression techniques include analyzing the program’s control-flow structure (9), analyzing changes in functions, types, variables, and macro definitions (12)(39), using def-use chains (29), constructing procedure dependence graphs (13)(51), and analyzing code and class hierarchy for object-oriented programs (40)(50). These techniques are not directly applicable to finding useful abstraction for testing, since regression information is derived from changes made to the source code.

Partition testing is a set of strategies that divides the program’s input domain into subdomains (subsets) from which test cases can be derived to cover each subset at least once. The goal of such a partitioning is to make sure that the resulting test set is a good representation of the entire domain (55). General guidelines on how to create an effective partition have been discussed on (25). Various systematic approaches have been proposed to partition the input space of programs as surveyed in (57). Some used specifications to create the partition (45), others used actual programs to conduct a control flow and data flow analysis (22), (23), (28), (35). In general, the input domain for partition testing is typically infinite, while it has a finite set of execution profiles. Closely related is the work by Dickinson et al (18), which use clustering analysis execution profiles to find failures among the executions induced by a set of potential test cases. Although we both used clustering techniques, our work differs in that we cluster the

execution profiles based on the length of the execution time and number of methods have been invoked, and we target the performance bugs instead of functional errors.

CHAPTER 7

CONCLUSION

7.1 Conclusion

In this thesis, we offer an implementation of the *ASSIST* approach. This approach starts with generating tests for a part of the application and grows by learning about the application, forming rules which steer the tests towards a region which demands more resource allocation. While the factors involved to determine whether a particular part of the web application is prone to performance degradation or not are numerous, we propose using the total elapsed time taken by the tests as a parameter to be considered.

There has been a lot of research on generating automated test suites (20),(21),(36). This thesis focuses specifically on creating automated test cases for load testing. We have built a tool that does automatic test generation. Although it starts off as a completely random test generation system, over time it learns certain rules from the various application runs and generates test suites based on those rules. The tool we built is adaptive, feedback-directed learning testing system. This tool is not only for automation of load tests but also helps in fault localization. For a commercial web application system, there can be huge number of resources (number of web pages on the web server) which are in the scope of performance testing. This tool helps the test engineer by narrowing down the location of occurrence of performance degradation issues.

7.2 Future Work

This research has left many questions unanswered. There is a lot of scope for further research on this topic.

Firstly, we have a very small scale implementation of the load testing. Our testing environment simulates 5 users performing the actions for 5 iterations. But in reality, load testing simulates thousands of users. We were unable to simulate such realistic testing environments due to resource constraints. One of the bigger challenges of further research would be to validate the results of this research using real world data.

Also, our implementation forms rules based on the count of number of URLs present in each profile. To form rules, we need to know all the possible URLs of the web application. Since we picked up a relatively small web application like JPetstore for our case study, we could list out all the possible URLs by simple manual inspection. For real world applications, there needs to be a module which implement a crawler which will collect all possible navigational paths of the application under test.

Another area of further research would be to try to form abstractions depending on the values of URL parameters instead of the count of the URLs in the test script. This would lead to input space partitioning depending on the value of URL parameters for different URLs. The attributes of the ARFF file which is fed to the rule learner will contain names of all possible URL parameter and the data section of the ARFF will contain the values it takes.

CITED LITERATURE

1. Embedding apache derby in tomcat and creating an ibatis jpetstore, <http://db.apache.org/derby/integrate/derbytomcat5512jpetstor.html>, 2004.
2. Jpetstore an online petstore, <http://sourceforge.net/projects/ibatisjpetstore/>, 2004.
3. Michael Achenbach and Klaus Ostermann. Engineering abstractions in model checking and testing. *IEEE Intl Workshop on SCAM*, 0:137–146, 2009.
4. Marcos Kawazoe Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, pages 74–89, 2003.
5. Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008.
6. Glenn Ammons, Jong-Deok Choi, Manish Gupta, and Nikhil Swamy. Finding and removing performance bottlenecks in large systems. In *ECOOOP*, pages 170–194, 2004.
7. Machine Learning Group at University of Waikato. Weka - data mining software in java, <http://www.cs.waikato.ac.nz/ml/weka>.
8. Alberto Avritzer and Elaine J. Weyuker. Generating test suites for software load testing. In *ISSTA*, pages 44–57, New York, NY, USA, 1994. ACM.
9. Thomas Ball. On the limit of control flow analysis for regression test selection. In *Proceedings of ISSTA-98*, volume 23,2 of *ACM Software Engineering Notes*, pages 134–142, New York, March2–5 1998.
10. Kent Beck. *Test-Driven Development: By Example*. The Addison-Wesley Signature Series. Addison-Wesley, 2003.
11. B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.
12. John Bible, Gregg Rothermel, and David S. Rosenblum. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):149–183, 2001.

13. D. Binkley. Reducing the cost of regression testing by semantics guided test case selection. In Gianluigi Caldiera and Keith Bennett, editors, *ICSM*, pages 251–263, Washington, October 1995.
14. Inc. Clarkware Consulting. Java package dependency analyzer that generates design quality metrics, <http://www.clarkware.com/software/jdepend.html>, 2009.
15. William W. Cohen. Fast effective rule induction. In *In Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.
16. IBM Corporation. Eclipse test and performance tools platform project, <http://www.eclipse.org/tptp/platform/documents/probokit/probokit.html>, 2006.
17. Wieger Cornelissen, Ad Klaassen, Aart Matsinger, and Gerhard van Wee. How to make intuitive testing more systematic. *IEEE Softw.*, 12(5):87–89, 1995.
18. William Dickinson, David Leon, and Andy Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE*, pages 339–348, 2001.
19. Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
20. Elfriede Dustin, Jeff Rashka, and John Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley, September 2004.
21. Mark Fewster and Dorothy Graham. *Software Test Automation: Effective Use of Test Execution Tools*. Addison-Wesley, September 1999.
22. Phyllis G. Frankl and Elaine J. Weyuker. An analytical comparison of the fault-detecting ability of data flow testing techniques. In *ICSE*, pages 415–424, 1993.
23. Phyllis G. Frankl and Elaine J. Weyuker. Provable improvements on branch testing. *IEEE Trans. Software Eng.*, 19(10):962–975, 1993.
24. Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. Mock roles, objects. In *Companion to OOPSLA '04*, pages 236–246, New York, NY, USA, 2004. ACM.
25. John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Trans. Software Eng.*, 1(2):156–173, 1975.
26. Scott Grant, James R. Cordy, and David Skillicorn. Automated concept location using independent component analysis. In *WCRE '08*, pages 138–142, Washington, DC, USA, 2008. IEEE Computer Society.

27. Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination testing strategies: A survey. *Software Testing, Verification, and Reliability*, 15:167–199, 2005.
28. Richard G. Hamlet and Ross Taylor. Partition testing does not inspire confidence. *IEEE Trans. Software Eng.*, 16(12):1402–1411, 1990.
29. Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM TOSEM*, 2(3):270–285, July 1993.
30. Jean Henrard, Vincent Englebert, Jean-Marc Hick, Didier Roland, Jean-Luc Hainaut, D. Rol, and J l. Hainaut. Program understanding in databases reverse engineering. In *In Proceedings of DEXA'98*, 1998.
31. Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements Engineering*. SpringerVerlag, 2004.
32. A. Hyvärinen and E. Oja. Independent component analysis: algorithms and applications. *Neural Netw.*, 13(4-5):411–430, 2000.
33. IEEE. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. Institute of Electrical and Electronics Engineers, January 1991.
34. Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automated performance analysis of load tests. In *ICSM*, pages 125–134, 2009.
35. Bryan F. Jones, David E. Eyres, and H.-H. Sthamer. A strategy for using genetic algorithms to automate branch and fault-based testing. *Comput. J.*, 41(2):98–107, 1998.
36. Cem Kaner. Improving the maintainability of automated test suites. *Software QA*, 4(4), 1997.
37. Cem Kaner. What is a good test case? In *Software Testing Analysis & Review Conference (STAR) East*. STAR, 2003.
38. Gregory M. Kapfhammer and Mary Lou Soffa. A family of test adequacy criteria for database-driven applications. In *ESEC/FSE-11*, pages 98–107, New York, NY, USA, 2003. ACM.
39. Jung-Min Kim and Adam A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE*, pages 119–129, 2002.

40. David C. Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, and Cris Chen. On regression testing of object-oriented programs. *The Journal of Systems and Software*, 32(1):21–31, January 1996.
41. Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.
42. Ian Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., 2009.
43. Thomas E. Murphy. Managing test data for maximum productivity. Technical report, Gartner, Stamford, CT, USA, 2008.
44. Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.
45. Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Commun. ACM*, 31(6):676–686, 1988.
46. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.
47. Simon Parsons. Independent component analysis: A tutorial introduction by james v. stone, mit press, pp. 193, isbn 0-262-69315-1. *Knowl. Eng. Rev.*, 20(2):198–199, 2005.
48. Apache Portals. ibatis jpetstore demo portlet metrics, <http://portals.apache.org/bridges/multiproject/jpetstore/jdepend-report.html>, 2007.
49. Apache Jakarta Project. The apache jakarta project - apache jmeter, <http://jakarta.apache.org/jmeter/>, 2004.
50. Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA*, pages 432–448, 2004.
51. Raúl A. Santelices, Pavan Kumar Chittimalli, Taweessup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Test-suite augmentation for evolving software. In *ASE*, pages 218–227, 2008.
52. Gary Sevitsky, Wim De Pauw, and Ravi Konuru. An information exploration tool for performance analysis of java programs. In *In TOOLS 01*, page 85. IEEE Computer Society, 2001.

53. Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. Degrees of separation: Multi-dimensional separation of concerns. In *ICSE*, pages 107–119, 1999.
54. Malcolm R. Westcott. *Toward a contemporary psychology of intuition. A historical and empirical inquiry*. New York: Holt Rinehart & Winston, Inc., New York, NY, USA, 1968.
55. Elaine J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *IEEE Trans. Softw. Eng.*, 17(7):703–711, 1991.
56. Elaine J. Weyuker and Filippas I. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Trans. Softw. Eng.*, 26(12):1147–1156, 2000.
57. Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

VITA

NAME: Aswathy Nair

EMAIL: anair6@uic.edu

EDUCATION: B.Tech, Information Technology, Anna University,
Chennai, 2008
M.S, Computer Science, University of Illinois at
Chicago, Chicago, 2011

WORK EXPERIENCE:

Summer 2011, Intern, Bank Of America

Summer 2010, Intern, VMWare

2009-2010, Part time Application Developer, Of-
fice of Vice Chancellor for Research, UIC

2008-2009, Application Developer, Inautix Tech-
nologies (Subsidiary of Bank of New York Mellon)