A Novel Tool for Resource Utilisation Regression Testing of JVM-based Applications

A thesis presented in partial fulfilment of the
requirements for the degree of

Master of Science
in
Computer Science

at Massey University, Manawatu, New Zealand

Fergus Hewson

2013

Abstract

In recent years, automated regression testing with tools, like JUnit, has become a cornerstone of modern software engineering practice. Automated testing focuses on the functional aspects of software. Automated *performance* testing is less popular. Where used, it is based on the principle of comparing measurements against static thresholds representing parameters such as system runtime or memory usage.

This thesis presents an alternative approach to automatically generate test oracles from system calibration runs. This approach is particularly useful to safeguard the investment made when software is manually fine-tuned. A proof-of-concept tool was developed that works for all applications that can be deployed on the Java Virtual Machine (JVM), and is capable of testing all properties that can be accessed through Java Management Extensions (JMX) technology.

Acknowledgement

It would not have been possible to write this thesis without the help and support of the people around me, to only some of whom it is possible to give particular mention here. In particular I would like to thanks my supervisors Jens Dietrich (Massey University) and Stephen Marsland (Massey University). Without your guidance and support this thesis would not have been possible.

Jens, your supervision over the past 3 years has been invaluable, your attitude towards Software Engineering is inspiring and I am thankful you have had you as a supervisor. Stephen, thanks for the encouragement over the past few months and your insights into machine learning and performing academic research.

A big thanks goes to my parents, Jayne and Brian, for supporting me during this thesis, without their help I would not be where I am today. Thanks also goes out to my fellow students in the computer science department at Massey University.

Contents

List of Figures **11**

List of Tables **13**

1 Introduction **17**
1.1 Software testing . . . 18
1.2 Overview . . . 19
1.3 Motivation Example . . . 19
1.4 Research Goals . . . 24
1.5 Road Map for Thesis . . . 25

2 Existing work **27**
2.1 Model-based approaches . . . 29
2.2 Measurement-based approaches . . . 30
2.2.1 Benchmarking . . . 31
2.2.2 Profiling . . . 32
2.2.3 Instrumentation . . . 34
2.2.4 Performance test drivers . . . 35
2.3 Summary . . . 35

3 Methodology **37**
3.1 Test Programs . . . 38
3.1.1 Java reference types . . . 39
3.1.2 Object leaker . . . 40
3.1.3 Classloading . . . 41
3.1.4 Consumer Producer . . . 43
3.2 Monitored Data . . . 45
3.2.1 Dynamic Properties . . . 46
3.2.2 Meta-data Properties . . . 46
3.2.3 Notifications . . . 46
3.2.4 Custom Properties . . . 49
3.3 Analysis . . . 49
3.4 Experimental Design . . . 50

4 Design **53**
4.1 Instrumentation . . . 53
4.2 Data Collection . . . 55
4.2.1 Data Storage . . . 56

4.3 Training Manager . . . 58
4.4 Dynamic-property Classifier . . . 58
4.4.1 Smoothing . . . 58
4.4.2 Time-point generation . . . 60
4.4.3 Classification Algorithms . . . 62
4.4.4 Simple Classifier . . . 62
4.4.5 Kalman filter . . . 63
4.4.6 Particle filter . . . 64
4.4.7 Bounds Matcher . . . 66
4.5 Meta-Data Classifier . . . 67
4.6 Notification Classifier . . . 68
4.7 Test-Run Classifier . . . 71

5 Experimental Results 73
5.1 Training Size . . . 73
5.2 Smoothing . . . 75
5.3 Time-point generation . . . 75
5.4 Bounds Matching . . . 76
5.5 Classifier tuning . . . 76
5.5.1 Simple Classifier . . . 77
5.5.2 Kalman classifier . . . 77
5.5.3 Particle Classifier . . . 78
5.6 Overall Results . . . 80
5.6.1 Java references . . . 81
5.6.2 Eden Space Memory used . . . 84
5.6.3 Heap Memory Committed . . . 85
5.6.4 Object leaker . . . 87
5.6.5 Scavenge garbage collector collection count . . . 87
5.6.6 Marksweep garbage collector collection count . . . 89
5.6.7 Class loading . . . 90
5.6.8 Compilation Time . . . 91
5.6.9 Permanent generation memory used . . . 92
5.6.10 Consumer Producer . . . 93
5.6.11 Process job count . . . 93
5.6.12 Buffer size . . . 95
5.7 Notification classifier . . . 96
5.7.1 Different example . . . 97
5.7.2 Similar example . . . 98

6 Evaluation 101
6.1 Future Work . . . 102

Bibliography 105

A Repeating experiments 117

B Test program code **119**
B.1 Java references 119
B.2 Object Leaker 120
B.3 Classloading 121
B.4 Consumer Producer 123
B.4.1 JMX 124
B.4.2 domain 126
B.5 Shut-down thread 128

C Analysis code **129**
C.1 Kalman Filter 129
C.2 Particle filter 130
C.3 Bounds matcher 132
C.4 Meta-data classifier 134
C.5 Notification classifier 135
C.6 Test-run classifier 137

Acronyms **139**

List of Figures

1.1 Proposed performance test life-cycle 20
1.2 Motivation heap-memory-used graphs 24

3.1 Classloading dependency diagram. 42

4.1 Design overview. 54
4.2 Training sequence diagram. 57
4.3 Connection sequence diagram. 59
4.4 Dynamic-property data flow diagram. 60
4.5 Affects of smoothing 61
4.6 Time-point generation example. 61
4.7 Notification classification algorithm diagram 70

5.1 Effects of training-set size 82
5.2 Effect of smoothing size on data and generated bounds. 83
5.3 Eden-space memory used graphs for Java reference types. 86
5.4 Heap-memory committed graphs for Java reference types 86
5.5 Scavenge GC collection count graphs for object leaker test-program. 88
5.6 Marksweep GC collection count graphs for object leaker test-program. 90
5.7 Compilation time graphs for class-loading test-program. 91
5.8 Permanent generation used graphs for class loading test program. . 93
5.9 Processed job count graphs for consumer producer test-program. . 95
5.10 Buffer-size graphs for consumer producer test-program. 96
5.11 Notification classifier results processed job count different example 97
5.12 Notification classifier results processed job count similar example 98

List of Tables

3.1 Java reference type experiments. 39
3.2 Object leaker experiments 40
3.3 Class loading experiments. 43
3.4 Consumer producer variation points 44
3.5 Consumer producer test program experiments. 45
3.6 Default dynamic-properties 47
3.7 Standard meta-data properties 48

4.1 Smoothing tuning test plan 61
4.2 Time-point generation tuning test 62
4.3 Simple classifier test plan. 63
4.4 Kalman filter test plan. 64
4.5 Particle filter parameters. 66
4.6 Bounds matcher test plan. 67

5.1 Training tuning test results 74
5.2 Smoothing tuning test results. 75
5.3 Time-point generator tuning test results. 76
5.4 Bounds matching tuning test results. 76
5.5 Simple classifier average method tuning test results. 77
5.6 Kalman filter average type tuning test results. 77
5.7 Kalman classifier process noise tuning test results. 78
5.8 Default parameter values for particle filter tuning tests. 78
5.9 Particle filter iterations tuning test results. 78
5.10 Particle classifier re-sample percentage tuning test results. 79
5.11 Particle classifier propagate method tuning test results. 79
5.12 Particle classifier weighting method tuning test results. 79
5.13 Particle classifier particle count tuning test results. 80
5.14 Overall classification results 81
5.15 Heap-memory-used rankings aggregated across all test programs. . 82
5.16 Java references test-program overall rankings. 84
5.17 Eden space memory used rankings for Java references test program. 85
5.18 Java reference types Eden space memory used example result. . . 85
5.19 Heap-memory committed rankings for Java reference test program. 85
5.20 Java reference types heap-memory-used example result. 87
5.21 Object leaker test-program overall rankings. 87
5.22 Scavenge GC collection count rankings for object leaker test program. 88
5.23 Object leaker scavenge GC collection count experiment result. . . 89

5.24 Marksweep GC collection count rankings for object leaker test-program. 89
5.25 Object leaker Marksweep GC collection count experiment result. . 90
5.26 Class-loading overall rankings. 90
5.27 Compilation time rankings for class-loading test program. 91
5.28 Class-loading compilation time experiment result. 92
5.29 Permanent generation memory used rankings for class-loading test-program. 92
5.30 Class-loading permanent generation used experiment result. . . . 93
5.31 Consumer producer test-program overall rankings. 94
5.32 Processed job count rankings for consumer producer test-program. 94
5.33 Consumer producer processed job count example result. 95
5.34 Buffer size rankings for consumer producer test-program. 96
5.35 Consumer producer buffer size experiment result. 96
5.36 Notification classifier positive data classification from different example 98
5.37 Notification classifier negative data results from different example 98
5.38 Notification classifier results for different notifications rate. 99
5.39 Positive data results from notification classifier similar example. . 99
5.40 Negative data results from notification classifier similar example. . 99
5.41 Notification classifier results for similar notifications rate. 99

Listings

1.1 Motivation example 22
1.2 Object Manager Interface 22
1.3 Strong object manager implementation 23
1.4 Soft object manager implementation 23
4.1 Simple classification algorithm 63
B.1 Weak object manager implementation 119
B.2 Object leaker test program 120
B.3 Class loading test program runner 121
B.4 Class loading test domain object interface 121
B.5 Class loading test program domain object 121
B.6 Class loading test program domain object factory 122
B.7 Classloading configuration object 122
B.8 Consumer Producer Algorithm 123
B.9 Consumer object definition. 123
B.10 Producer object definition. 124
B.11 Interface for ProductionMXBean 124
B.12 Interface for CounterMXBean 124
B.13 ProductionMBeanInfo object definition. 125
B.14 Counter object definition. 125
B.15 Student object definition. 126
B.16 StudentFactory object definition. 127
B.17 WebSiteBuilder object definition. 127
B.18 Shutdown Thread object definition. 128
C.1 The Kalman Filter classification algorithm. 129
C.2 Method used to calculate the process noise for the Kalman filter. . 130
C.3 The Particle Filter classification algorithm. 130
C.4 Effective sample size evaluation method. 131
C.5 Method used to re-sample particles with respect to their weightings. 131
C.6 Method used to update the particles weight after each iteration. . . 132
C.7 Method used to update particle weights. 132
C.8 Simple Bounds Matcher Algorithm, matches time-stamps to the nearest bounds. 132
C.9 Interpolation Bounds Matcher Algorithm, uses interpolation to generate bounds for a given time-stamp based on the bounds the time-stamp falls between. 133
C.10 Method to interpolate a Y value between two points given an X value. 133
C.11 Meta-data training algorithm 134
C.12 Meta-data classification algorithm 135
C.13 Notification Training Algorithm 135

C.14 Notification Classification Algorithm 136
C.15 Test-Run Classifier Training Algorithm 137
C.16 Test-Run Classification Algorithm 137

Chapter 1

Introduction

With the advent of internet technologies, businesses now operate in a fast paced environment where customers expect their needs to be met in a timely fashion. Performance has become a vital aspect of today's software systems, and often businesses live and die by the performance of mission-critical software applications (Molyneaux, 2009). However, performance testing is a neglected practice and often left to the end of the development life-cycle or even after deployment. Smith and Williams (2002) identified that neglecting performance testing exposes companies to serious risk due to the costs of performance failures. This can lead to damaged customer relations, lost income, and time and budget overruns.

Performance is becoming a more important non-functional requirement for applications. There is an expectation by end users that software will execute in a timely manner and not have any significant performance problems. The introduction of cloud computing has lead to multi-tenant platforms where applications are required to execute on the same system and compete for runtime resources like processor time and memory. More applications are expected to be able to run on platforms where runtime resources are scarce, like tablets, smart-phones, embedded devices and single-board computers like the Raspberry Pi (Raspberry Pi Foundation, 2013). The rise of digital and mobile communication has made the world become more connected, networked, and traceable and has typically lead to the availability of such large scale data sets (Big Data) (Harrison Rainie and Wellman, 2012). The requirement for processing Big Data places more pressure on software performance increasing the need for software performance testing tools.

Performance testing is a form of testing that aims to determine how well a system performs in a test environment. It differs from functional testing in that it validates quality attributes of a system as opposed to testing it's correctness. Performance testing is often neglected during development and performed towards the end of the development life-cycle, as a release exit check or during application deployment (Molyneaux, 2009). Ignoring software performance testing incurs a significant risk (Molyneaux, 2009), which may lead to either a delay in release or releasing a product with performance defects. Either of these scenarios will have a significant negative impact on the business.

1.1 Software testing

Software testing is often focused on checking whether the application does what it is expected to do. The most common method of code validation is Unit testing. A unit test is designed to test an individual aspect of the system and show that it is correct. Unit testing is a common software engineering practice and has many benefits to the development of a software system. It helps to identify problems early in development, provides a safety net for refactoring and code changes and simplifies integration testing (Unhelkar, 2012). Unit testing is often associated with agile methodologies.

Over the past decade, the agile software movement has had significant impact on software engineering research and practice (Madeyski, 2010). The Agile Manifesto (Beck et al., 2001) states that agile development is focused on quick responses to change and continuous development. A core practice of agile methodologies is Test-Driven-Development (TDD), where engineers write tests before developing related pieces of production code (Beck and Andres, 2004). The TDD life-cycle, as proposed by Beck, begins with adding a test which must fail because the tested feature has not been implemented. Next, all tests are run to determine if the new test fails and does not mistakenly pass without requiring new code. Code is then quickly developed that causes the test to pass, at this stage code quality is not important. All tests are then re-run and if they pass the engineer can be confident the new code meets the tested requirements. The code is then re-factored and cleaned up to meet quality standards. Tests are then re-run to ensure refactoring has not introduced bugs into the program.

Erdogmus et al. (2005) propose several reasons for the adoption of TDD. It provides instant feedback as to whether new functionality works as intended and whether it interferes with existing functionality. TDD aids low-level design decisions, such as which classes and methods to create, how they will be named, what interfaces they will possess, and how they will be used. Having up-to-date tests provides assurance of the quality of the software. TDD supports refactoring and maintenance activities by providing a safety net for engineers so they can introduce changes without the risk of breaking existing functionality. Any introduced errors will be highlighted by the test suite (Madeyski, 2010). TDD supports the agile principles of constant integration and automated testing, and improves a team's ability to respond to changes in requirements (Madeyski, 2010). Most automated testing is focused on the evaluation of software correctness and non-functional tests are usually performed manually by specialist engineers (Unhelkar, 2012).

There are many tools available to perform performance testing. Molyneaux (2009) distinguishes two groups of performance indicators which tools utilise. Service-oriented indicators are availability and response time; they measure how well an application is providing a service to the end users. Efficiency-oriented indicators are throughput and utilisation; they measure how well an application makes use of the application landscape. Service-oriented indicators can be test using benchmarking and load testing to assess availability and response time. However there is a distinct lack of tools to perform testing for efficiency orientated indicators.

1.2 Overview

The aim of this thesis is to design a framework for the detection changes in a software system's performance, focusing on efficiency-oriented performance indicators like the system's resource utilisation. To fulfil this aim, Buto was developed, a tool to perform resource utilisation regression testing for JVM based applications. The tool includes a framework for instrumenting and monitoring program resources, and as well a classifier that can generate performance test oracles that can identify changes in the performance characteristics of the program under test. The framework aims to bridge the gap between unit-testing and performance testing by allowing engineers to write performance tests, create a model of the tests runtime performance and then compare that model with future runs of the test. Future tests that do not conform to the model will be flagged as an anomaly for performance engineers to further investigate.

Buto was evaluated by a set of test programs that exercise important resource utilisation characteristics of Java programs. Each of these scenarios were designed to be configurable to produce different runtime resource utilisation signatures, Buto was assessed on how well it could identify these changes. Three classification methods were explored, as well as different methods for smoothing, collating training-run data and model matching were developed and tested against each other. Tests were ranked based on the Mathews Correlation Coefficient (MCC), a balanced measure for binary classification. Tuning tests were performed for each of the stages in the analysis pipeline and the classifications algorithms. Finally, the best configurations of the classification algorithms were tested against each other to determine the best classification method.

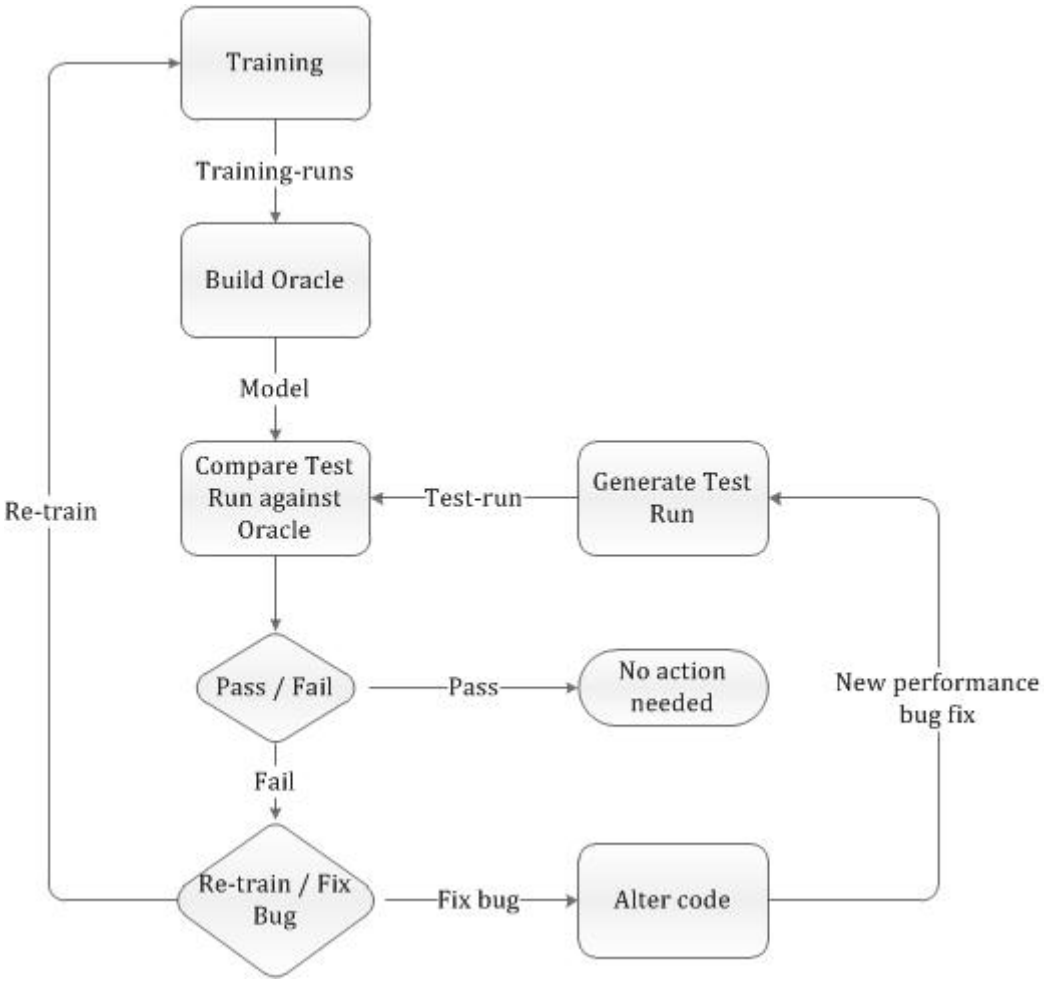
The proposed work flow for using Buto is outlined in Figure 1.1. Performance testing can't reasonably start until a feature passes the TDD unit and functional tests (Johnson et al., 2007). The process has to be initiated by generating a training-set that represents the current state of the performance test. This is done by running performance tests many times and collecting performance counters during execution. The training-set then can be used to build an oracle to compare future performance test runs against. When a significant alteration in code has been made, the performance test can be re-run and compared against the oracle. If the test run passes then there was no significant difference between the test run and the training-run set. If it fails then there are two options; make alterations to the code to fix the regression or accept the regression and generate a new oracle. If an attempt to fix the regression has been made a new test run should be generated and compared against the oracle to ensure the performance regression has been fixed.

A framework for runtime resource utilisation regression testing will likely provide many of the benefits of TDD. It will allow code changes to be made quickly, help engineers better understand their code, provide instant feedback when performance regressions have been introduced and reduce the amount of manual performance testing required.

1.3 Motivation Example

An example has been provided to illustrate the need for a framework to perform automated runtime resource utilisation regression testing. The example, written in

Figure 1.1: Proposed performance test life-cycle. To begin evaluating a performance test, first a training set needs to be generated. After training the data is used to build a model of the expected resource utilisation. The model can then be used to classify future tests. If a test fails the classification engineers can choose to accept the change and re-train a new oracle or fix the performance defect.



Java, uses two methods for keeping references to objects. Different references are treated differently by the garbage collector causing alterations in the programs' utilisation of heap memory. The difference between the two methods is a small change in code that produce a significantly different runtime signature. Figure 1.2 shows the heap memory used for the two tests. Both graphs have similar minimum and maximum levels as well as similar distribution. Traditional methods of comparing these two graphs, like applying static thresholds, would not be able to differentiate between the two.

The example takes advantage of different methods to keep references in Java and how the garbage collector interacts with the objects managed by these references. In Java there are four methods of keeping references to objects; strong, soft, weak and phantom references. A strong reference is the standard reference created when using assignment statements. An object that has a strong reference to it will not be garbage collected. Soft, weak and phantom references are used to keep references to objects that may be garbage collected depending on conditions monitored by the garbage collector. This example uses data structures that are typical when implementing caching e.g for Object-Relational Mapping (ORM) frameworks.

This test example compares the use of strong references and weak references, to illustrate the affect on utilisation of heap memory. The strong referencing method uses an instance of `java.util.List` to keep references to generated objects, clearing the list when it becomes too large. When the list is cleared the objects are de-referenced and are free to be collected during the next garbage collector cycle. The second method uses a list of soft references to manage objects. An object that is only reachable through soft references is termed softly reachable. A softly reachable object may be collected at any time depending on the collection algorithm. The only guarantee is that a softly reachable object will be cleared before the virtual machine throws an `OutOfMemoryError`. The difference between the two methods is minimal, but it has a significant impact on the resource utilisation of the performance test.

Listing 1.1 contains the main code for the example. Parameters are used to set the running time of the example, the object generation rate and the referencing method to use. The program enters a loop which generates objects and passes them to the selected object manager. Object generation is offset by sleeping after the set number of objects are created. The program executes until the shutdown thread fires and terminates the Virtual Machine (VM). The only parameter changed in this example is the object manager used.

```
public class Example {
  public static void main(String[] args) throws InterruptedException {
    final int executionTime =  60000;
    final int objectCount = 1000;
    final int sleepInterval = 100;
    final int strongMaxListSize = 80000;
    final String objectManagerType = args[0];
    ObjectManager objectManager = null;
    // Switch for Object manager type
    switch(objectManagerType){
    case "strongObjectManager": objectManager = new StrongObjectManager(
    strongMaxListSize);break;
    case "softObjectManager": objectManager = new SoftObjectManager();break;
    default:return;
    }
    // Thread to shut down the JVM after executionTime
    new ShutdownThread(executionTime);
    // Run the example
    run(objectManager, objectCount, sleepInterval);
  }
  public static void run(ObjectManager objectManager, int objectCount, int
    sleepInterval) throws InterruptedException{
    while(true){
      // Generate objects and add to manager
      for(int i = 0 ; i < objectCount ; i++)
        objectManager.addObject(generateObject());
      // Slightly offset the leaking
      Thread.sleep(sleepInterval);
    }
  }
  private static Object generateObject(){
    StringBuilder sb = new StringBuilder();
    for(int j = 0 ; j < 100; j++)
      sb.append( " " + System.currentTimeMillis());
    return sb.toString();
  }
}
```

Listing 1.1: Motivation example runner. Selects an object manger based on command line parameters and adds objects to it until the shutdown thread closes the VM.

Listing 1.2 shows the interface for an object manager. An object manager has two methods; one to add a new object and the other to get a list of currently held objects. Both of the examples used will implement this interface, but use different methods to keep references to objects. The different methods interact differently with garbage collector and cause different resource utilisation signatures.

```
public interface ObjectManager{
  public void addObject(Object obj);
  public List<Object> getObjects();
}
```

Listing 1.2: Interface for classes that manage a list of objects.

The strong object manager is outlined in Listing 1.3. The method uses strong references to hold onto objects, releasing them when the number of objects gets to

large. It requires a value for the max size of the list to be passed into the constructor, this value will specify how large the list can get before it is cleared and the objects are de-referenced for garbage collection. The collection `java.util.ArrayList` is used to keep strong references to objects. Before each addition the size of the list is checked and cleared if the list exceeds the maximum size.

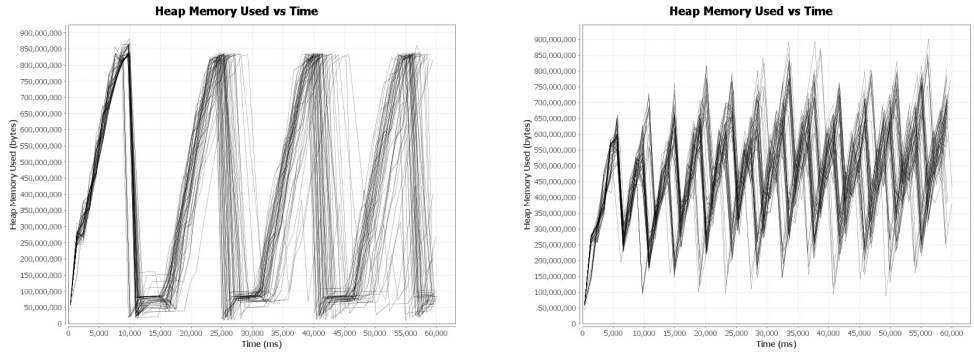
```
public class StrongObjectManager implements ObjectManager{
  private List<Object> referenceList = new ArrayList<Object>();
  private int maxSize;
  public StrongObjectManager(int maxSize){
    this.maxSize = maxSize;
  }
  @Override public void addObject(Object obj) {
    if(referenceList.size() > maxSize){referenceList.clear();}
    referenceList.add(obj);
  }
  @Override public List<Object> getObjects() {
    return referenceList;
  }
}
```

Listing 1.3: Strong object manager implementation. Uses strong references to hold onto objects, releasing them when the count gets too high.

Listing 1.4 outlines the soft object manager. It uses soft references to manage objects. A soft reference is a wrapper for an object which marks it as softly reachable for the garbage collector. The only guarantee regarding the collection of softly reachable objects is they will be collected before an `OutOfMemory` error is thrown. Before returning the list of managed objects all of the soft references need to be checked for validity. Every 1000 additions the list of soft references is checked for validity, if an object has been garbage collected the soft reference is discarded to prevent a memory leak.

```
public class SoftObjectManager implements ObjectManager{
  private List<SoftReference<Object>> softList = new ArrayList<SoftReference<
    Object>>();
  @Override public void addObject(Object obj) {
    softList.add(new SoftReference<Object>(obj));
    if(softList.size() % 10000 == 0){
      Iterator<SoftReference<Object>> iter = softList.iterator();
      while(iter.hasNext()){
        SoftReference<Object> ref = iter.next();
        if(ref.get() == null){iter.remove();}
      }
    }
  }
  @Override public List<Object> getObjects() {
    List<Object> outputList = new ArrayList<Object>();
    for(SoftReference<Object> softRef : softList){
      Object obj = softRef.get();
      if(obj != null){outputList.add(obj);}
    }
    return outputList;
  }
}
```

Listing 1.4: Object manager implementation using soft references.



(a) Soft memory manager heap-memory-used vs time

(b) Strong object manager heap-memory-used vs time

Figure 1.2: Plots of heap-memory-used for the tests of soft and strong memory references. It can be seen that although the maximum amount of memory used is fairly similar, the pattern of how it is used changes significantly between the two implementations.

Figure 1.2 shows plots of the heap memory consumption as the two implementations are executed using the script shown in Listing 1.1, with 50 runs shown in each graph. It is easy to see that in both cases the maximum amount of memory used is around `800,000,000` bytes, but the pattern of that memory usage over time is very different. Thus, performance testing with a static threshold would not be able to capture the difference in performance between these two scenarios. It can also be seen that there is a definite general pattern of performance for both implementations. However, there are some runs that appear to be a little different. The distinctive pattern of increasing and then decreasing memory usage is caused by the garbage collection process, and the fact that it is not entirely regular is due to the slightly non-deterministic nature of the Java garbage collector.

There is a noticeable difference between the two examples despite them having roughly the same max heap-memory-used. The soft object manager uses less heap memory in the initial stages of the test but increases to the level of the strong object manager towards the end. This difference in resource consumption will not be identified by traditional performance testing methods, but is significant enough to impact the programs performance. Being able to identify these changes in performance will help engineers reduce their program's footprint which is important for programs which run on multi-tenant systems or run in resource-restricted environments.

The example demonstrates how a small change in code can have a significant effect on the resource utilisation of a performance test. A tool that used a static threshold would not be able to distinguish between these two programs. This illustrates the need for a tool that can identify these changes in a performance test's resource utilisation and provide useful feedback to engineers to help them improve the quality of their software.

1.4 Research Goals

The main aim of this thesis is to produce a framework to detect changes in a program's resource utilisation. To explore this concept a tool called Buto was

developed, a framework for resource utilisation regression testing for JVM base applications. Through Buto this thesis aims to achieve the following research goals.

1. Identify machine learning techniques that are suitable for generating robust statistical test oracles.
2. Identify a good set of benchmarking scenarios that illustrate important runtime features.
3. Determine the robustness of the machine learning techniques relative to the set of benchmarking scenarios.

1.5 Road Map for Thesis

The need for a tool that can perform resource utilisation regression testing has been outlined in the introduction. To support this, an example has been shown that demonstrates the weaknesses of traditional methods. The rest of this thesis is organised as follows; existing work is presented (Chapter 2), methodology of research (Chapter 3), implementation and design (Chapter 4), experimental results (Chapter 5) and evaluation (Chapter 6).

Chapter 2 explores existing software performance methodologies and tools. The two branches of software performance engineering are explored, with the measurement-based approach covered in depth. This aims to identify the lack of resource utilisation regression testing in both industry and academic research.

Chapter 3 identifies the methodology used to develop Buto, outlining test programs, monitored properties, analysis framework and the experimental design.

Chapter 4 discusses the design and implementation of Buto. The methods used for instrumentation of resources, data collection and data storage are outlined. The data analysis pipeline is then described in detail.

Chapter 5 presents the experimental results. First, the dynamic-property classifier tuning experiments are presented. This involves determining the training and test set sizes and identifying the best configurations for smoothing, time-point generation and bounds-matching. Then tuning tests for each classifier are presented, determining the optimal configuration for each classifier. The overall results are then presented along with detailed results for each test program. Finally the notification classifier evaluation is presented.

Chapter 6 evaluates the results against the aims of this thesis before presenting future work.

Chapter 2

Existing work

In this chapter the key concepts of software performance engineering will be discussed in order to provide a foundation for the design and implementation. Woodside et al. (2007) outlined two general approaches to software performance engineering: an early-cycle predictive model-based approach, and a late-cycle measurement-based approach. This thesis is concerned with a measurement-based approach for assessing the runtime performance of a system, however the predictive model-based approach will be touched on. It has been acknowledged recently that a paradigm shift towards performance analysis is necessary to improve software performance (DeRose and Poxon, 2009) as well as the difficulties associated with detecting performance regressions in non-deterministic software environments (Georges et al., 2007a).

Definition: *Software Performance Engineering (SPE) represents the entire collection of software engineering activities and related analyses used throughout the software development cycle, which are directed to meeting performance requirements* (Woodside et al., 2007).

The standard practice for software testing is Unit testing which is usually supported by some form of unit testing framework. Most languages have an implementation of a unit testing framework available, JUnit (Junit-team, 2013) is the Java implementation. Unit tests are concerned with testing a program for correctness, ensuring it does what it is expected to. Unit tests are not performance aware, however some libraries allow for benchmarked unit tests to be written. Benchmark unit tests require engineers to choose a limit on a test's execution time, which if it exceeds will cause the test to fail.

Regression testing is a form of software testing that ensures changes made to software, such as adding, removing or modifying features, does not adversely affect other features of the software (Rothermel et al., 2001). Regression testing can involve re-running all or some of the test cases generated at earlier stages of development and can include benchmarked performance tests. It provides a safety net for engineers when making code changes, allowing them to act with a higher level of impunity knowing that any errors introduced will be identified by the regression testing suite.

A set of common problems surrounding performance issues with benchmarking Java applications is presented by Georges et al. (2007a). The non-determinism of the JVM is outlined with all the factors that affect it. These include the VM, heap size parameters, Just-In-Time Compiler (JIT) compilation and optimizations, thread scheduling, Garbage Collector (GC) scheduling and various system effects.

The current state of benchmarkers are then surveyed, stating there is a wide range of Java performance in use at the time. Only a small minority of papers surveyed (4 out of 50) used confidence intervals to characterize variability across test runs.

Georges et al. (2007a) also addressed experimental design, evaluating the different methodologies : One VM invocation versus multiple VM invocations, including compilation versus excluding compilation, forced GCs before measurement, multiple hardware platforms, multiple VM invocations and back-to-back vs interleaved measurements. They advocate two methods for performance evaluation. For start-up performance evaluation they advise to make multiple measurements where each measurement comprises a single JVM invocation and a single benchmark iteration. These measurements are then used to generate confidence intervals for the benchmark. For steady state performance they advise to take multiple measurements where each measurement comprises one VM invocation and multiple benchmark iterations. In each of these measurements, performance data is collected for different iterations once performance reaches steady-state, i.e., after the start-up phase. Confidence intervals are then computed using multiple benchmark iterations across multiple VM invocations.

A tool called JavaStats (Georges et al., 2007b) has been proposed that allows Java benchmarks to be run automatically until a sufficiently narrow confidence interval is obtained for a given performance metric. The factors presented in this paper are important to ensure that tests are conducted in environments that are as similar so they can be comparable, this was an important consideration when developing the Buto test life-cycle.

Cray Performance Analysis Tools are a set of tools for performance analysis of High Performance Computing (HPC) applications proposed by DeRose et al. (2008). The system they propose is aimed at addressing performance issues in peta-scale computing systems. In the introduction they propose a five-phase cycle for typical performance analysis,

1. Program Instrumentation
2. Measurement of predefined specific events during execution
3. Post-mortem and user-controlled analysis of the performance data
4. Presentation of these data via textual and graphical tools
5. Optimization of the program and its data structures under control of the user

The most relevant aspects to this thesis are program instrumentation, measurement during execution and post-mortem analysis of performance data.

Performance tuning is the end result from performance analysis. It involves altering code, changing configurations to optimize performance for the resources the application has available. Empirical tuning has been used as the standard approach to tuning applications to improve application performance in different scenarios (Pan and Eigenmann (2006); Clint Whaley et al. (2001)). Rahmen et al. proposed an automated empirical tuning framework that can be configured to optimize for both performance and energy efficiency (Rahman et al., 2011). Wang et al. (2012) use automatic performance tuning to improve Transactional Memory(TM) using static and dynamic properties to select the best TM algorithm at runtime. Automatic performance tuning for multi-core applications has been a recent field of interest (Karcher et al. (2009); Karcher and Pankratius (2011);

Ţăpuş et al. (2002)). Agakov et al. (2006) applied machine learning techniques to iterative compiler optimizations to adapt compiler optimizations based on learnt models. The aim of this thesis is to develop a new method to detect performance anomalies which can be used to tune applications, however, the tuning aspect is outside the scope of this thesis.

2.1 Model-based approaches

The model-based approach to software performance engineering uses performance models early in the development life-cycle and uses results from these models to adjust architecture to meet performance requirements (Woodside et al., 2007).

Design artefacts like Unified Modelling Language (UML) models are used to build and generate an executable model of the system that can be tested for performance issues. Results from these models are used to adjust the architecture and design before implementation has begun. Fixing performance bugs at the design stage of development is significantly less expensive than addressing the issue once features have been implemented. Commonly used performance models are queueing networks, stochastic Petri nets, stochastic process algebra, and simulation models (Balsamo et al., 2004).

In SPE (Smith and Williams, 2002), performance properties are specified as part of a model (usually either UML or one of its extensions, or an Architecture Description Language (ADL). Some of these models have been standardised by the Object Management Group (OMG) (Object Management Group consortium (2003b,a)). The idea is to use these models to validate the performance of a system either by using static analysis of this model (Cortellessa et al., 2005), or by simulating a system (Arief and Speirs (2000); Marzolla and Balsamo (2004); Siegmund et al. (2012)). This is different to the approach presented in this thesis: in SPE, performance characteristics and test oracles are explicitly defined on an abstract (model) level, whereas we generate oracles from running systems that have been manually fine-tuned. While SPE is conceptually more aligned with Model-Driven Engineering (MDE), this thesis fits more into agile, test-driven development practises.

Babka et al. (2010) suggested to completely automate the process of generating systems from models. As models are abstractions, there can be more than one system that complies to a certain model. Babka et al. propose to evaluate performance predictions by checking multiple randomised systems derived from the same model. This is somewhat related to the approach used in this thesis, as a statistical approach is used. However, in this thesis a test oracle is extracted from multiple runs of the same system, while they generate and simulate multiple systems from one model.

Anti-patterns are a common software engineering tool that describe commonly occurring solutions to problems which generate negative consequences (Neill and Laplante, 2005). Anti-pattern based approaches have been used to identify common performance defects and offer solutions to engineers (Arcelli et al., 2012; Smith and Williams, 2003). (Xu, 2010) proposed using a Layered Queueing Network (LQN) model and a rule based approach to identify performance bottlenecks in the design stage of development. Parsons (2007) identifies performance anti-patterns in component base systems, specifically Enterprise Java Beans (EJB). The approach is applied at the code level of software to identify performance design

and deployment anti-patterns. Wert et al. (2013) proposed Problem Diagnostics (PPD) that systematically searches for well-known performance problems (anti-patterns) within an application. The anti-pattern approach is useful at identifying common architectural performance defects early in development but is less applicable to the later stages of development when performance requirements need to be met.

2.2 Measurement-based approaches

The measurement-based approach is the most common form of software performance engineering. It applies testing, diagnosis and tuning late in the software development life-cycle when the system under development can be run and measured (Woodside et al., 2007).

Measurement-based software performance engineering is used to ensure that the system meets performance specifications as well as to help build and validate performance models. Measurement-based methods require a program be instrumented with probes to monitor program behaviour. When the software executes, the probes collect data about the state of the program which is used to generate profiles of the software's execution. There has been a lot of research into efficient methods for instrumenting program resources for profiling (Dmitriev (2004); Binder et al. (2007); Caserta and Zendra (2011)).

GCviewer (Tagtraum Industries, 2011) is a Java performance assessment tool that uses data produced from the JVM's GC log file to visualise garbage collector activity. GCViewer can also calculate performance related metrics such as throughput, longest pause and accumulated pause. The main object of the tool is to aid in choosing good tuning parameters for the garbage collector and heap sizes. The GCViewer does not provide a means to perform regression tests on the data collected from the GC log files.

A feature of Java is the ability to take heap-dumps of an executing JVM to aid in detecting performance defects. Tools that provide visualisations of Java heap-dumps include the Java Heap Analysis Tool (JHAT) (Oracle Corporation, 2011b), IBM Heap Analyzer (IBM Developer Works, 2013) and the Eclipse Memory Analyzer (Eclipse Foundation, 2013). These tools parse heap-dumps and output reports that allow engineers to identify causes of memory leaks.

Mostafa and Krintz (2009) propose a framework for "performance-aware" repository and revision control for Java programs. Their system tracks the behavioural differences across revisions to provide feedback as to how changes impact performance of the application. Their regression analysis technique is based on Calling Context Tree (CCT) which looks for differences in the call trees of two executions. This tool is very similar to aims of Buto but the data the method used to detect performance regressions is different.

Lee et al. (2012) proposed a framework for performance anomaly detection and analysis for Database Management System (DBMS) development. They applied the concept of statistical process control to a set of performance metrics that are relevant to the database performance. If an anomaly was detected in these performance metrics they use differential profiling to investigate the root cause of the detected anomaly.

Another approach to automated performance analysis and tuning is presented by Cong et al. (2012). They build a framework that combines expert knowledge,

compiler information and performance research into performance diagnosis and solution discovery. They define performance patterns using expert knowledge and actively search for these patterns, as well as collecting and comparing performance data to diagnose faults. When a fault is identified a common set of solutions are recommended to engineers. The framework is aimed at providing tuning for HPC applications and is not an automated performance regression testing framework.

Lee and Park (2012) proposed a new method for aggregating performance related counters. The method involves normalising the results of related performance counters using the z-value, converting them to t-scores and then calculating an aggregate t-score for each time interval. Changes in the aggregate t-score is then used to indicate a change in application performance.

Heger et al. (2013) propose a method for automated detection of performance regression root causes. Their system first detects performance regressions in unit tests and then, using version control revision information, tries to identify the code changes that caused the regression. A performance regression is considered to be a significant increase in response time of a tested method. Analysis of Variance (ANOVA) is used to compare the response time distribution of two revisions. Once a regression has been detected, analysis of the revision history is used to find the revision that caused the problem. Then call tree analysis is used to identify the root cause of the problem. The proposed system works by benchmarking method execution time which is different from resource utilisation, this method could be a useful extension for the work in this thesis.

Nguyen et al. (2012) proposed an approach to analyse performance counts across test runs using statistical process control techniques. They use the distribution of performance counters to calculate upper and lower control limits for the expected performance of each monitored metric. A measure called the violation ratio is used to identify regressions. The violation ratio is the percentage of tests that fall outside the control limits and is used to express the degree to which the current operation is out of control. When the violation ratio for a test is exceeded it is flagged as abnormal, performance engineers then further investigate the problem.

In Foo et al. (2010) a method to perform regression testing is proposed using pre-computed performance metrics extracted from performance regression testing repositories. Their analysis approach involves normalising metrics, descretizing metrics, deriving performance signatures from a historical dataset and then using these signatures to flag anomalies in a new test run. Finally a report is generated that identifies all the flagged metrics. A measure of severity is used to decide if a behaviour is to be flagged as anomalous. The severity is calculated as the number of flagged instances divided by the total number of instances. The severity is very similar to the violation ratio presented in (Nguyen et al., 2012).

2.2.1 Benchmarking

Benchmarking measures the execution time of important operations, usually across many runs, to detect poor performing areas of code. Benchmarking is easy to implement, as it only requires the subtraction of timestamps collected pre and post test. Software performance regression testing using benchmarking has been implemented in many ways, some of which are discussed below. Benchmarking is

a simple performance assessment method but it neglects other aspects of software performance like resource utilisation.

The use of benchmarking tools is quite common for Java engineers and there exists many different tools to aid in benchmarking an application. Benchmarking works by measuring the time it takes for a test to execute, often running it many times to calculate the average execution time. This allows performance tests to be written for individual tasks that the software is required to perform. Benchmarking has been used to implement performance regression testing.

The Java Application Monitor (JAMon) (Souza, 2011) is a Java Application Programming Interface (API) that allows engineers to monitor production applications. It tracks statistics like benchmarking, marker hit counts and also concurrency statistics. JAMon requires engineers instrument their code with calls to the JAMon API. JAMon does not provide regression testing capabilities.

Perfidix (University of Konstanz, 2012) is a Java benchmarking tool based on work by Kramis et al. (2007). It is a Java benchmarking tool that allows for JUnit like performance test cases to be written. Annotations are used to define set-up and tear-down methods, the methods being benchmarked and the number of times to run a test for benchmarking. The output of a Perfidix test is a report file outlining the execution time for each benchmark. The tool does not provide a means for regression testing to be performed on the data collected. A similar unmaintained tool to Perfidix is JBench (Skeet, 2013).

JUnitPerf (Clarkware Consulting, 2009) is a tool that is similar to Perfidix in that it provides JUnit like benchmarking tests. It provides mechanisms for `TimedTests`, which imposes a maximum elapsed time for the execution of a test and `LoadTests` which are an extension of `TimedTests`, which runs a test with a simulated number of users and iterations. Both types of tests fail when their elapsed time has exceeded the maximum time set for the test. JUnitPerf allows engineers to set benchmarks for code and enforce them using unit tests. It does provide some regression analysis because changes in test results indicate a regression in performance. It is also dependant on engineers choosing good limits for the expected execution time for the tests.

ContiPerf (Databene, 2013) is a Java testing tool that enables engineers to write JUnit 4 unit test cases as performance tests. It uses annotations in a similar way to JUnit 4 that allows for the definition of test execution characteristics and performance requirements. Tests can be marked to run a certain number of times or to be repeatedly executed for a certain amount of time and performance requirements defined for any percentile of execution time. Contiperf does provide a means for enforcing performance requirements through unit testing but does not perform regression analysis or use resource consumption information.

2.2.2 Profiling

Profiling is a form of software analysis that measures metrics important to software performance, like memory use or CPU utilisation. Profiling is a form of software analysis that investigates a program's behaviour as it executes.

There are two kinds of profiling tools: instrumenting profilers (source or binary) and sampling profilers. Instrumenting profilers work by altering an application code or binary, adding calls to functions that will count how many times each procedure was called and records the execution time of each call. Sampling

profilers do not require instrumentation and operate by periodically interrupting a process and inspecting call stacks. Sampling profilers typically have less overhead and are less intrusive than instrumentation based profilers, but are susceptible to inaccuracies if a function executes completely within the sampling interval.

Many tools exist that help engineers obtain profiling data, however the analysis of the data is often performed manually. There are many types of profilers that use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, program sampling, operating system instrumentation, and performance counters. Manually investigating profiling data is a cumbersome task. There exist some tools like JConsole (Oracle Corporation, 2011a), VisualVM (Project Kenai, 2013) and the NetBeans profiler (Net Beans, 2013) which aid in the collection and visualization of data, but do not provide a means to compare data between subsequent test runs.

JPerfAnal (Meyers, 2013) analyses CPU data generated from Java's hprof profiler, generating graphs that break down data in different ways. The main function of JPerfAnal is to allow programmers to detect where and on whose behalf, an application is spending most of its time. It allows method CPU usage to be broken out by caller (10% of Foo.foo()'s time is spent in its call to Bar.bar()), callee (8% of Foo.foo()'s time is spent when it's called from Baz.baz()) and per line usage inclusive and exclusive of calls. The tool provides a good mechanism to view call graph related performance data, but no means to perform resource utilisation regression testing on the data collected.

There are also a range of commercial profilers available for Java. All of these offer similar functionality to previously mentioned profiling tools. Commercial tools include JProfiler (EJ Technologies, 2012), YourKit JProfiler (YourKit, 2013), DevPartner (Borland, 2013), CollectionSpy (Software Pearls, 2009), AppPerfect Java Profiler (AppPerfect, 2013) and InspectIT (NovaTec, 2013). The main selling point of these tools is how the data is collected and in the case of CollectionSpy, which monitors information about the use of objects in the Java Collections framework, a different set of information is collected. None of these tools, to the best of my knowledge, offers a means to perform resource utilisation regression testing.

Calling context profiling is a common technique to analyse the behaviour of programs and to locate hotspots. A common data structure is the Calling Context Tree (CCT) (Ammons et al., 1997), which stores metrics, such as CPU time, separately for each calling context. The collection of profiling data is highly intrusive due the high frequency of method calls and large stack sizes for some applications. There has been a lot of research into low overhead methods of calling context profiling but most of these methods suffer from poor accuracy or poor portability (Moret et al., 2009b). Approximate call trees can be constructed from partial traces (Serrano and Zhuang, 2009) reducing the overhead of the profiler. Other research into CCTs offer compromises between profile accuracy, overhead and portability (Arnold and Grove, 2005; Whaley, 2000; Moret et al., 2009b). There has also been research into the visualisation of CCTs (Moret et al., 2009a) to improve performance engineers' abilities to comprehend large call graphs.

A process known a Differential Profiling has been proposed to identify differences between profiling data collected from performance tests. McKenney (1999) proposes an approach to identify performance bottlenecks in software systems by repeating performance tests with a varying set of workloads so that functions with a sensitivity to the workload could be identified. Schulz and de Supinski (2007) proposed a method for differential profiling that allows two performance profiles

to be subtracted and differences identified, providing an implementation in the form of egprof (Martin Schulz, 2008). Their method uses data from gprof, a profiler which collects call graph data about where a program spends its time and which functions call other functions during programs execution (Graham et al., 2004). It provides a mechanism to subtract call graphs and visualise differences in a programs execution. Differential profiling is very similar to the aims of this thesis but the data used in the proposed methods is different.

Traditionally program profiling is performed off-line, where data from one run of software is used to tune a future run. There has been research into online profiling where data from the current execution is used to dynamically tune aspects of software systems (Duesterwald and Bala, 2000). Profiling data has been used by just-in-time compilers (Burke et al., 1999), dynamic optimizers (Bala et al., 2000; Diamos et al., 2010) and binary translators (Chernoff et al., 1998; Baiocchi et al., 2007) to dynamically improve their performance at runtime.

There are alternative methods of profiling that have been explored. Yan et al. (2012) proposes reference propagation profiling which tracks objects over their lifetime and generates reference propagation graphs to describe a software execution. Container profiling is presented by Xu and Rountev (2008), where instead of tracking arbitrary objects they track containers. A confidence interval, based on the containers memory consumption and staleness (time since last retrieval) is used to identify leaks. Xu (2012) monitors object allocation points to identify data-structures whose lifetimes are disjoint, and whose shapes and data content are always the same. The life-cycle of these objects can then be managed by a cache or pool to improve performance.

2.2.3 Instrumentation

Instrumentation involves adding extra code, known as probes, to an application for monitoring some program behaviour. It can be performed either statically (i.e., at compile time) or dynamically (i.e., at runtime) (Kempf et al., 2007). These probes provide information engineers need to know about the internal functioning of the program. Profilers use data collected from probes to display the collected data in a meaningful way (Woodside et al., 2007).

There already exists many available software performance testing and profiling tools for engineers. There is a variety of mechanisms to collect performance data from executing JVMs. The two most prominent of which are JMX and Java Virtual Machine Tool Interface (JVMTI). However there exist other implementations of Java profilers and there is also a set of commercial profilers available for Java.

The Java platform has many means of instrumentation built into it. JMX (Oracle Corporation, 2013c) is a Java technology that provides a mechanism for managing and monitoring running Java applications. The JMX specification outlines the architecture, a set of design patterns and a set of APIs that allow Java applications to be instrumented and monitored (Perry, 2002). JVMTI (Oracle Corporation) is a native Java API that allows external programs to inspect the state and to control the execution of applications executing on a JVM. JVMTI was introduced in Java and replaces the Java Virtual Machine Profiler Interface (JVMPI). The interface is commonly used for debugging, profiling and monitoring as it provides universal access to JVM internals.

Instrumentation can also be added dynamically. Aspect-Oriented Programming (AOP) can be used to automatically insert instrumentation code into applications with out affecting source code (Debusmann and Geihs, 2003; Biberstein et al., 2005; Seyster et al., 2010). JFluid (Dmitriev, 2004) is an attempt to address the high performance overhead of instrumentation based profilers. They propose a method to dynamically instrument program resources at runtime using dynamic byte code instrumentation and only instrumenting relevant aspects of the program under test. Measurements showed that profiler overhead can be substantially reduced using their technique.

2.2.4 Performance test drivers

Writing and designing of performance tests is an active area of research. Recently, particular interest has been paid to simulating concurrent users accessing websites and simulating server load. Load testing is the most common method for simulating production environment and is used to assess the two Quality of Service (QoS) factors of applications: availability and response time (Draheim et al., 2006).

Load testing is often used when testing the performance of websites. In this scenario it is important to understand how the server reacts when facing different levels of load. Stress testing is a form of load testing that tests the boundaries of what the system can handle before it breaks or becomes unusable. Typical stress includes resource exhaustion, bursts of activities, and sustained high loads (Pan, 1999). Current load testing techniques use benchmarking as their means for determining performance issues and don't take into account resource utilisation of the system.

Another branch of software performance engineering is the development of performance tests and tools that simulate production environments. Most load testers mimic typical user behaviour and range in complexity from simple request generators to complex human behaviour simulators (Draheim et al., 2006; Lutteroth and Weber, 2008; Cai et al., 2007). Testers apply a load to a system by emulating a number of concurrent users, this number can be altered depending of the aims of the performance test. Recent research has focused around automatic detection of performance regressions under load tests (Jiang, 2010; Maalej et al., 2013). Autonomic loading for transactional systems is presented by Ghaith (2013). Their approach explores the workload space searching for points that cause the worst case behaviour for the system. These points are used as configurations for stress tests. Ghaith et al. (2013) also explores the concept of transactional profiles to perform load-independent anomaly detection for load tests. Malik (2013) proposed an automated approach to determine if a system is meeting its service level agreements during load tests.

2.3 Summary

The current state of technology and academic literature has been outlined in this chapter. Methods used in both the early-cycle predictive model-based approach, and a late-cycle measurement-based approach (Woodside et al., 2007) have been discussed. This thesis falls under the umbrella of the measurement-based approach. However, this work may be suitable for validating the predictions of

models generated using the early cycle approach.

Current automated performance regression techniques would not be able to identify the differences in memory utilisation in the motivational example presented in Section 1.3. Benchmarking techniques would not identify any difference in the test's execution time. Differential profiling and call graph analysis may identify some changes in program behaviour, but this is expected when changes to code have been introduced. The changes in resource utilisation will go unnoticed by current performance testing tools. These anomalies are left for performance engineers to manually diagnose. This thesis aims fill this gap in software performance engineering by exploring the concept of resource utilisation regression testing.

Chapter 3

Methodology

The previous section outlined the current state of software performance engineering testing and the need for a tool that can perform resource utilisation regression testing. This section outlines the methodology used to test the concept of resource utilisation regression testing, specifically for JVM based applications. A set of test programs were developed that exhibited common characteristics of Java programs. These programs were made configurable to alter the resource utilisation profile of the test in a way that should be recognised by the classifier. Data for each configuration was generated and used to test the robustness of the analysis components. These tests compared the different analysis components ability to recognise the different configurations and ranked them based on this ability. The results of the tests will show which analysis components perform the best and how well the different configurations can be recognised.

The aim of the project is to detect changes in a performance test resource utilisation. The problem lends itself to the field of machine learning and artificial intelligence. In this case we are interested in detecting changes in time-series data. The analysis algorithm needs to be able to detect when a change in the programs runtime has occurred without flagging changes that have not occurred. This project wants to be able to learn the runtime signature of a performance test and identify when future test runs deviated from this signature.

To achieve the aim of this thesis a method was developed to instrument a program with probes to collect data about the programs resource utilisation. Important properties that are to be monitored during a test's execution needs to be defined. These properties are related to aspects of the program's state, which are relevant to engineers (memory usage, Central-Processing-Unit (CPU) usage, garbage collector activity ....). A method of ensuring the testing environments are homogeneous when classifying a test is required and finally a method of detecting changes in the monitored properties needs to be developed.

Due to the non-deterministic nature of Java programs, tests will need to be run many times to generate accurate models of a program's runtime signature. We expect to see fluctuations between two performance tests, despite being executed in the same test environment. There are many factors that create these fluctuations including; the VM, heap size parameters, JIT compilation and optimizations, thread scheduling, GC scheduling and various system effects. To account for these fluctuations tests will need to run many times so the distribution of a tests resource utilisation can be determined.

As with any monitoring tool the overhead of monitoring the application should

interfere as little as possible with normal execution of the application. This is commonly known as the observer effect where the impact of observing a process alters its behaviour (Röck, 2007). This effect is prevalent in many other sciences including physics and psychology. The mechanism used to collect probe data needs to have low overhead, require few changes in the source of the application and not alter the state of the program or its execution. Overhead of monitoring cannot be avoided, however steps to mitigate its impact on the application need to be taken.

The environment used to run performance tests is important and needs to be considered. The nature of computers these days has become very non-deterministic with many factors influencing how a program executes at runtime. Tests run on different machines with different hardware and software configurations may produce significantly different runtime signatures. When comparing data collected from two performance tests homogeneity of the two test environments used to run the tests needs to be ensured.

The use of mocked data is not enough to test the robustness of the classification methods. The test data needs to contain all the characteristics that we expect to see from an executing application in particular the non-determinism of a programs' execution. To address this a set of test-programs was developed that exercised common aspects of typical JVM based applications and are configurable to produce difference resource utilisation signatures. Test data will be generated using different configurations of each test program by running each test many times and collecting performance counts. The analysis techniques will then be evaluated on how well they identify the changes between each configuration of a test program. The performance of the analysis system will be assessed on the accuracy, precision and MCC.

3.1 Test Programs

A set of test programs will be required for unit testing of the framework as well as for generating data to evaluate the robustness of the classifier. Different applications will be more suited to testing during different stages of development and so a range of applications will be required. Programs will need to contain technical features that exercise all the features of the JVM that we are monitoring. Code for all test programs can be found in Appendix B.

We are monitoring the performance of a JVM, so programs that use native libraries and other process for tasks will no have no effect on the data being collected. Test programs' also need to be simple enough so that changes in the resource utilisation can be linked to the changes in code. Additional libraries can complicate the situation and without in depth knowledge of the libraries it will be difficult to identify the causes of changes identified in the results. Because of this it was decided to use simple test programs that were developed from scratch.

The aim of Buto is to identify regressions in the resource utilisation of Java programs. To test it's ability to do this, changes that alter a programs resource utilisation signature will be introduced to each test program. Changes could include altering buffer sizes, thread counts, data structures used and different algorithms. The changes will be configurable by command line parameters that alter the performance of the test.

Programs used for testing the analysis components will need to possess technical features that alters resource consumption of the JVM. Programs should make use of core Java features like threading, garbage collection, custom class loading, perform I/O and use custom MBeans. A single program need not exhibit all of these characteristics to be considered as a candidate since multiple programs will be used for testing, however total coverage of these characteristics will be required from the set of applications used for testing.

3.1.1 Java reference types

The first test program presented is the soft and strong reference example that was described in Section 1.3. This is a very simple example, where a change in implementation can make a noticeable difference in performance as measured by some metrics. The only real parameter for this experiment is the number of objects that are created. We created 500 objects per second. The variation point for the example is the object manager used, each object manager will have a different impact on the test's memory utilisation.

Another implementation of an object manager is also used, this method uses weak references. Weak references are collected at the discretion of the garbage collector and there is no guarantee about collection. Listing B.1 outlines the code for the weak object manager implementation, it is very similar to the soft object manager presented in Listing 1.4.

The variation points for the Java reference type test program are as follows.

Strong reference list uses a list of standard Java references. References are cleared from the list when it's size reaches `100,000` objects.

Weak reference list uses a list of `java.lang.ref.WeakReference`. References are cleared from the list at any time, depending on the garbage collector algorithm.

Soft reference list uses a `java.lang.ref.SoftReference`. References are cleared at the discretion of the garbage collector in response to memory demand.

Table 3.1 outlines the experiments conducted for the Java reference types test program. Six experiments will be conducted, two for each variant comparison. Pairs of variants are compared from both perspectives, using one for training and the other for negative data and vice versa.

Table 3.1: Java reference type experiments based on variation points. The columns indicate the training variant and rows the negative variant. Each of the + indicates an experiment.

	strong	weak	soft
strong		+	+
weak	+		+
soft	+	+	

The variants have a direct effect on the utilisation of heap memory and the way the garbage collector interacts with the objects being held by the managers. We predict the following properties will be affected by the following variations.

- Heap memory used / committed
- Eden space memory used / committed
- Old generation memory used / committed
- Scavenge GC collection count / time
- Marksweep GC collection count / time

3.1.2 Object leaker

Identifying memory leaks is a core use case for a tool that identifies regressions in resource utilisation. Memory leaks are a common problem for performance engineers and the object leaker provides a simple example of a memory leak. At defined intervals the program creates a set of new objects and stores references to them to prevent them from being garbage collected. Command line parameters are used to define the rate of leaking, the type of object to leak and the execution time for the test.

Listing B.2 shows pseudo code for the leaker test program. A list is used to keep references to leaked objects. The test's execution time is controlled using a thread (Appendix B.5) which sleeps for the duration of the test and shuts down the VM when woken. The program enters an infinite loop waiting for the shut-down thread thread to wake and close the VM. Each execution of the loop generates a set of objects and stores a reference to them before sleeping for the specified leak interval.

The variation points for the object leaker test program are as follows.

String Generates a large string using string concatenation and keeps reference to final string object.

String builder Generates a large string a string builder (buffer) and keeps reference to final string object.

Thread Generates a thread object which prints to the standard output stream.

Table 3.2 shows the experiments conducted for the object leaker test program. There are three variants so six experiments will be conducted, two for each variant comparison.

Table 3.2: Object leaker experiments. The columns indicate the training variant and rows the negative variant. Each of the + indicates an experiment.

	string	string builder	thread
string		+	+
string builder	+		+
thread	+	+	

The object leaker has a direct effect on the program's memory consumption and the activity of the garbage collectors. The following properties are predicted to be affected by the variation points.

- Heap memory used / committed
- Eden space memory used / committed
- Old generation memory used / committed
- Scavenge GC collection count / time
- Marksweep GC collection count / time
- Thread count

3.1.3 Classloading

Memory leaks are a common problem for Java programmers. Usually they are caused by a collection somewhere with references to objects that should have been cleared, but never were. The leaking of class loaders and classes are a special case of a memory leak, which occurs often in applications that require classes to be reloaded at runtime. A common use case of classloaders is to reload servlets in web servers like Jetty or Tomcat. Poor programming can lead to non-heap memory leaks if the class resources are not properly disposed of leading to `OutOfMemoryError` after a few redeploys of a web-application.

In the Java programming language, every object has a reference to it's associated class object, which in turn has a reference to the classloader that loaded it. Also each classloader has a reference to each of the classes it has loaded, which holds static fields defined within the class. This means, if a classloader is leaked so will all associated classes and their static fields. Static fields commonly hold caches, singleton objects and application configuration objects. This demonstrates how the leaking of classloaders can be expensive and cause `OutOfMemoryError`'s to be thrown. For a classloader to be leaked it is enough to have a reference to any object created from any class loaded by that classloader. The object will reference its class object, which in turn references the classloader that loaded it. This means a single class that survives a redeploy, and doesn't perform a proper clean up of resources, can sprout a memory leak.

The example consists of a runner class, an interface and implementation for a typical application domain object, and a factory to instantiate instances of the domain object. Listing B.3 is the main class to the classloading test program. It contains a single static field referencing an interface for a domain object. The field is initialized by creating a new domain object from the the domain object factory. Each iteration of the loop simulates an application redeploy where a new domain object is generated from the factory and the the old object state is copied to the new object.

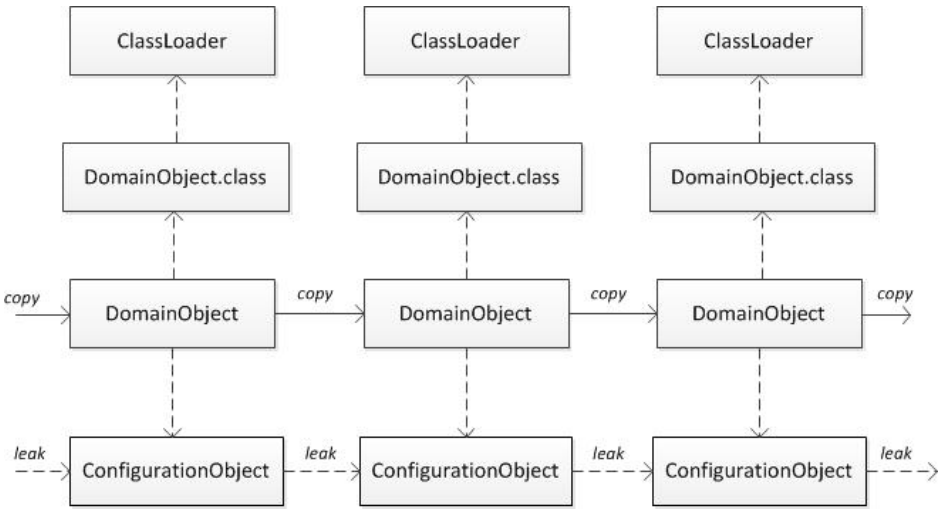
Listing B.5 shows the implementation of the domain object factory. It contains a name property and a configuration object. The copy method copies over the name of the provided domain object and also the configuration object. The getter for the configuration property creates a new configuration object using the held configuration object as it's parent.

Listing B.6 outlines the factory that generates new domain object. A URL classloader is used to load class definitions for the domain object, the loading of all other classes is deferred to the parent classloader. A new classloader is used each time a new instance is generated, so if the generated domain object is leaked so will the classloader used and the object's class definition.

Listing B.7 is the object responsible for the leak of domain objects. The configuration object simulates a property of the domain object. Like a set of parameters to tune a cache or database connection. This reference is responsible for the leaking of domain objects and their associated classes and classloaders. Each time configuration object is cloned a reference to the parent configuration is kept in the new object. This reference creates the leak.

Figure 3.1 illustrates how the memory leak occurs. Each time a new domain object is created and the properties of the old domain object are copied to the new object. This involves creating a copy of the configuration object. Each copy of the configuration object keeps a reference to the parent object, this is what creates the leak. The configuration object acts like a daisy chain between the new domain objects, ensuring objects are not garbage collected. The creation of each new domain object uses a new classloader. This causes new instances of the domain object class to be loaded and also leaked when an instance of that class is leaked. The bug in this example is obvious, however this example is over simplified and bugs similar to this can easily occur.

Figure 3.1: Classloading dependency diagram. Each time a leak occurs a new domain object is created by copying the old one, a configuration object is copied which contains a link to the parent configuration which causes the leak.



For the classloading test program there is only one parameter to configure, that is the leak interval. Changing the leak interval will affect the utilisation of non-heap memory, compilation time and the class loaded count. Increasing the leak interval will increase the rate at which non-heap memory is consumed, increase utilisation of the perm-gen memory pool, increase compilation time and increase the class loaded count. Decreasing the leak interval will have the opposite affect. Non-heap memory will be utilised less, with less compilation time required and fewer classes loaded.

Three variation points using different leak rates were used. Different leak rates should alter the consumption rate of non-heap resources and load different amounts of classes. The following variants of leak interval were chosen.

- 10ms
- 100ms
- 1000ms

The experiments conducted based on the variation points are presented in Table 3.3. Each variation point is compared against each other variant, from both perspectives (a used for training and b for testing and vica versa).

Table 3.3: Class loading experiments. The columns indicate the training variant and rows the negative variant. Each of the + indicates an experiment.

	10ms	100ms	1000ms
10ms		+	+
100ms	+		+
1000ms	+	+	

The classloading test program has a direct effect on the program's non-heap memory consumption and classloading mechanisms. The following properties are predicted to be affected by the variation points.

- Heap memory used / committed
- Non-heap memory used / committed
- Eden space memory used / committed
- Old generation memory used / committed
- Permanent generation memory used / committed
- Scavenge GC collection count / time
- Marksweep GC collection count / time

3.1.4 Consumer Producer

The consumer producer problem is a classic example of a multi-process synchronization problem (Donlin, 2002). It consists of producer processes which generate data items for consumer processes to consume. Data items are passed between processes using a shared buffer, which will block producers if the queue is full and block consumers if the queue is empty. The example illustrates a classic concurrency problem that should have an interesting effect on the resource utilisation of the test. Custom instrumentation has been used in this example. Two custom properties are monitored; the buffer-usage and the processed job count (throughout).

The program simulates a small website that displays student information. Producers generate lists of 100 students and consumers generate a HTML page based on a list of students. The program can be configured by three parameters; the number of producers to use, the number of consumers, and the size of the shared-buffer. Altering these parameters should affect the program's resource utilisation signature.

Custom instrumentation is used in this example to monitor the buffer-usage and the processed job count. Notifications are also emitted by the program. When

ever the processed job count passes a log interval a notification is emitted from the program to inform listeners. Instrumentation was performed following the JMX API and can be found in Appendix B.4.

The main class for the program is presented in Listing B.8. The parameters for the test are read off the command line and the example initiated. First the runner is registered with an MBean sever so it can be accessed by the data collector. Then the shared-buffer is created, and the set of producers and consumers. The method `getProductionMBeanInfo` is used by the data collector to retrieve the data for the custom properties . The example is further explained in Appendix B.4.

The consumer process is responsible for taking data items from the shared buffer and processing them. In this example the consumer takes lists of students and generates a HTML page based on the list. Listing B.9 presents the code for the consumer class. The consumer is passed a shared-buffer and a shared-counter during construction. The buffer supplies the consumer with data-items, and the counter is used to monitor the consumption rate of the program.

The producer process is responsible for generating data items for the consumers. In this scenario, lists of students are data items. Students are generated by randomly selecting first names, last names and degrees from lists. Each data item consists of 100 students. Listing B.10 presents the implementation for the producer object.

Experiments for the consumer producer test program vary from the other test programs. In this case a variant is used as the benchmark and all other variants are compared with the benchmark. The benchmark variant uses 10 producers, 10 consumers and a buffer size of 10. The other variants consists of combinations of high and low counts for the producer count, consumer count and buffer-size. Table 3.4 shows the seven different variations of the configuration parameters.

Table 3.4: Consumer producer variation points. V0 is the baseline variant which will be compared against the other configurations. Other configurations are variants of high (1000) and low (10) counts for each parameter.

variant	producers	consumers	buffer-size
v0	10	10	10
v1	1000	10	10
v2	10	1000	10
v3	10	10	1000
v4	1000	1000	10
v5	10	1000	1000
v6	1000	10	1000
v7	1000	1000	1000

Table 3.5 outlines the the experiments performed. The benchmark variant is `v0`, it is compared against each of the other variants indicated by the + column and each variant is compared against the benchmark indicated by the + row. A total of 14 experiments will be performed, two for each variant comparison.

It is expected that the producer count will increase the rate at which data items are produced and the rate at which the buffer fills up. When the buffer becomes full the producer threads will have to block until buffer space is freed by the consumers. Increasing the consumer count will increase the rate at which data items are consumed from the buffer and the rate at which the buffer empties. When the

Table 3.5: Consumer producer test program experiments. The columns indicate the training variant and rows the negative variant. Each of the + indicates an experiment.

	v0	v1	v2	v3	v4	v5	v6	v7
v0		+	+	+	+	+	+	+
v1	+							
v2	+							
v3	+							
v4	+							
v5	+							
v6	+							
v7	+							

buffer becomes empty the consumer threads will have to block and wait for new data items to be produced. Increasing the buffer size will allow more data items to be stored in memory and, depending on the rate of production and consumption affect how full the buffer gets. However, increasing the thread count will increase the overhead of threading which may lower the throughput of the program.

The program exhibits a common problem for software engineers and will have an effect on a large amount of resource utilisation properties. We predict the following properties will be affected by the variation points.

- Heap memory used / committed
- Eden space memory used / committed
- Thread count
- Old generation memory used / committed
- Scavenge GC collection count / time
- Marksweep GC collection count / time
- Buffer size
- Processed job count

3.2 Monitored Data

Each test run has a set of properties that are monitored during execution. These properties can be broken down into two groups; dynamic-properties and static-properties. Dynamic properties have values that are expected to change during execution and need to be continuously collected. Static properties are not expected to change during execution and only need to be collected once. These properties can be considered meta-data about a test run. JMX provides a standard set of MBeans which provide statistics on the performance of the VM, these are known as platform MBeans. The standard of dynamic properties monitored by Buto are collected from platform MBeans. Custom properties can also be defined. The consumer producer test program monitors the buffer-usage and processed job count of the example. Notifications are data items that are pushed from the test-program to the data collector to indicate significant events.

3.2.1 Dynamic Properties

Dynamic properties are the properties whose value is expected to change and is monitored over the duration of the test. These properties represent an aspect of the application being run that is expected to change over the duration of the test. During classification, each dynamic-property is treated individually and a model for each of the dynamic-properties is generated. Common examples of dynamic properties are the throughput of a buffer, number of threads used in a thread pool or time spent waiting on I/O. The set of standard dynamic-properties monitored by Buto is presented in Table 3.6. An example of a custom MBean with dynamic properties can be found in the consumer producer test program.

3.2.2 Meta-data Properties

Meta-data properties are properties whose value is static and does not change for the duration of the test. These properties are used as meta-data for a test, describing information about the software configuration or the hardware the software is running on. Meta-data is used to ensure training-runs used to build a classifier and test against a trained classifier all have a similar execution environment. Common meta-data properties are size of thread pools, buffer sizes and other software configuration options. The set of default meta-data properties monitored by Buto is presented in Table 3.7. An example of a custom MBean with static properties can be found in the consumer producer test program.

3.2.3 Notifications

Notifications are a mechanism to push data from an executing JVM to clients using the JMX API. This tool provides a mechanism to listen and collect different types of notifications emitted during a test. Notifications are commonly used to signal state change, a detected event or a problem. When building a classifier and testing against a classifier, notifications from corresponding runs are compared to each other by grouping them by type, source object and sequence number and then testing the time-stamp is within a certain range.

Type The type of notification being emitted.

Source Object The name of the object that emitted the notification.

Message The message of the notification.

Sequence number The number of notifications that have been emitted by the source object.

Time-stamp The time since the JVM's invocation when the notification was emitted.

The consumer producer test program illustrates how notifications work. Listing B.14 (Appendix B.4) defines the counter object which is an object shared amongst consumers that tracks the number of data items consumed. The `increase()` method is used to increase the counter. When the counter passes a log interval a notification is emitted by the counter. If the data collector has subscribed to notifications from the MBean the notification will be stored and saved with the training run.

Table 3.6: Table of default dynamic-properties that can be monitored during a test execution. These properties are made available through the platform MXBeans distributed with the JRE SE7.

Parameter	Description
Total loaded class count	The total number of classes that have been loaded since the JVM has started execution
Loaded class count	The number of classes that are currently loaded in the JVM
Unloaded class count	The total number of classes unloaded since the JVM has started execution
Total compilation time	Approximate accumulated elapsed time (in milliseconds) spent in compilation
Object pending finalization count	The approximate number of objects for which finalization is pending
Heap memory committed	The amount of memory (in bytes) that is available for use by the JVM for Heap Memory
Heap memory used	The amount of heap-memory-used in bytes
Non-heap memory committed	The amount of memory (in bytes) that is available for use by the JVM for Non-Heap Memory
Non-heap memory used	The amount of Non-Heap memory used (in bytes)
Daemon thread count	The number of live daemon threads
Thread count	The number of live threads
Total started thread count	The total number of threads started
Buffer pool *(for each buffer pool)*	
Count	Number of buffers in the pool
Memory used	Memory for this pool
Total capacity	Total capacity of the buffers in this pool
Garbage collector *(for each garbage collector)*	
GC collection count	Total number of Garbage Collections so far
GC collection time	The approximate accumulated time spent doing garbage collection
Memory pool *(for each Memory pool)*	
Pool memory committed	The amount of memory in bytes that is available for this memory pool
Pool memory used	The amount of memory used by this pool, in bytes
Memory threshold	The usage threshold value of this memory pool
Memory usage threshold count	The number of times that the memory usage has crossed the usage threshold

Table 3.7: Meta-data that can be collected from the platform MXBeans distributed with the Java SE 7.

Parameter	Description
System load average	The system load average for the last minute
Boot class path	The boot class path that is used by the bootstrap class loader to search for class files
Classpath	The Java class path that is used by the system class loader to search for class files
Input arguments	The input arguments passed to the JVM which does not include the arguments to the main method
Library path	The Java library path
Management spec version	The version of the specification for the management Interface implemented by the running JVM
Spec name	The JVM specification name
Spec vendor	The JVM specification vendor
Spec version	The JVM specification version
Start time	The start time of the JVM in milliseconds
System properties	Key value pairs for all system properties
Uptime	The uptime of the JVM in milliseconds
VM name	The JVM implementation name
VM vendor	The JVM implementation vendor
VM version	The JVM implementation version
JIT name	The name of the Just-in-time (JIT) compiler
Heap initial size	The amount of memory in bytes that the Heap memory area initially requests for memory management
Non-heap initial size	The amount of memory in bytes that the Non-Heap memory area initially requests for memory management
Memory pool *(for each memory pool)*	
Memory managers name	The name of the manager of this memory pool
Name	The name representing this memory pool
Type	The type of this memory pool

3.2.4 Custom Properties

This tool takes advantage of the instrumentation method made available by the JMX API. The fundamental concept of the API is managed beans or MBeans. An MBean represents a resource that needs to managed or monitored. Custom MBeans can be defined by following the JMX API and registering objects with the JVM's MBean server at runtime. Once MBeans are instrumented and registered the data collector needs to know the fully qualified object name, list of dynamic properties to monitor, list of meta-data properties to collect and whether to monitor notifications emitted by the MBean.

The Consumer Producer example contains a custom MBean that is monitored during tests. It contains a dynamic-property queue-size which is the current number of items in the buffer, a static property buffer-size which is the number of slots available in the buffer and dynamic-property processed job count which is the total number of jobs processed by consumers. Notifications are also emitted by the counter object when processed job counts reach passes a set interval.

3.3 Analysis

The main aim of this thesis is to detect changes in application performance which involves comparing and classifying data collected from performance tests. There are three types of properties monitored; dynamic, meta-data and notifications. A method for identifying changes for each of these property types was developed. The meta-data and notification classifiers are relatively simple only consisting of a single processing step. The dynamic-property classifier is more complex, with a pipe-line of processing steps.

Both the meta-data and notification classifiers consist of a single classification step. Raw data is extracted from the training-set and used to build a model. Raw test data is then extracted from test-runs and compared directly against the model. Implementation of the meta-data classifier can be found in Section 4.5 and for the notification classifier in Section 4.6.

The dynamic-property analysis pipe-line consists of three stages, pre-processing, model creation and testing against the model. Pre-processing is concerned with converting data from the collection format and transforming it so it is ready to be used by the classification algorithms. Two-preprocessing steps are performed; smoothing and time-point generation. The classification algorithm then generates a model using the provided training data set. The model consists of a set of upper and lower bounds that represent the resource utilisation signature of the training-set. Pre-processing is applied to test-data before comparison against a model. A bounds matcher is used to associate test-data points with bounds so they can be tested for validity.

Smoothing is used to remove noise from the data to improve the robustness of the classifier. Smoothing can be configured by two parameters; smoothing type and the smoothing window-size. There are two main smoothing methods, median and mean smoothing. No smoothing is also considered. The window-size parameter controls how many data points are used to calculate the average. Increasing the smoothing size will remove more noise from the data but also remove information that can be used by the classifier. A tuning test will be required to determine the optimal window-size and smoothing method.

A test-run consists of a series of snapshots of application state. For a training-set to be used by a classifier, the individual training-runs need to be merged into a single data structure that can be handled uniformly by the classification algorithms. The concept of a time-point is used to address this requirement. A time-point represents the distribution of data points at a given point in time. Time-point generators convert smoothed training data into a list of time-points. Two methods for time-point generation are explored; rounding and linear.

Models are created by classification algorithms. Three methods of generating models were explored by this thesis; simple method, Kalman filter and the particle filter. These methods operate on a list of time-points that represent the distribution of training-set data. The model consists of a series of upper and lower limits that define the limits for resource utilisation at that point in time. Each classification method interprets the time-point data differently and generates different models.

Once a model is generated test-runs need to be compared against the model. First, test-run data is pre-processed using the same method as the training data that was used to generate the model. Then the processed data is compared against the model. This involves matching each data point to a bounds and testing if it falls within the limits. A bounds matcher is used to retrieve bounds objects from the model given a time-stamp. Two methods of matching bounds are explored; nearest bound and interpolation.

3.4 Experimental Design

This section presents the design of the experiments to test the concept of resource utilisation regression testing. A set of test programs have been defined to test this concept. Each test program has a set of tunable parameters which affect it's runtime performance. From these parameters a set of configuration variations and experiments have been defined. For each variant datasets of 100 test runs were generated and stored for use in the experiments.

Testing of the meta-data classifier is not presented in this thesis. The method used is not a novel technique and involves comparing primitive data types. Unit-testing was performed to ensure the classifier met specification and no further testing is required. The meta-data classifier was used in testing to validate the data used in the experiments.

Across all test-programs there are 32 experiments. Six for the Java references, object leaker and classloading test-programs. The consumer producer is the most complex with 14 experiments. All experiments will be performed for each of the tuning tests. The first tuning test determines the training set size and therefore the test set size (100 trainings for each variant). Once this is determined the number of positive and negative test instances can be determined.

First a set of tuning tests for the classifier need to be performed. This consists of tests to determine the optimal training-set size, pre-processing techniques, bounds matching, and individual classifier tuning.

To determine the optimal training-set size, experiments were performed with a range of training-set sizes. Increasing the training-set should improve the accuracy of the classifier because it has more knowledge to use when generating bounds. However, increasing the training-set size will increase the computation cost, a point of diminishing returns needs to be identified to balance these two requirements. Datasets are of size 100 and 20 will need to be reserved for validation of the

model, so the test training-set sizes will be `[10,20,30,40,50,60,70,80]`.

There are two pre-processing techniques that need to be tuned; smoothing and time-point generation. Smoothing needs to determine the optimal smoothing type (mean, median) and the smoothing window size. The number of data points in the training run is roughly 50, so only smoothing size testing was limited to roughly a third of that to prevent over smoothing the data set. The range of window-sizes was chosen as the set of powers of two less than 20 `[2,4,8,16]`. Smoothers only operate using odd smoothing sizes to one was added to each value to convert it resulting in `[3,5,8,17]`. There are two time-point generation methods that need to tested; linear and rounding.

The final tuning test involves identifying the best parameter set for each classifier. Three classifiers were developed, the simple classifier, Kalman classifier and the particle classifier. Each classifier has a deviations parameter which is not tuned in these experiments but is used in the overall experiments. The simple classifier has one parameter to tune; the average type. The Kalman classifier has two parameters; the average type and the process noise method. The particle classifier has five parameters; the number of iterations, re-sample percentage, propagate method, weighting method and the particle count

Once the optimal configurations for training-set size, preprocessing, bounds matching and each classifier has been determined, the overall experiment is conducted. This involves running each classifier against each other with a range of deviations to determine the best method. A range of deviations was used for each classifier, the range was determined from experimentation. Most classifiers performance dropped off after 20 deviations. So the set of deviations was chosen as values between 0 and 20. In the interest of reducing running time even values were chosen. The set of deviations used is `[2,4,6,8,10,12,14,16,18,20]`.

Notification classifier tests are performed during the overall experiment. The results are presented separately. The consumer producer test program is the only test program that emits notifications. Data from this test-program is used to evaluate the method used to classify notifications. Results from classifying similar and different tests are explored in the results.

Evaluation is performed by assessing how well a configuration performs on both positive and negative data. This example is a binary classification whose results can be broken down into four categories.

True Positive (TP) A positive instance that is correctly classified as positive.

False Negative (FN) A positive instance that is incorrectly classified as negative.

True Negative (TN) A negative instance that is correctly classified as negative.

False Positive (FP) A negative instance that is incorrectly classified as positive.

For each analysis configuration the number of true-positives, false-negatives, true-negatives and false-negatives are counted. The results are aggregated for each dynamic-property, across each experiment, for each test-program and for overall results. For the training-set tuning test overall results are used to rank the configurations. Using these measures three metrics to rank performance are used; accuracy, precision and MCC.

Accuracy - *Accuracy is how close a measured value is to the actual (true) value. In binary classification the accuracy is the proportion of true results in the*

population. (Sokolova et al., 2006).

$$accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{3.1}$$

Precision - *Precision is how close the measured values are to each other. In binary classification it is the proportion of true positives against all the positive results.* (Sokolova et al., 2006).

$$precision = \frac{TP}{TP + FP} \tag{3.2}$$

Mathews Correlation Coefficient (MCC) - *is commonly used in machine learning as a measure of the quality of binary classifiers, it is a balanced measure which can be used even if the classes are of different sizes. It returns a value between - 1 and +1. +1 meaning perfect prediction, 0 no better than random and -1 indicates total disagreement between datasets.* (Vasanthanathan et al., 2009).

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{3.3}$$

Results are ranked by the MCC rating. The accuracy and precision metrics are useful for drawing conclusions about the classifiers performance on the data set.

Chapter 4

Design

In the previous chapter the methodology used to develop and test the concept of resource utilisation regression testing was outlined. Test programs were described, along with monitored properties, analysis techniques and experimental design. This chapter presents the design of Buto, a framework to test the concept of resource utilisation regression testing. The design consists of seven components; instrumentation, data collection, training manager, data storage, classifiers, graphing and reporting.

Figure 4.1 outlines the design for Buto. Instrumentation is used by the data collector to collect data about executing performance tests. The training manager schedules and monitors performance tests. Collected data is stored for later use by the classifier, which consists of a property classifier, meta-data classifier and notification classifier. The output from the classifier is used by the graph generator and report generator to create output for engineers to inspect.

Java was chosen as the implementation language for the framework, however there is no reason why this could not be implemented for other languages. Java is a commonly used language, second behind C according to TIOBE (TIOBE Software, 2013) and contains many libraries for performance monitoring. Instrumentation is provided by the JMX API which exposes a number of resource utilisation metrics related to the JVM's performance (Oracle Corporation, 2013d). Custom instrumentation was implemented by providing a means to access custom MBeans through the JMX API.

4.1 Instrumentation

Instrumentation of performance tests is required to collect performance data. For Buto, instrumentation was implemented using the JMX API (Oracle Corporation, 2013d). JMX exposes a number of JVM performance counters and a provides a mechanism for custom instrumentation. There are three types of data collected; dynamic-data, meta-data, and emitted data. There is no difference between instrumenting static and dynamic data, however emitted data needs to follow the JMX specification for notifications and the Buto API.

The main concept of the JMX API is the Managed Bean (MBean), which is a managed object that represents a resource that needs to be managed. MBeans expose an interface consisting of a set of readable and writeable attributes, invokable operations and MBean meta-data. MBeans must follow the design patterns

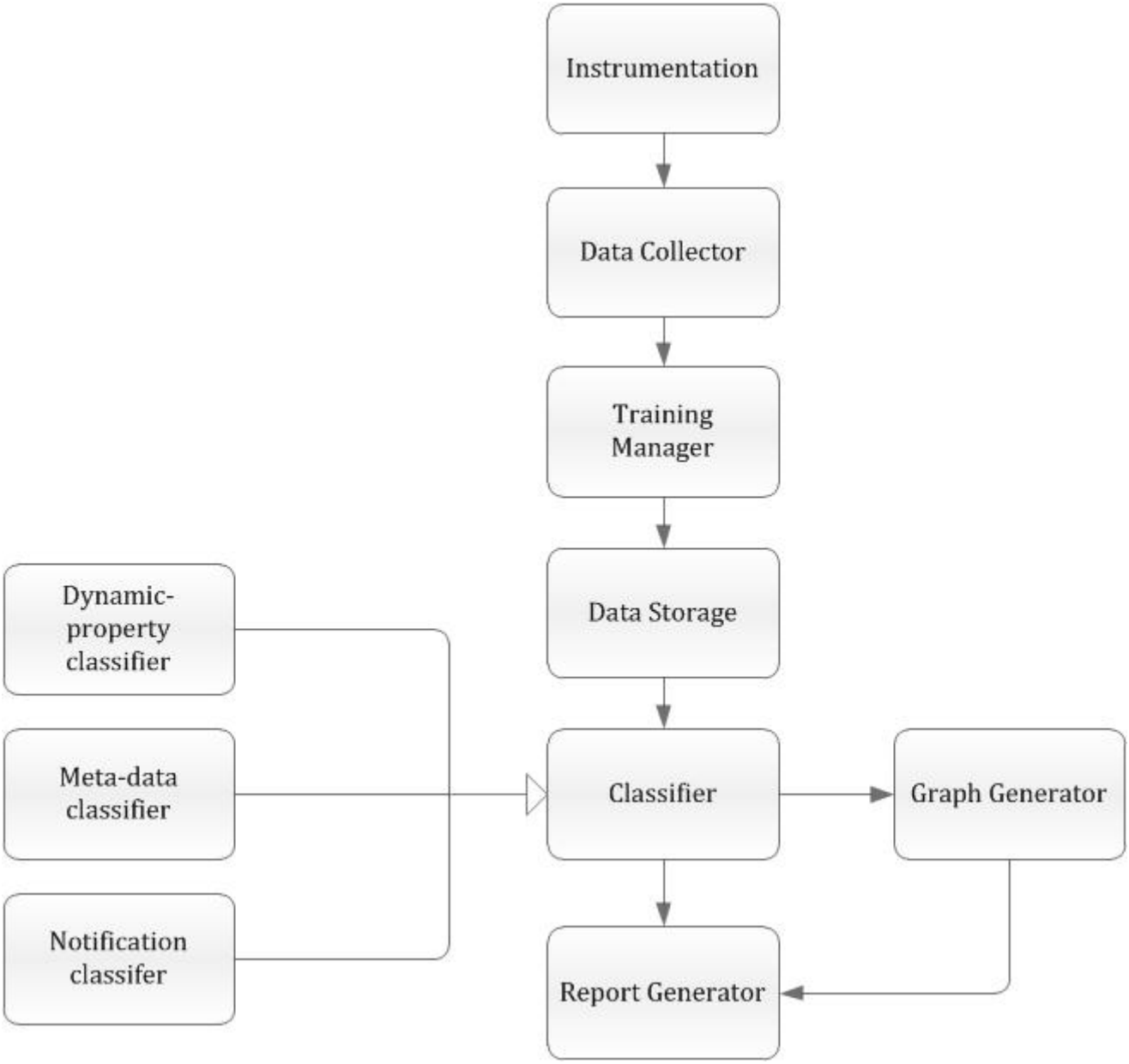


Figure 4.1: Design overview. Each shape represents a component of the framework and the arrows represent the flow of data. The main flow starts with instrumentation and finishes with the report generator. The classifier consists of three sub-classifiers, one for each of the property types.

and interfaces defined in the JMX specification. MBeans can also push data to listeners using the notifications API. MXBeans are a sub-type of MBeans that only reference a pre-defined set of data types. Once a resource has been instrumented by MBeans, it can be managed through a JMX agent.

Platform MXBeans are a set of MXBeans that are provided with the Java SE platform. They provide a monitoring and managing interface for resources inside JVM and other components of the Java Runtime Environment (JRE) (Oracle Corporation, 2013e). These beans contain information about class loading, buffer pools, garbage collector, memory usage, operating system, runtime and thread data (Oracle Corporation, 2013b). By default the data in these beans are collected, however the data collector can be configured to monitor different sets of properties.

Standard JMX monitoring requires that clients request data from the MBean server. Notifications are a mechanism for managed objects to push data to listening clients. There are no standard notifications emitted by the JMX API, only custom MBeans emit notifications. The consumer producer test program (Section 3.1.4) emits notifications when the processed job reach count passes log intervals. Notifications consist of a type, source object, sequence number, time-stamp and a message.

To monitor executing performance tests a JMX client was implemented in the form of the data collector. The training manager is responsible for setting up the performance test and getting a JMX connection to the test. A data collector is then attached to the connection and begins monitoring the state of the VM.

4.2 Data Collection

This section outlines the design of the data collection mechanism of Buto. The data collector is responsible for monitoring an executing performance test. When constructed it is given a connection to the target VM and a property set to monitor. Three types of properties are collected; dynamic-properties, static-properties (meta-data) and notification emissions. Once connected to a performance test, the data collector collects the meta-data, sets up notification monitoring and begins polling for dynamic-properties.

The test life-cycle is outlined in Figure 4.2. Once a connection has been established with the process a data collector is attached and it begins monitoring the test. Figure 4.3 outlines the process of polling a performance test. Standard output, which is normally direct to the console, is sent to a log file and added to the training run after the test has terminated. First the meta-data is collected. Next, notification subscriptions are set-up. There are no notifications emitted by the platform MXBeans, notifications are only emitted by custom MBeans. Once notification subscriptions have been set-up, notifications are asynchronously emitted by the VM and processed by the data collector.

The data collector then begins the collection of dynamic-properties. It enters a loop of data collection events until the test terminates. Throughout this cycle standard output is sent to the log file and notifications are processed asynchronously by the data collector. In each cycle a snapshot is collected by querying all monitored MBeans for their dynamic-properties. A snapshot is used to represent the state of the process at a given point in time and is a collection of all the dynamic-properties. The data collector then sleeps until the next collection event. The

interval between the collection events can be configured to optimize performance. If the connection times out then the test program has terminated and the test is over. Once the test has terminated the training manager will process the standard output log file and add it to the training run as meta-data.

Two parameters are used to configure the data collector; the interval between collection events and the properties being monitored. Changing the interval between collection events alters the granularity of the data collected and can help manage the size of the resulting data. For long performance tests (greater than 12 hours) collecting data every `1000ms` is impractical and will provide a lot of redundant data, a `60000ms` interval might prove more useful. Changing the monitored property set alters the amount of data added to each snapshot. Some performance tests may not perform class loading or be single threaded, so monitoring these metrics would not prove useful. It is a good idea to reduce the number of data types collected for a performance test to reduce the overhead of the collector and reduce the complexity of the later analysis. Altering both parameters will have an effect on the interference made by the collector on performance test's execution.

The overhead of the tool is an important consideration. Altering the collection time interval or the monitored properties can have an impact of the overhead of the tool. A balance needs to be struck between overhead and the amount of data collected, this needs to be assessed by engineers on a case by case basis. The overhead of the tools is recorded and added to snapshots. For each snapshot and each MBean monitored, the time spent collecting data is recorded and added to the snapshot. This allows the overhead of the tool to be treated like other dynamic-properties and can be used in analysis in the same way. Monitoring and analysing the overhead can make the overhead of monitoring known to the engineers and also identify slow-downs in the collection of Custom MBean data.

Configuration of the data collector is an important part of testing and a poor choice of configuration may have a negative impact on the performance of the classifier. A balance needs to be struck between the overhead of the collector and the amount of data collected about the performance test.

4.2.1 Data Storage

The data collected during test-runs needs to be stored for later analysis. A test-run consists of a series of snapshots, a map of meta-data values and a list of notifications. A snapshot is a map of dynamic-properties and their measured value, associated with a time-stamp. Text files were used to store the data for the testing of this tool, but the API allows for other data sources to be implemented.

Analysis can be time consuming, depending on the length of the performance test, number of training runs used and the number of dynamic properties being analysed. The model generated by the classifier is a list of bounds (upper and lower limit) which can be persisted and reloaded in text files.

The output of analysis is a directory structure contain text files and images of the results of classification. For each dynamic-property in a classification, the data used in each stage of the classification pipeline is outputted to graphs and text files. A result file for the classification is outputted containing all the instances anomalies detected for dynamic-properties, meta-data and notifications. The output for the experiments can be repeated following the guide in Appendix A.

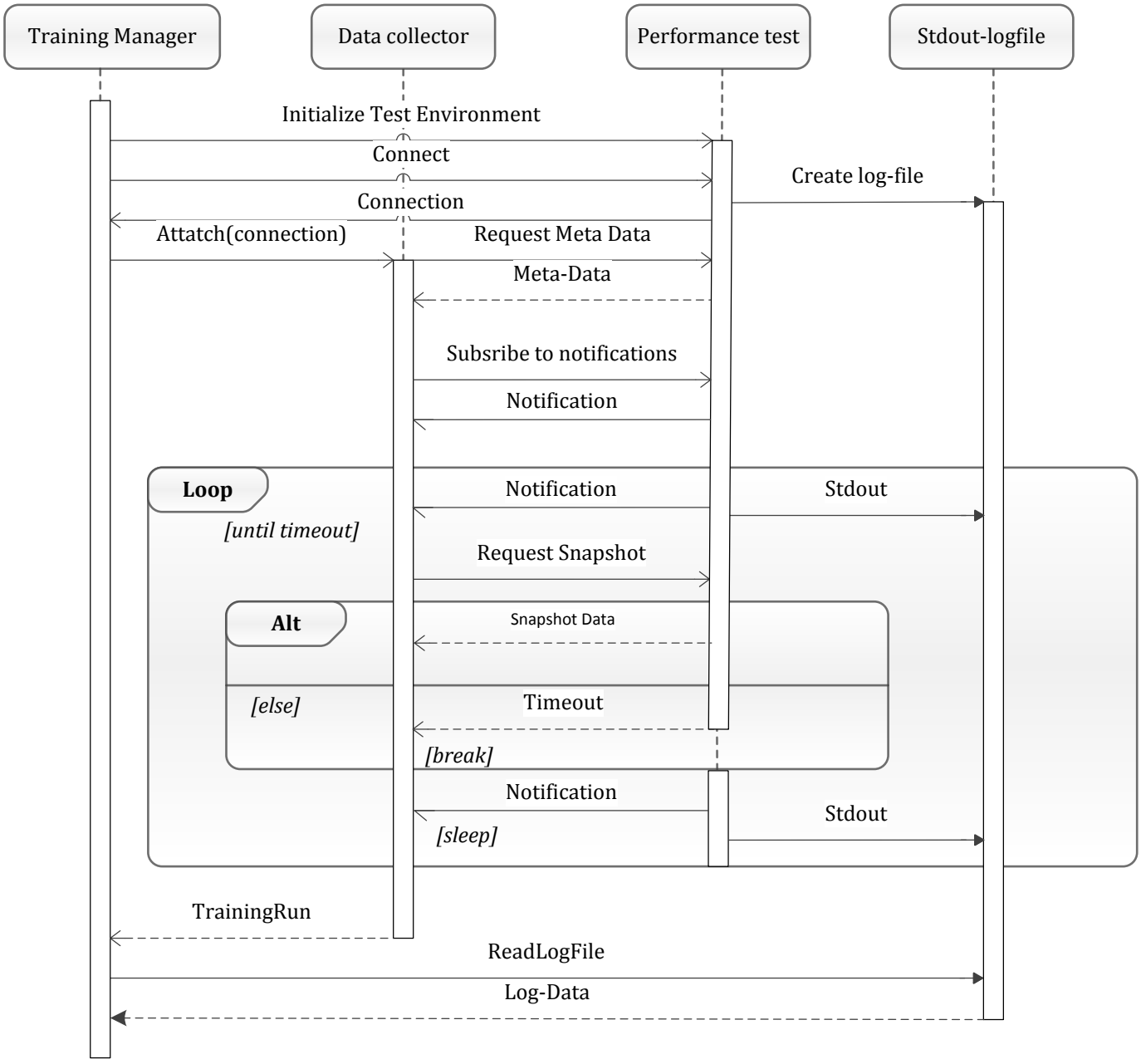


Figure 4.2: Training sequence diagram. UML 2.0 sequence diagram. Shows the interaction between the training manager, data collector, and the performance test.

4.3 Training Manager

The training manager is responsible for managing the test environment and running batches of performance tests. It manages the testing environment by initiating a performance, attaching a data collector to the process, monitors the VM state during execution and ensures the VM exits correctly. The training manager generates training sets by running performance tests many times, collecting data from each execution and storing as a training-set.

The test life-cycle is outlined in Figure 4.2. First, the test needs to be initialised and connection made to it using the JMX API. The JVM is initialised using the provided class-path and VM options for the variant being tested. Figure 4.3 shows the method used to connect to the test program. The data collector first initializes a new JVM using the provided class-path, VM options, and main class. The training immediately requests a connection to the performance test. If a connection cannot be made, the training manager then sleeps for 100 ms and tries again. This process is repeated until either a connection has been established or the maximum number of connection attempts has been meet. In the case of the latter it is assumed the VM has terminated unexpectedly.

The standard output from the performance test is redirected to a log file. Once the test has terminated the log file is read and added to the meta-data for the test. The console output is often a valuable resource when debugging an application and it makes sense to monitor it. However, console output is often highly varied especially in multi-threaded systems and is not suitable to be used in classification. The standard output stream is redirected to a log file and once the performance test has terminated the log is added to the meta-data.

4.4 Dynamic-property Classifier

This section discusses the data analysis techniques used to identify changes in dynamic properties. Classification of dynamic-properties involves smoothing raw training data and test data, grouping of training-data into time-points, generation of dynamic-property models and comparison of test-runs against the model. Dynamic-property data is considered a time-series with each individual property being treated individually. The classification methods used in the dynamic-property classifier are suited to model time-series data.

The analysis pipeline for dynamic properties is outlined in Figure 4.4. Smoothing is applied both to the training-runs from the oracle and the test run being classified. Training runs need to be merged into a list of time-points which represent the distribution of training of values at a point in time. Time-points are passed to the classification algorithm which generates a model in the form of a list of bounds. A bounds is used to test if the data from the training run fits the model. The result of classification is a list of classification differences, which describe differences identified between the model and the test run.

4.4.1 Smoothing

The first step in the dynamic-property analysis pipeline is to smooth the raw training data. The data that is collected tends to be noisy, and shows sharp variation, so

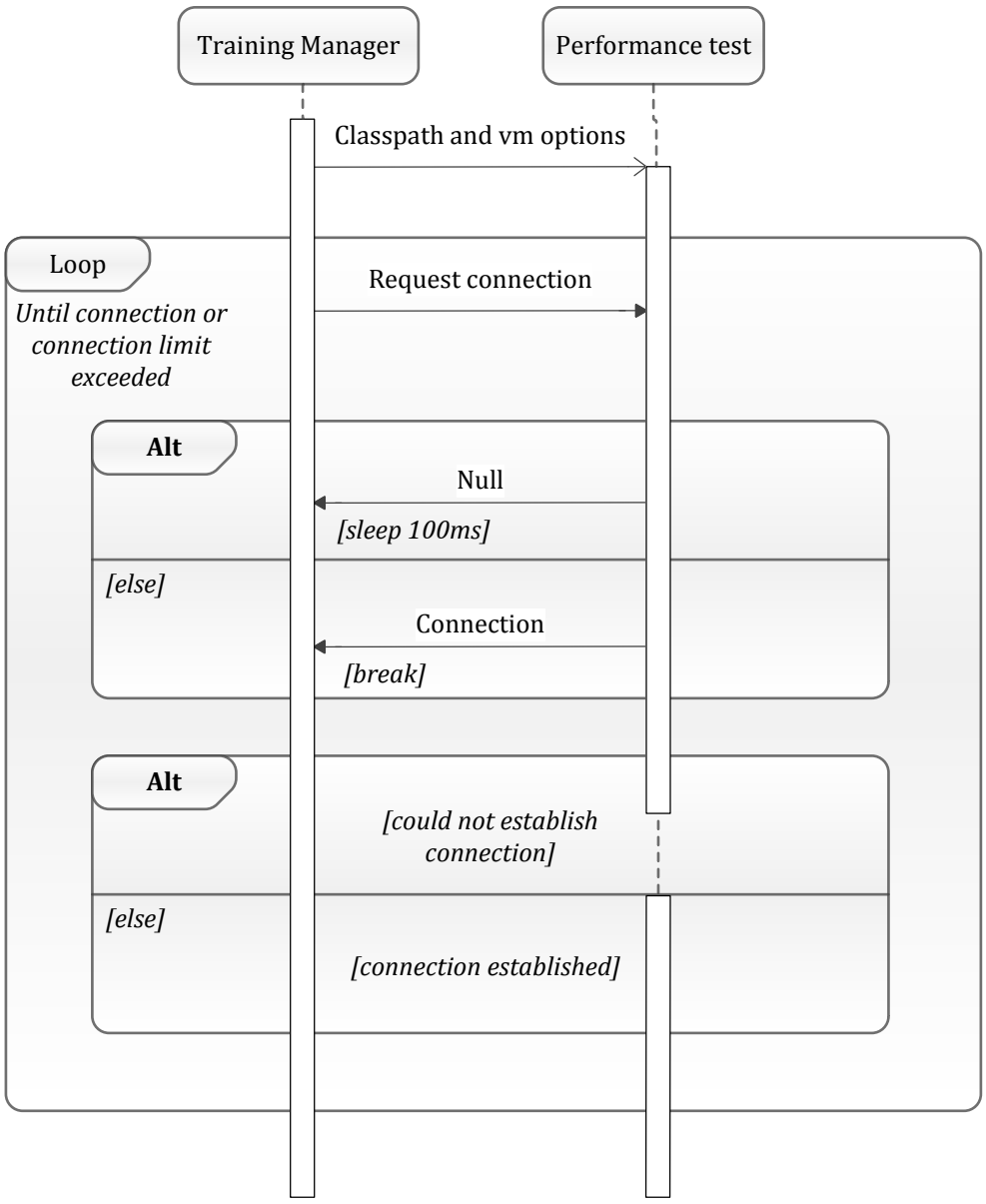


Figure 4.3: Connection sequence diagram. UML 2.0 diagram. Shows the interaction between the training manager and the performance test when establishing a connection.

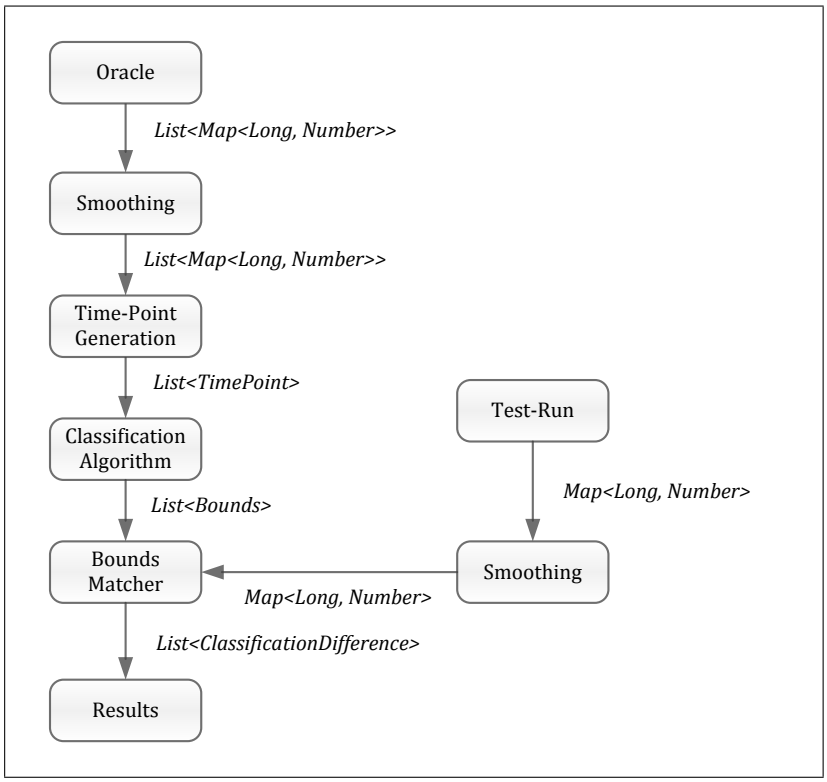


Figure 4.4: Dynamic-property data flow diagram. Shows the flow of data from the raw training data to the model generation, and from test-run to bounds-matching.

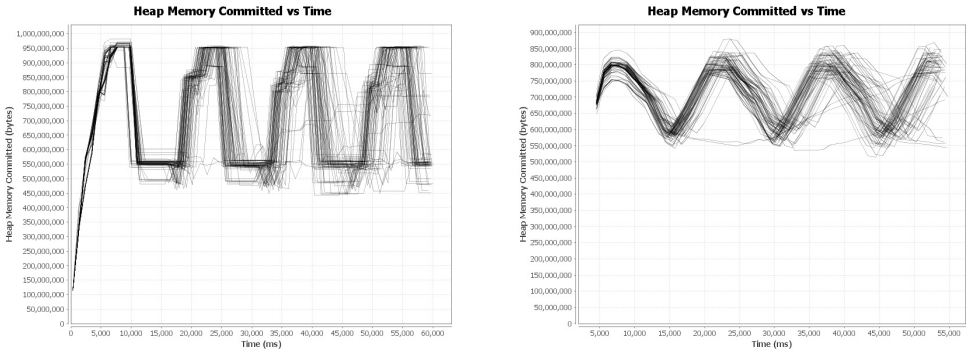
it is useful to employ a filter to smooth the data. Smoothing is applied to both the training data and test data. Two methods of smoothing were tested, mean smoothing and median smoothing. No smoothing was also considered. Smoothing was implemented using a simple smoothing window. It works by replacing the centre value of the window with either the mean or the median of the data within this window, with the size of the window being a parameter that needs to be chosen (Brown, 2004). Smoothing does remove data from the start and end of the test run because there is not enough data to fill the window. The data removed is equal to half a smoothing size at both ends of the test.

Figure 4.5 demonstrates the effects of smoothing. A window size of 9 was used. Smoothing has removed the initial four time-points (four seconds) and the initial build up of heap memory committed. The last four time-points have also been removed, the data now ends at the peak of a garbage collection cycle. The flat periods at the bottom of the cycles are removed and the overall shape of the cycles has been smoothed.

Determining the most effective smoothing configuration will require a tuning test comparing each method of smoothing and Figure 4.1 outlines the parameters to be configured in the smoothing test. Both the mean and median smoothers will be tested using all the window-sizes defined in Section 3.4, the non-smoother will only be tested once.

4.4.2 Time-point generation

Before a set of training data can be used by classification algorithms, the training runs need to merged into sets of time-points. A time-point generator groups to-



(a) Training data (b) Smoothed training data

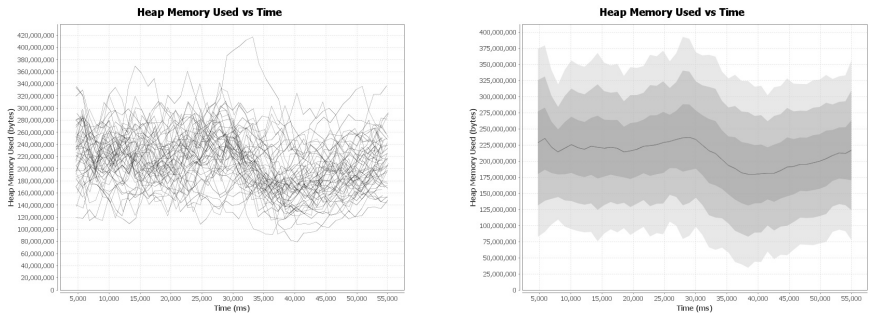
Figure 4.5: Affect of smoothing on data and generated bounds. Figure (a) shows the raw data and (b) the smoothed data. From Java references test program using a mean smoother with a window size of 9.

Table 4.1: Smoothing tuning test plan, no smoothing is only tested once.

Parameter	Values
Smoothing	none, mean, median
Window-size	3,5,7,9,17

gether data points, across multiple training runs, into time-points based on their time-stamp (Hannan, 2009). A time-point represents the distribution of values gathered at that point in time. The classification algorithms then operate on the distribution of the values at the time point to calculate a bounds.

Figure 4.6 demonstrates the effects of time-point generation. 4.6(a) shows the smoothed training data before it is merged into time-points, 4.6(b) shows the distribution of training runs over the course of the test, the middle line indicates the mean value of data points and each level of shading indicates a standard deviation away from the mean. This is how data is received by the classification algorithm.



(a) Smoothed training data (b) Time-point data

Figure 4.6: Smoothed data grouped together using linear time-point generation. Different levels of shading represent the number of standard deviations away from the average value.

Two proposed methods of time-point generation are explored in this thesis. Linear time-point generation groups data based on their index in the collection. Rounding time-point generation places time-points in groups based on their time-stamp. A tuning test to determine which method is more robust is required. Figure

4.2 outlines the test plan for the time-point generation tuning test. The results for the tuning test can be found in Section 5.3.

Table 4.2: Time-point generation tuning test

Parameter	Values
Method	linear, rounding

4.4.3 Classification Algorithms

A classification algorithm is responsible for generating models for the dynamic-properties. Three methods of generating models were explored in this thesis; a simple classifier, Kalman filter based classifier (Welch and Bishop, 1995) and a particle filter based classifier (Sanjeev Arulampalam et al., 2002). The output of a classification algorithm is a list of bounds for each time-point in the training-set.

To build an Oracle, a model for each of the dynamic-properties needs to be generated. A classification algorithm has three input parameters; the data (list of time-points), the dynamic-property's meta-data, and classifier configuration parameters. The property's meta-data is used to enforce bounds on the permissible range of that property, for example, preventing bounds being generated for a negative thread count. The result of a classification algorithm is list of bounds that specify the model. A bounds-matcher is then used to compare test-runs against the model and identify where the test does not match the model.

4.4.4 Simple Classifier

The simple classifier generates bounds using simple statistics regarding the distribution of the test run data at each time-point. An averaging method is used to calculate the expected value at a time-point, the standard deviation of the points is then used to calculate an upper and lower bound. The number of standard deviations used for bounds calculation is a parameter that can be configured. Two averaging methods are considered, the mean and the median.

Listing 4.1 shows pseudo code for the simple classifier implementation. The classifier processes each time-point in succession using the mean and standard deviation of the time-point to calculate the bounds.

```
simpleClassifier(timepointList){
  // Output bounds
  boundsList = []
  // Iterate over time-points
  for timepoint in timepointList{
    // Predicted distribution is the measured distribution
    measuredState = timepoint.mean \\ or timepoint.median
    measuredStdev = timepoint.stdev
    // Calculate devation from predicted state
    deviation = measuredStdev * deviations
    // Calculate bounds
    lowerBound = measuredState - deviation
    upperBound = measuredState + deviation
    // Add new bounds
    boundsList.add(new Bounds(timepoint.timestamp, lowerBound, upperBound))
  }
  return boundsList
}
```

Listing 4.1: The simple classification algorithm, uses the distribution of training data at each time-point to generate bounds.

The simple classifier is configured by two parameters.

Average method The method of averaging to use when calculating bounds.

Deviations The number of predicted standard deviations from the predicted state to use for generating bounds.

Only the average method will require a tuning test, the deviations are used in the overall test. Table 4.3 shows the test plan for the simple classifier. Results for the simple classifier tuning test can be found in Section 5.5.1.

Table 4.3: Simple classifier test plan.

Parameter	Values
Average method	mean, median

4.4.5 Kalman filter

In 1960, R.E. Kalman published his famous paper describing a recursive solution to the discrete-data linear filtering problem (Kalman et al., 1960). Since then the Kalman Filter has been applied in numerous ways. It is commonly used for guidance, navigation and control of vehicles, particularly aircraft and spacecraft. The algorithm uses a series of measurements observed over time, containing noise and other inaccuracies, to produce estimates of unknown variables that often are more accurate than single measurements alone. The implementation used in this thesis works in one dimension.

The Kalman classifier can be configured by the following three parameters.

Average method The method of averaging used when calculating the measured state.

Deviations The number of predicted standard deviations from the predicted state to use for generating bounds.

Process noise method The method used to calculate the process noise for the Kalman filter equations.

The average method and default predicted variance parameters will require tuning tests to find the optimal configuration of the classifier before it is tested against the other classification methods. Table 4.4 outlines the parameters that need to be tuned and their possible values. Results for the Kalman filter tuning tests can be found in Section 5.5.2.

Table 4.4: Kalman filter test plan.

Parameter	Values
Average method	Mean, Median
Process noise method	Average variance, Average difference

Pseudo code for the Kalman filter implementation used in this thesis is presented in Listing C.1. Listing C.2 contains the code for the two methods of calculating the process noise for the Kalman filter equations. The average difference method uses the difference between each subsequent pair of time-points, the average value is used as the estimate for the noise in the process. The average variance method uses the average standard deviation for each time-point as the estimate for the process noise.

4.4.6 Particle filter

Particle filters comprise a broad family of sequential Monte Carlo algorithms for approximate inference in partially observable Markov chains (Thrun, 2002). Particles filters have been applied in many ways most prominently in robotics, solving the global localization problem (Fox et al., 1999) and the kidnapped robot problem (Engelson and McDermott, 1992). It has also been used heavily in tracking (Montemerlo et al. (2002); Schulz et al. (2001); Yang et al. (2005)). The particle filter was chosen as the counterpart to the Kalman filter. Both algorithms are based on similar notions, however the Kalman filter is optimal in systems with Gaussian noise. In a system that is non-linear, the Kalman filter can still be used for state estimation, but the particle filter may give better results. If the process noise is not linear the particle filter should perform better than the Kalman filter (Sanjeev Arulampalam et al., 2002).

This implementation of the particle filter can be configured using the following properties.

Deviations The number of deviations from the predicted state to use for bounds calculations.

Particle count The number of particles to sample.

Weighting method Method used to calculate the weightings of particles given an observation distribution. `Distance`, the euclidean distance between the observed state and a particle's state is used to calculate weightings.

`Gaussian`, the density of the particles state compared to the distribution of the observation.

Propagate method Method used to propagate particles for the next time step. Use the current velocity of the predicted states and add system noise. Or just add system noise.

Iterations The number of iterations of the algorithm to average bounds over. Can be any integer greater than 0.

Re-sample percentage The percentage of effective sample size required for re-sampling. A double between 0 and 1.

Because the particle filter uses random sampling to select the next generations of particles, it is a random process whose results may vary between executions. To account for this variation the particle classifier has a parameter iterations which specifies the number of iterations of the algorithm to average bounds over. Increasing the iterations also increases the running time of the algorithm, too many iterations will cause the running time to become too expensive.

A metric known as effective sample size is used to determine when the particle filter re-samples its population of particles. Effective sample size is a measure of how well the distribution of the particles match the measurement distribution. It is calculated using all of the particles updated weightings. The re-sample percentage parameter determines at what level of degeneracy compared to the predicted is required before the particle filter re-samples it's particle set.

For each iteration of the particle classifier the particle's state can be updated based on the current velocity of the estimated state. This tests whether using the velocity update method or no update method has a better performance.

The first step in processing a new measurement by the particle filter is to update the weightings of each particle based on the measurement distribution. Two methods were implemented for this classifier, absolute distance and probability density function.

The particle count is the number of samples used for each iteration of the filter. Increasing the particle count should increase the accuracy of the filter but also will increase the running time of the algorithm, so selecting a particle count value that is not too large is important.

The particle filter is the most complex classification method implemented with six parameters, five of which will require tuning tests. These five parameters need to be tuned so an an optimal configuration can be selected for testing against other classification algorithms. Each parameter has a set of values that will be compared against each other and a default value that will be set when testing other parameters. Results for the particle filter tuning tests can be found in Section 5.5.3.

Psuedo code for implementation of the particle filter used in this thesis is presented in Listings (C.3 - C.7). Listing C.3 outlines the core of the algorithm. First the particles are distributed around the initial state of the time-point and are assigned an initial weighting.

The first step in processing a time-point is to update each particles state based on the current velocity of the predicted state. Process noise is added to each particle during updating. Next, the measurement is taken as the average and standard deviation of the time-point and used to update the weightings of all particles. A particle's weighting is proportional to its distance from the measured state. Two

Table 4.5: Particle filter parameters.

Parameter	Values
Particle count	10, 100, 1000, 10000
Iterations	1, 2, 4, 8
Re-sample %	0.1, 0.2, ..., 0.9, 1.0
Propagate method	none, velocity
Weighting method	pdf, distance

methods for calculating this distance are been explored. Particle weights are then normalised.

The effective sample size is a measure of the degeneracy of the particle distribution (Sanjeev Arulampalam et al., 2002) and when its value falls below the re-sample percentage parameter the particle set is re-sampled. The predicted state and predicted variance is then calculated based on the distribution of the particles. The prediction is then used to calculate the bounds for the time-point. Finally the velocity of system is updated by calculating the change since the previous time-point. Listing C.4 shows the method used to calculate the effective sample size.

The method used to re-sample the particle set is outlined in Listing C.5. Particles are sampled from the collection with replacement and relative to their weighting, heavily weight particles will be sampled more often. At this point in the algorithm all particle weights are normalised. First the particle list is sorted based on their weighting, smallest to largest. A cumulative weighting list is generated based on the sorted particle list. A particle is sampled by generating a random number between 0 and 1 and searching for the index where it should be inserted in the cumulative weightings list, this index is the particle that is sampled. A new set of particles of the same size as the old set is generated.

The two methods for updating the particles state are shown in Listing C.6. Particle state can be either be update with or without using the velocity of the system. Process noise is added at this point in the algorithm.

Listing C.7 shows the two methods to calculate the weightings of particles. The first method uses the inverse of the absolute distance between the measured stated and the particles' state. The second method uses the measurement distribution to calculates the density of the particle's state against the distribution.

4.4.7 Bounds Matcher

The bounds matcher is responsible for comparing test runs against a generated model. It takes test run data and matches it to a bounds object. Two methods of matching a test data point to bounds have been implemented for this thesis. The first method matches a data point to the nearest bounds object, the second uses interpolation to estimate the bounds values between time-points. A bounds matcher is created using a generated model and test runs are compared against the model resulting in a list of differences between the test run and the model.

When a data point does not fit the model a classification model difference object is generated. The object contains the time-stamp, the value of the test-run data and the bounds object which the value was classified against. Classification

model differences are generated in two situations; when the test data is matched to a bounds but falls outside of it and when the time-stamp cannot be matched to a bounds object.

A tuning test was performed to determine the best performing bounds matcher. The best performing method will be used in the overall test. The test plan for the bounds matcher is outlined in Figure 4.6. Results for the bounds matcher tuning test can be found in Section 5.4.

Table 4.6: Bounds matcher test plan.

Parameter	Values
Bounds matcher	nearest bound, interpolation

The two methods differ in their implementation of the `getBounds()` method, which matches a bound to a time-stamp. Pseudo code for the implementation of the nearest-bound matcher is outlined in Listing C.8. The list of bounds is traversed and the bound that is the closest to the time-stamp is returned. If the time-stamp is outside the range of the model `null` is returned.

The interpolation bounds matcher is outlined in Listing C.9. The method searches the model for the bounds the time-stamp falls within. It then uses linear interpolation to estimate an upper and lower bound for the time-stamp, this new bounds object is then returned. If the time-stamp is less than the first bound, the first bound is returned, or, if it is greater than the last bound and within a collection interval, the last bound is returned.

Interpolation is a method of curve fitting using linear polynomials. In this case two points are known as well as the x value of a third which is between the first two points. This information is used to estimate the y value for the third point. The linear interpolant is a straight line between the first two points, the y value is calculated as the y intercept of the interpolant and the x value. Listing C.10 outlines the code to perform linear interpolation.

4.5 Meta-Data Classifier

The Meta-data classifier is responsible for identifying changes in a performance test's meta-data.' It is essential when classifying test-runs or building a model from training-runs, that all the runs have been generated in a similar environment. Otherwise changes in the resource utilisation of a performance test may be caused by a change in the environment the test is executed in. The Meta-data classifier is responsible for ensuring the meta-data is consistent between the training-set for learning and for the test run being classified.

A filter list is used to ignore meta-data that is often inconsistent. An example of meta-data that is normally inconsistent is the standard output for JVM. This data is useful for debugging purposes, but is too variant to use in meta-data classification. The filter list is used to ignore meta-data items that not relevant to classify, like the standard output.

Meta-data is handled in the form of maps of keys to objects, all objects are primitive types that can be compared using built in Java comparison functions. First a model of the meta-data is constructed using the training data, then test run meta-data is compared against the model and a list of differences is generated.

Creating the meta-data model simply checks all training maps for consistency. This ensures all training-runs used in classification have the same meta-data and were run in similar test environments. Each meta-data property contained in a training-run must be present in each other training-run and the values must be equal. If one of the properties of one of the training-runs do not adhere to this then an error is thrown when trying to create the meta-data model. This indicates that the meta-data of the training-set is not homogeneous and some test runs may have been executed in a different environment.

Errors can be thrown during the meta-data model creation process. Two types of errors can occur.

- A training-run contains a meta-data item that is not present in other training-runs.
- A training-run contains a meta-data item whose value differs from values in other training runs.

Once a model has successfully been built test-runs can be classified against it. The results of a classification is a list of differences between the test-run and the model. Differences have the three following causes.

- The test-run contains a meta-data item whose value differs from the model's value.
- The test-run is missing meta-data items which are present in the model.
- The test-run contains meta-data items that are not present in the model.

Listing C.11 contains pseudo code for the meta-data training algorithm. First, all data in the filter list is removed from all training-runs. If there is one training-run it is used as the model, otherwise the first map is chosen as the base map. Every other training-run is then compared against the base-map. Each key from the base map is checked for in the other maps, values compared and extra keys are tested for. If a map is found to be inconsistent against the base map an error is thrown and the model cannot be generated. If all maps are consistent then the base map is used as the model meta-data classifier.

Classifying a test-run against a model is similar to building the model, the test-run map is tested for consistency against the training-run map. Listing C.12 outlines the method used to classify test-runs. First data on the filter list is removed from the test-run meta-data, then all keys are checked for and their values tested for equality against the model. Finally any extras present in the test-run meta-data are tested for. If a difference is found, its reason is noted and added to the list of differences.

4.6 Notification Classifier

The notification classifier is responsible for building a model for the notification emissions of performance test and testing test-runs against models. A model is built by first grouping together instances of notifications from each training run. Then, using the distribution of the groups time-stamps a set of bounds is calculated for the grouping. When classifying a test-run, notifications are matched to their

grouping and its bounds is used to test classify the instance. If its time-stamp falls outside the bounds it is flagged as erroneous.

Notifications consist of the following fields.

Type The type of notification being emitted.

Source Object The name of the object that emitted the notification.

Message The message of the notification.

Sequence number The index of the notification relative to the source object.

Time-stamp The time since the JVM's invocation when the notification was emitted.

Notifications are grouped together by generating a key using the type, source object and sequence number. A notifications sequence number is a serial number identifying a particular instance of notification in the context of the notification source (Oracle Corporation, 2013d). This combination of properties will uniquely identify a notification in a test-run. A map is used to group notifications into lists based on their key.

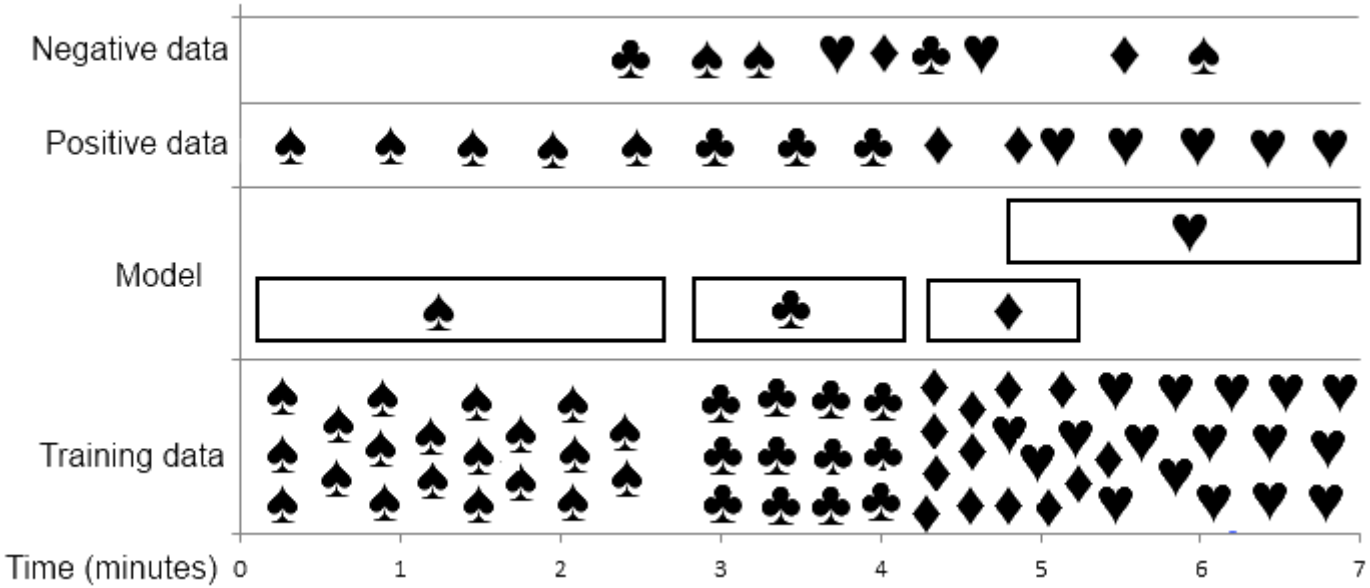
A diagram to demonstrate how the notification classifier works is presented in Figure 4.7. Training data supplied in the form of a list of notification lists which is used to generate the model, and test notifications can then be classified. Notifications a grouped using their properties. A set of bounds for each grouping is calculated. This forms the model used to classify test-runs. Positive and negative test data is demonstrated at the top of the figure. The result classification is a list of notification difference objects which describe how notifications from the test-run differ from the model.

When creating a model an error can be thrown when the groupings form inconsistently sized bunches. This indicates that the data used to build the model is inconsistent and there was either more, less or different emissions in one of the training-runs. The following differences between a notification model and test-run can occur.

- A test-run notification's time-stamp falls outside of its corresponding bounds.
- A test run notification cannot be matched to a bounds.
- A bounds cannot be matched to test-run notification.
- A test-run contains notifications that map to the same key.

Listing C.13 illustrates the method to generate notification models. A map of lists is created to group the notifications. Each notification from each training-run is processed and added to the grouping map. This consists of generating a key for each notification, checking if a list is present in the map and adding to that list, else a new list will be crated and added to the map. Once all notifications have been processed the groupings are tested for regularity. If one grouping is smaller or larger than the others this indicates abnormal training data where different notifications were emitted between runs, and an error is thrown to prevent model creation. Next the bounds for each grouping is generated by calculating the distribution of the time-stamps amongst the group, an upper and lower bound for the grouping is calculated based on this distribution.

Figure 4.7: Notification classifier algorithm explanation. Training data is shown at the bottom, with each instance coming from a separate training run. The model based on the training data is shown above. Positive and negative data instances are then shown above. Each shape represents a different notification type.



Listing C.14 illustrates the method used to classify test-runs against the notification model. First a list of unprocessed bounds is created from the model. Then, for each notification a key is generated and the model is checked for the corresponding grouping. If a grouping is found the time-stamp of the notification is tested against the bounds of the grouping, else a missing notification difference is added to output list. If the grouping has already been processed then there are duplicate notifications in the test-run and a difference is added to the output list. If a grouping was matched to a bounds, that bounds object is removed from the unprocessed bounds list. After all test-run notifications have been processed all unprocessed bounds are considered to be a difference between the model and test, and are added to the differences list.

4.7 Test-Run Classifier

The test-run classifier is responsible for building a model of a set of training-runs and classifying test-runs against the model. It is responsible for building models for each of the dynamic properties and initialising the meta-data classifier and the notification classifier. It applies the dynamic analysis pipeline by performing smoothing, time-point generation, model generation and bounds matching. The result from a test-run classification is a set of differences identified by the dynamic-property classifier, differences identified by the meta-data classifier and differences identified by the notification classifier.

Pseudo code for training the test-run classifier is outlined in Listing C.15. First, models for each of the monitored dynamic properties are generated. This involves extracting property data from each test-run, smoothing it, applying time-point generation and model generation. Models are placed in a map for later use when classifying test-runs. Next the meta-data classifier and the notification classifier are constructed using data from the training-runs. Once this is complete the classifier is trained and ready to classify training runs.

The classification method is outlined in Listing C.16. First the dynamic-properties are classified against their models. This involves extracting property data from the test-run, smoothing the data and using a bounds matcher to compare the test-run to the model. The results from each classification are stored in a map. Next the meta-data and notification data is extracted from the test-run and classified. The results from all the classifiers are returned in a classification result object.

Chapter 5

Experimental Results

This section presents the results for the experiments defined in Section 3.4. First, a tuning test to assess the optimal training-set size was performed. Next the different preprocessing methods were evaluated, before tuning each classification method, and then the methods were compared against each other. Finally, the results from the notification classifier are presented. The implementation of the meta-data classifier is not novel, and the results from the meta-data classifier do not need to be examined.

All experiments were run on a machine running Windows 7 Enterprise 64bit, Intel(R) Core(TM) i7 CPU 920 @2.67GHz with 6GB of RAM. The Java version used was the Java HotSpot(TM) 64-Bit Server VM version 23.7-b01.

A build of the tool has been provided, along with the training data used in the experiments for this thesis. All the experiments presented here can be repeated so the results can be re-examined. Guidelines for downloading the build and executing the tests can be found in Appendix A.

All tables are ranked based on the MCC rating of each configuration. The accuracy and the precision is shown to provide more insights about the robustness of the classification methods. The MCC rating is defined in Section 3.4.

5.1 Training Size

A tuning test was performed to assess the optimal training-set size. A range of training-set sizes was tested and their performance compared with each other. Increasing the size of the training-set should improve the accuracy of the classifier because it has more knowledge of the test to use when generating bounds. However increasing the training-set size will increase the computation cost. Also, sufficient training runs need to be kept in reserve to perform cross validation of the model.

The training-set size tuning test was conducted using all of the four test programs, see Section 3.1. Only a single classifier configuration was used, this was to reduce the running time of the experiment and condense the output size. The simple classifier, using the mean average type and four deviations was chosen. This is due to the fact that it is the simple solution to the problem and considered to be the benchmark. Based on other tuning tests, the optimal configuration of the analysis components was used. Smoothing was performed using the mean smoother with

a window-size of 9, using linear time-point generation and the nearest-bounds matcher.

There are a total of 32 experiments across all of the test-programs. All of the experiments will be run for each of the tuning tests. A range of training-set sizes from 10 to 80 were tested. For each variant, 100 training-runs were generated, leaving 20 test-runs left over for validation of the model. Positive and negative data set sizes were set at 20 test-runs. Performing 32 experiments with 20 positive and negative test instances each experiment, produces 640 positive and negative test runs to classify.

Table 5.1 shows the results for the training-set size experiments. The results follow the intuition that increasing the training-set size will increase the robustness of the classifier. The training-set size of 80 performed the best with an MCC rating of `0.86`, contrasted with a training-set size of 10 which performed at `0.50`. There is a marked jump from 10 to 20 training runs with a `13%` increase in accuracy and a `0.21` increase in the MCC rating. The returns for an increase after a size of 20 are less in both accuracy and MCC. Diminishing returns occur around a training-set size of 50 giving an MCC rating of `0.86`, `0.02` less than a training-set size of 80. A training-set size of 50 will be used for all future experiments, giving positive and negative data set sizes of 50. With 32 experiments and a test set size of 50 there will be 1600 positive and negative test instances for all future tests.

Table 5.1: Training tuning test results. Shows results across all test programs using the simple classifier (deviations 4). All experiments used a test-set size of 20.

Training-set size	TP	FN	TN	FP	Accuracy %	Precision %	MCC
80	557	83	632	8	93	98	0.86
70	543	97	631	9	92	98	0.84
60	540	100	591	9	91	98	0.84
50	531	109	598	2	91	99	0.84
40	514	126	600	0	90	100	0.82
30	481	159	600	0	88	100	0.78
20	428	212	600	0	83	100	0.71
10	159	381	600	0	70	100	0.50

Figure 5.1 shows the affect of altering the training size on the generated model. These graphs are examples of the Eden space memory committed from the Java references example (strong object manager compared against the soft object manager). There is a significant difference between a training-size of 10 5.1(a) and 20 5.1(b), which is only apparent after investigating the y-scale of the two graphs. The peak value of the model at `20,000 ms` has been reduced by `20,000mbs` as well as the trough at `15,000ms` has been reduced by `2,0000mbs`. The models from training-set size 40 (5.1(c)) to 80 (5.1(d)) remain relatively similar with slight changes to the bounds as more data is available.

5.2 Smoothing

This section presents the results for the smoothing tuning tests. Smoothing is used to remove noise from the data to improve the robustness of the classifier. The two smoothing methods, outlined in Section 4.4.1, are compared using a range of smoothing window-sizes. No smoothing is also tested.

The chosen window-set sizes are `[3,5,8,17]`, as defined in Section 3.4. Tests were performed with a training-set size of 50 with positive and negative set sizes of 50. The simple classifier using the mean average type and four deviations was used, along with linear time-point generation and nearest neighbour bounds matcher.

Table 5.2 shows the results for the smoothing tuning tests. The best performing method tested is the mean smoother, using a window-size of 17, receiving a MCC rating of `0.89`. No smoothing is the worst performing method tested with a `0.63` MCC rating and `79%` accuracy. Overall, mean smoothing outperformed median smoothing, the median smoother for the same window-size always performed worse than the mean smoother. Also, as the smoothing size increased, for both smoothers, the accuracy of the results increased.

Table 5.2: Smoothing tuning test results. Shows rankings mean and median smoothers with a range of window-sizes, as well as non smoothing.

Configuration	TP	FN	TN	FP	Accuracy %	Precision %	MCC
Mean - 17	1408	192	1600	0	94	100	0.89
Mean - 9	1358	242	1594	6	92	100	0.85
Median - 17	1258	342	1584	16	89	99	0.79
Mean - 5	1244	356	1561	39	88	97	0.77
Median - 9	1190	410	1572	28	86	98	0.75
Mean - 3	1157	443	1570	30	85	97	0.73
Median - 5	1096	504	1578	22	84	98	0.7
Median - 3	1035	565	1576	24	82	98	0.67
No smoothing	945	655	1578	22	79	98	0.63

Figure 5.2 shows the affect of smoothing on the training data and the bounds generated. The five figures show the heap-memory-used for an experiment performed with the consumer producer test program. No-smoothing is shown in Figure 5.2(a), it consists of a relatively regular pattern oscillating between `500,000mbs` and `1,100,000mbs`, which appears to degrade towards the end of the test. There is marked difference between the no smoothing and using a window size of 3 (4.5(b)). The initial portion has been smoothed reducing the range of the oscillation and the pattern change half-way through the test is more apparent. The same process continues through smoothing sizes 5 (5.2(c)), 9 (5.2(d)) and 17 (5.2(e)). The graph produced by a smoothing window of 17 is noticeably different from the smoothed graph, which is to be expected.

5.3 Time-point generation

This section presents the results for the time-point generation tuning tests. Two methods for merging training-runs into time-points have been tested in this thesis.

Tests were performed with a training-set size of 50 with positive and negative test sizes of 50. The simple classifier using the mean average type and four deviations was used, along with mean smoothing (window size of 9) and nearest neighbour bounds matching.

Table 5.3 shows the results for time-point generator tuning test. The linear time-point generator performs the best with an MCC rating of `0.85`, while the rounding method provides an MCC of `0.33`. The linear time-point generator performs better and will be used for all future tests.

Table 5.3: Time-point generator tuning test results.

Time-point generator	TP	FN	TN	FP	Accuracy %	Precision %	MCC
Linear	1358	242	1594	6	92	100	0.85
Rounding	1038	562	1595	5	65	100	0.33

5.4 Bounds Matching

This section presents the results for the two bounds matching algorithms. The bounds matcher is responsible for matching data points from test runs to the bounds that will be used to classify the data value. Two methods of matching bounds were tested; nearest bound and interpolation.

The test was performed with a training-set size of 50 with positive and negative test sizes of 50. The simple classifier using the mean average type and four deviations was used, along with linear time-point generation and nearest neighbour bounds matching.

Table 5.4 shows the results for the bounds matcher tuning test. Both methods perform reasonably well, the nearest bound method performed the best with `0.85` MCC rating, but the interpolation method was close behind with `0.80`. The nearest bound method offers superior results and will be used in classification from now on.

Table 5.4: Bounds matching tuning test results.

Method	TP	FN	TN	FP	Accuracy %	Precision %	MCC
Nearest bound	1358	242	1594	6	92	100	0.85
Interpolation	1252	348	1595	5	89	100	0.80

5.5 Classifier tuning

This section presents the tests performed to tune each of the classification methods. The simple classifier has one parameter, the Kalman classifier two, and the particle classifier has five. For each parameter, in each classifier, a tuning test was performed to determine the best choice of parameters for that classifier. Determining the best configuration was an iterative process. For each of these tuning tests the other parameters for the classifier were fixed. Initially the best parameters

were picked based on intuition, but when tuning tests identified the best performing parameter the default set was updated and the rest of the tuning tests had to be re-run . Only the results from the last set of tuning tests with the best parameter selection have been presented here.

For all tests in this section the analysis parameters are fixed. Tests were performed with a training-set size of 50 with positive and negative test sizes of 50. Smoothing was performed using the mean smoother with a window-size of 9, with linear time-point generation and the nearest bound matcher algorithm.

5.5.1 Simple Classifier

The simple classifier uses the distribution of the values at each time-point to estimate the allowable bounds for the model being generated. The mean or median can be used to calculate the average and the standard deviations is used to generate the range of permissible values. Only one tuning is required to select whether the mean or median should be used to calculate the average value.

Table 5.5 shows the results for the average method tuning test. Using the mean as opposed to the median provides a `0.01` increase in MCC and the difference between the two methods is unlikely to be statistically significant.

Table 5.5: Simple classifier average method tuning test results.

Average type	TP	FN	TN	FP	Accuracy %	Precision %	MCC
mean	1358	242	1594	6	92	100	0.85
median	1338	262	1594	6	92	100	0.84

5.5.2 Kalman classifier

This section presents the tuning tests for the Kalman classifier. It is configured using three parameters, two will be tested here; the average method and process noise method. The average method decides whether to use the mean of the median as the measured state for the Kalman filter equations. The process noise method decides how to estimate the noise present in the process, which is an important parameter when tuning Kalaman filters.

The results for the Kalman filter average type tuning test are shown in Table 5.6. For this experiment the average variance process noise method was used. The MCC rating for both methods was `0.5`, however the mean average type performed slightly better with five more true positives. This variation is unlikely to be statistically significant, however the mean method will be used for future Kalman filter experiments.

Table 5.6: Kalman filter average type tuning test results.

Average type	TP	FN	TN	FP	Accuracy %	Precision %	MCC
mean	757	843	1550	50	72	94	0.5
median	752	848	1550	50	72	94	0.5

Table 5.7 shows the results for the Kalman classifier process noise tuning test. The average variance method performed better with a `0.15` higher MCC rating than the average difference method. However, the average difference method does offer better precision, but the MCC rating is still lower than the average variance method. The average variance method will be used for future Kalman filter experiments.

Table 5.7: Kalman classifier process noise tuning test results.

Process noise method	TP	FN	TN	FP	Accuracy %	Precision %	MCC
averageVariance	757	843	1550	50	72	94	0.51
averageDifference	371	1229	1600	0	62	100	0.36

5.5.3 Particle Classifier

This section presents the tests performed to tune the particle classifier. There are five parameters that need to tuned; iterations, re-sample percentage, propagate method, update weighting method, and particle count. The particle classifier relies on randomness for the algorithm to work, so the output is likely to vary slightly between test runs. Table 5.8 lists the default parameters for all particle filter tuning tests.

Table 5.8: Default parameter values for particle filter tuning tests. Also the best performing configuration for the particle classifier.

Parameter	Value
Deviations	6
Iterations	8
Re-sample %	0.6
Propagate method	Velocity
Weighting method	pdf
Particle count	1000

Table 5.9 shows the results for the iterations tuning test. There is not a significant difference between the tests, with all variants falling within `0.08` MCC rating of each other. The results follow the intuition that increasing the number of iterations will increase the accuracy of the classifier. An iteration count of eight performs the best with a `0.61` MCC rating and will be used for future tuning tests.

Table 5.9: Particle filter iterations tuning test results.

Iterations	TP	FN	TN	FP	Accuracy %	Precision %	MCC
8	960	640	1550	50	78	95	0.61
2	899	701	1571	29	77	97	0.6
4	908	692	1550	50	77	95	0.59
1	793	807	1550	50	73	94	0.53

Table 5.10 shows the results of the re-sample percentage tuning test. Changes in the parameter does not have a significant impact on the accuracy of the classifier, except when this is set to 1 which prevents re-sampling. The best configuration is 0.6 with an MCC rating of `0.65`. Performance drops as the re-sample percentage gets farther away from 0.6.

Table 5.10: Particle classifier re-sample percentage tuning test results.

Re-sample %	TP	FN	TN	FP	Accuracy %	Precision %	MCC
0.6	985	615	1563	37	81	96	0.65
0.7	976	614	1552	48	80	95	0.64
0.4	979	621	1550	50	80	95	0.64
0.9	895	705	1585	15	79	98	0.62
0.5	931	669	1552	48	79	95	0.61
0.8	837	763	1591	9	77	99	0.6
0.3	895	705	1550	50	77	95	0.59
0.2	825	775	1550	50	75	94	0.55
0.1	763	837	1550	50	73	94	0.52
1	495	1055	1600	0	66	100	0.43

For each iteration of the particle classifier a particle's state can be updated based on the estimated velocity of the process. Table 5.11 shows the results for the propagate method tuning test. The velocity based method slightly outperforms the no update method with a `0.01` increase in the MCC rating of the particle classifier. This not statistically significant, however the velocity method will be used in future tuning tests.

Table 5.11: Particle classifier propagate method tuning test results.

Method	TP	FN	TN	FP	Accuracy %	Precision %	MCC
velocity	1007	593	1551	49	80	95	0.64
none	988	612	1550	50	79	95	0.63

Two methods for updating particle weights were chosen; absolute distance and probability density function. Table 5.12 shows the results for the weightings method tuning test. The probability density function method outperforms the absolute distance function with a `0.45` increase the MCC rating. The distance function offers better precision but at the expense of accuracy.

Table 5.12: Particle classifier weighting method tuning test results.

Method	TP	FN	TN	FP	Accuracy %	Precision %	MCC
pdf	1012	588	1550	50	80	95	0.64
distance	115	1485	1600	0	54	100	0.19

The particle count dictates the number of samples used for each iteration of the filter. Table 5.13 shows the results for the particle count tuning test. The best performing particle count is 1000 with an MCC rating of `0.62` which is only `0.01` better than the next best performing configuration. Interestingly 10000 particles

performs worse than 1000 which might be explained by over fitting the distribution with too many particles. However, the difference is small and not statistically significant. 10 particles performs the worst by far, with `0.26` reduction of the MCC rating.

Table 5.13: Particle classifier particle count tuning test results.

Particle count	TP	FN	TN	FP	Accuracy %	Precision %	MCC
1000	897	703	1600	0	0.78	1	0.62
10000	962	638	1550	50	0.79	0.95	0.61
100	898	702	1550	50	0.77	0.95	0.58
10	362	1238	1600	0	0.61	1	0.36

5.6 Overall Results

This section presents the overall results for the classifiers. Each classifier is tested using the best configuration identified by the previous tuning tests. A range of deviations is used for each classifier. Results are aggregated across all test projects to show how the classifier works over a range of problems. Then the results for each test program are presented individually and interesting experiments are highlighted. Only a sample of the results produced from the classifier are presented here, the nature of the problem produces a significant amount of data which is impractical to include in this thesis. See Appendix A for repeating tests.

The overall results for the experiments are presented in Table 5.14. For all results; S = simple classifier, K = Kalman classifier, P = particle classifier. Values next to numbers are the deviations used. Only the top 15 results have been shown for the interest of saving space. These results are aggregated across all properties for each test-run, meaning that if one property of a test run is classified as negative then the whole test run is flagged as negative. The best performing configuration was the simple classifier using 20 deviations with `97%` accuracy, `97%` precision and a MCC rating of `0.94`. Most of the simple classifier's configurations are in the top 15 with only the two configurations not making it. The top particle and Kalman classifiers were 10^{th} and 12^{th} respectively.

The overall results of the classifier are biased towards the negative. This is due to each classification consisting of multiple dynamic-property classifications, which, if a change is detected in one dynamic-property the entire classification will be flagged as negative. This why the largest deviations for the simple and particle classifier perform the best, and why the classifier appears to have such good precision. However, this is to be expected due to the nature of the classifier. Investigating the rankings for individual dynamic-properties will provide more insight into the performance of the classifiers.

Table 5.15 shows the aggregate results across all test programs for the heap-memory-used dynamic-property. The results here are noticeably different from those presented in Table 5.14. The best performing classifier is still the simple method but with deviations of 4. The simple classifier also does not dominate the top of the rankings, the particle filter comes in 3^{rd} to 5^{th} and the Kalman filter coming in 7^{th}.

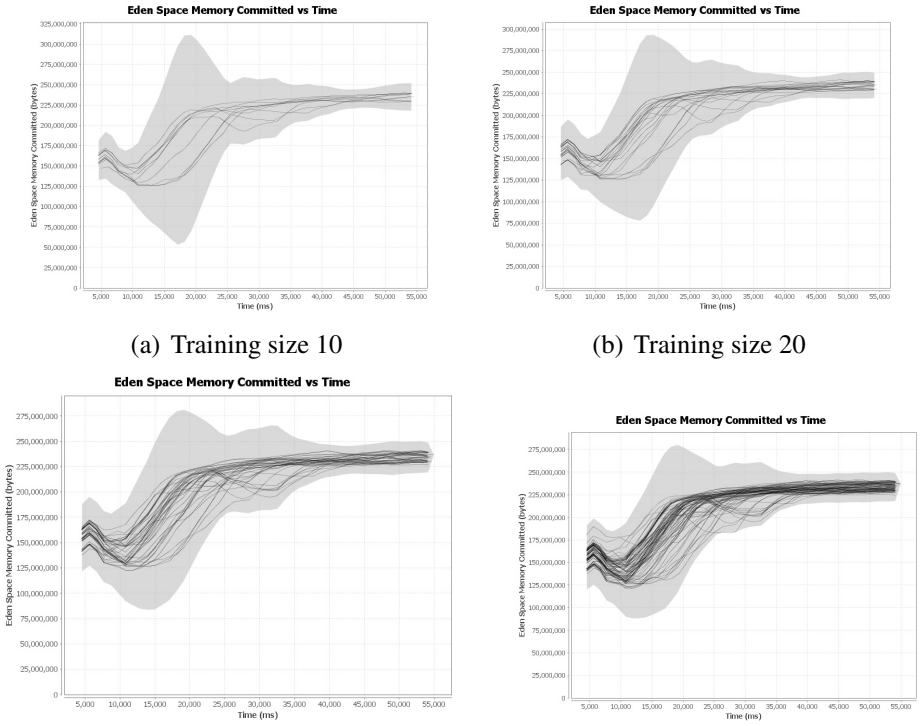
Table 5.14: Overall classification results aggregated across all test programs for all properties. Top 15 of 30 configurations shown, ranked by MCC. S = simple classifier, K = Kalman classifier, P = particle classifier. Values next to numbers are the deviations used.

Config	TP	FN	TN	FP	Accuracy %	Precision %	MCC
S - 20	1554	46	1550	50	97	97	0.94
S - 18	1552	48	1550	50	97	97	0.94
S - 16	1548	52	1550	50	97	97	0.94
S - 14	1544	56	1550	50	97	97	0.93
S - 12	1532	68	1550	50	96	97	0.93
S - 10	1516	84	1550	50	96	97	0.92
S - 8	1493	107	1550	50	95	97	0.90
S - 6	1458	142	1550	50	94	97	0.88
S - 4	1358	242	1594	6	92	100	0.85
P - 20	1143	457	1550	50	84	96	0.71
P - 18	1134	466	1550	50	84	96	0.7
K - 20	1127	473	1550	50	84	96	0.7
P - 16	1116	484	1550	50	83	96	0.69
P - 14	1095	505	1550	50	83	96	0.68
K - 18	1095	505	1550	50	83	96	0.68

The precision statistics are significantly different from the overall results, the negative rate is based on the classifiers results for a single metric. This is demonstrated by the top two results. The decrease in bounds for the simple classifier from 6 to 4 increased the false negative rate while reducing the false positive rate. The smaller bounds classified 63 more true examples as false, however 245 more negative examples were correctly classified, improving the MCC rating by `0.09`. The simple classifier with deviations of two did not make the top 15. This is the trade-off we expect to see from altering the bounds of a classifier. This is the opposite from Table 5.14 where negative results are influenced by all dynamic properties with a bias towards the negative.

5.6.1 Java references

This section presents the results for the Java references test program, showing overall results, aggregate results for different metrics and graphs for interesting metrics. Table 5.16 shows the overall results for the Java references test program. There are six experiments for the Java references test-program, with a test set size of 50 there will be 300 positive and negative classifications, 600 total. The table contains the same bias towards the negative as the overall results presented in Table 5.14, however there are some false positives as the rankings decrease. The best performing classifier is the Kalman method taking the top two spots with `0.97` and `0.98`. The Kalman classifier also occupies seven of the top 15 places. The simple classifier comes in 3rd to 5th. The particle classifier comes in 8th. All of the results in top five are relatively close together with a spread of `0.06`.



(a) Training size 10

(b) Training size 20

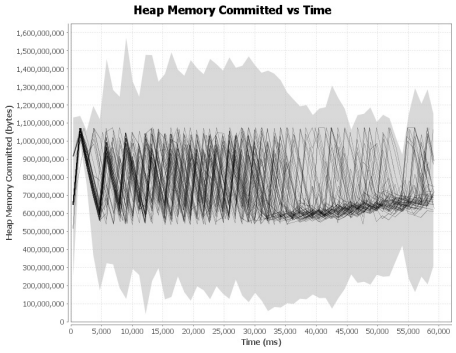
(c) Training size 40

(d) Training size 80

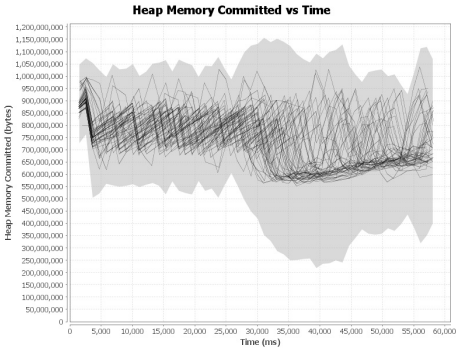
Figure 5.1: Affect of training-set size on data and generated bounds. Graphs show model generated using the same test-set but with varying test set sizes. The shaded area on the graph is the bounds, the darker lines are an instance of a training run.

Table 5.15: Heap-memory-used rankings aggregated across all test programs.

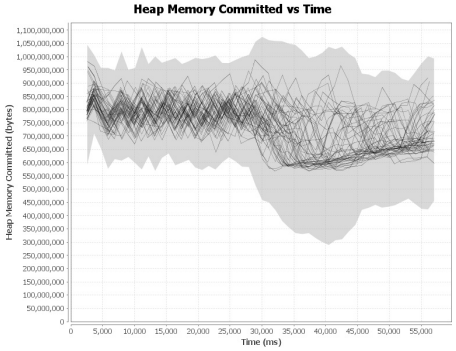
Config	TP	FN	TN	FP	Accuracy %	Precision %	MCC
S - 4	1477	123	1355	245	0.89	0.86	0.77
S - 6	1540	60	1115	485	0.83	0.76	0.68
P - 8	1552	48	1076	524	0.82	0.75	0.67
P - 6	1501	99	1126	474	0.82	0.76	0.66
S - 8	1558	42	979	621	0.79	0.72	0.63
P - 10	1564	36	967	633	0.79	0.71	0.63
K - 10	1552	48	960	640	0.79	0.71	0.61
P - 12	1575	25	916	684	0.78	0.7	0.61
K - 4	1105	495	1443	157	0.8	0.88	0.61
K - 6	1370	230	1190	410	0.8	0.77	0.6
P - 14	1584	16	874	726	0.77	0.69	0.6
K - 8	1454	146	1065	535	0.79	0.73	0.59
K - 12	1596	4	826	774	0.76	0.67	0.59
S - 10	1566	34	866	734	0.76	0.68	0.58
K - 14	1600	0	784	816	0.75	0.66	0.57



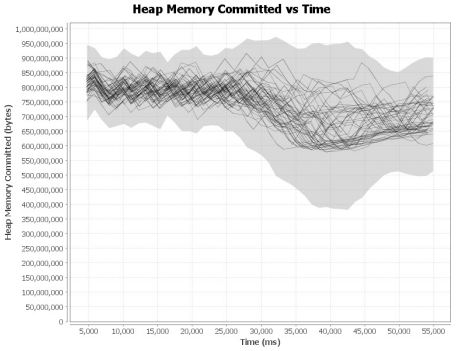
(a) No smoothing



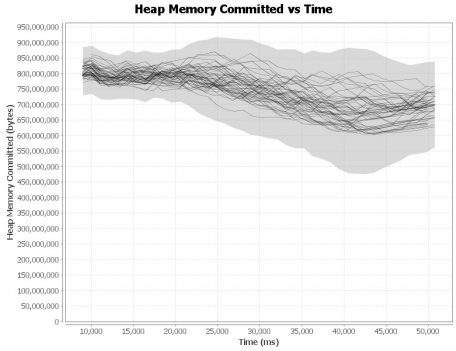
(b) Window size 3



(c) Window size 5



(d) Window size 9



(e) Window size 17

Figure 5.2: Affect of smoothing size on data and generated bounds. Figure (a) shows the raw training data, the other figures are smoothed using the mean smoother and a range of window-sizes.

5.6.2 Eden Space Memory used

The first dynamic-property presented for the Java references test program is Eden space memory used. The Eden space memory pool is where most objects are initially created, however there are a few exceptions, like if the object is too large.

The rankings for the Eden space memory used for the Java references test program is presented in Table 5.17. The best performing classifier is the simple method with 4 deviations, edging out the Kalman classifier by `0.01` MCC rating. The particle classifier with 6 deviations comes in 3rd with an MCC rating of `0.58`. Smaller bounds perform better on this particular example and simple classifier with 2 deviations made it into the top 15 results.

The trade off between deviations is illustrated in the middle of the table with the simple classifier. Using 6 deviations provides an MCC of `0.46`, increasing to 8 deviations causes the rating to drop by `0.06` points. Decreasing the bounds to 4 produces the best classifier with an MCC of `0.65`, decreasing even more to 2 deviations reduces the rating with an MCC of `0.40`

An example for the Eden space memory committed is now provided. This example classifies the weak object manager against the strong object manager. The classifier used in this example is the simple method with 4 deviations.

Figure 5.3 shows 4 graphs of the Eden space used for the Java-references test program. The training data from the strong object manager is illustrated in 5.3(a), and the training data against model in Figure 5.3. The positive and negative data sets are shown against the model in 5.3(c) and 5.3(d). The shaded area on the graphs is the model. This example would be difficult to differentiate using traditional approaches to identifying changes in resource utilisation. Both data sets have the roughly the same minimum and maximum values, but differ in the distribution of the training runs within those values. The model fitted in 5.3(a) is relatively tight with all training runs and positive data falling within the shaded area. The negative data is clearly different from the training-set with test-runs falling outside the bounds consistently over the entire graph.

The results for the experiment are presented in Table 5.18. A total of 46 of the positive instances were classified as positive and all of the negative instances classified as negative. The 4 positive instances classified as negative fell outside the range of last bounds.

Table 5.16: Java references test-program overall rankings. Top 10 of 30 results shown.

Config	TP	FN	TN	FP	Accuracy %	Precision %	MCC
K - 8	290	10	300	0	98	100	0.97
K - 6	288	12	300	0	98	100	0.96
S - 6	280	20	300	0	97	100	0.94
S - 8	282	18	296	4	96	99	0.93
S - 4	272	28	300	0	95	100	0.91
K - 10	298	2	258	42	93	88	0.86
K - 4	242	58	300	0	90	100	0.82
P - 12	279	21	267	33	91	89	0.82
P - 14	284	16	254	46	90	86	0.8
S - 10	282	18	252	48	89	85	0.78

Table 5.17: Eden space memory used rankings for Java references test program.

Config	TP	FN	TN	FP	Accuracy %	Precision %	MCC
S - 4	274	26	219	81	82	77	0.65
K - 4	266	34	223	77	82	78	0.64
P - 6	275	25	192	108	78	72	0.58
K - 6	292	8	159	141	75	67	0.56
K - 8	294	6	126	174	70	63	0.48
P - 8	278	22	146	154	71	64	0.46
S - 6	278	22	145	155	71	64	0.46
S - 8	278	22	124	176	67	61	0.4
S - 2	102	198	286	14	65	88	0.37
K - 10	296	4	81	219	63	57	0.37

Table 5.18: Java reference types Eden space memory used example result.

	Positive Data	Negative Data
Passed	46	0
Failed	4	50

5.6.3 Heap Memory Committed

The next dynamic-property for the Java reference example is the heap memory committed. The example shows the changes in heap memory consumption when classifying the soft object manager against the strong object manager.

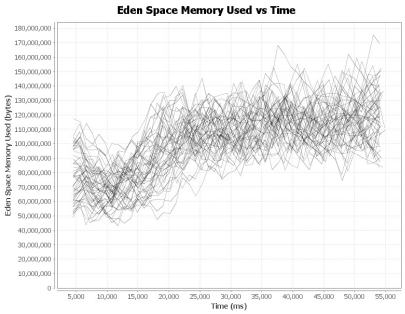
The overall rankings for heap memory committed are presented in Table 5.19. The Kalman classifier using 8 deviations was the best performing configuration with an MCC rating of `0.97`. There is not a lot of deviation between the top 5 classifiers with regards to accuracy and almost none with precision, after that the results begin to degraded.

Table 5.19: Heap-memory committed rankings for Java reference test program.

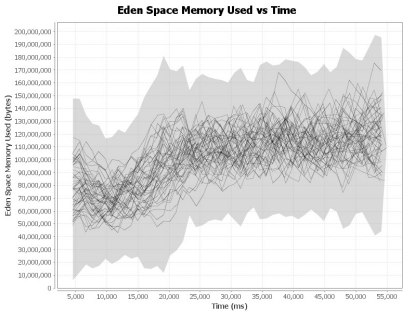
Config	TP	FN	TN	FP	Accuracy %	Precision %	MCC
K - 8	290	10	300	0	98	100	0.97
K - 6	288	12	300	0	98	100	0.96
S - 6	280	20	300	0	97	100	0.94
S - 8	282	18	296	4	96	99	0.93
S - 4	272	28	300	0	95	100	0.91
K - 10	298	2	258	42	93	88	0.86
K - 4	242	58	300	0	90	100	0.82
P - 12	279	21	267	33	91	89	0.82
P - 14	284	16	254	46	90	86	0.8
S - 10	282	18	252	48	89	85	0.78

The experiment shown for the heap memory committed classifies the soft object manager against the strong object manager. This example shows the Kalman filter with deviations of 8.

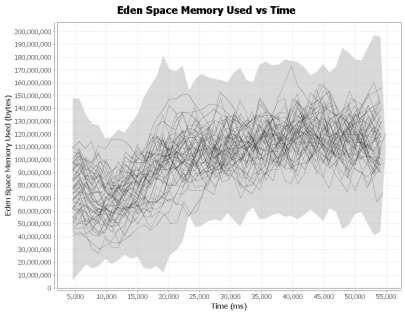
The graphs for the example are presented in Figure 5.4. The training data shows a wave like pattern oscillating between `500,000mbs` and `850,000mbs`.



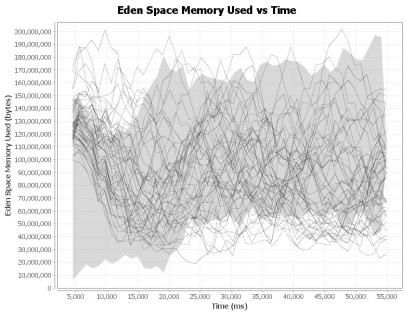
(a) Training data



(b) Training data vs model



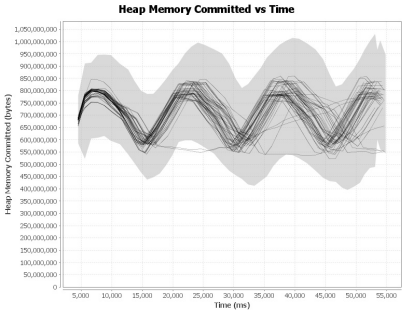
(c) Positive data vs model



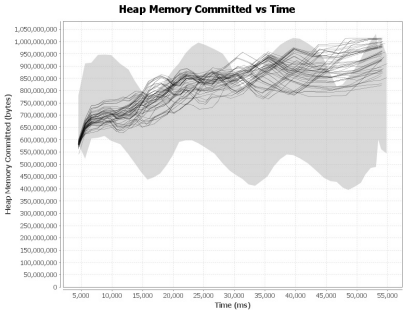
(d) Negative data vs model

Figure 5.3: Eden-space memory used graphs for Java reference types. Shaded area represents the model. Raw training data is show in (a), the model and training data in (b). The positive and negative data against the model in (c) and (d).

Negative data shows a steady increase from `550,000mbs` to `950,000mbs` directly through the wave pattern of the training data. The fitted model follows the wave like pattern of the training data. The positive data is mostly similar to training-set, there is an outlier which deviates from the model. The negative data initially falls within the bounds of the model, but when the oscillation decreases the data falls outside the model. When the oscillation increase the runs begin to fall back inside the model. This example requires good bounds to be fitted for the classification to work.



(a) Positive data vs model



(b) Negative data vs model

Figure 5.4: Heap-memory committed graphs for motivation Java reference types.

The classification results for the experiment are presented in Table 5.20. Five of the positive test instances were classified as negative due to a few outliers that deviated from the wave pattern. All of the negative examples fell outside of the generated bounds at some point and were classified as negative.

Table 5.20: Java reference types heap-memory-used example result.

	Positive Data	Negative Data
Passed	45	0
Failed	5	50

5.6.4 Object leaker

This section contains the results for the object leaker example, showing overall results and interesting metrics. The object leaker simulates a memory leak by keeping references to generated objects. It aims to affect the heap memory used, the different memory pools and the two garbage collectors.

Table 5.21 shows the overall results for the object leaker test program. There are six experiments for the object leaker test-program, with a test set size of 50 there will be 300 positive and negative classifications, 600 total. The table is dominated by the simple classifier with all but one configuration in the top 10, the only other classifier to get in was the Kalman with 20 deviations. This table shows a patters similar to the overall results table (Table 5.14) with a bias towards negative classification. Also large bounds performs the best due to there being no consequence for a false negative if other properties of the test run are classified as false. This is a common theme amongst the aggregated results.

Table 5.21: Object leaker test-program overall rankings.

Config	TP	FN	TN	FP	Accuracy %	Precision %	MCC
S - 20	282	18	300	0	97	100	0.94
S - 18	280	20	300	0	97	100	0.94
S - 16	276	24	300	0	96	100	0.92
S - 14	274	26	300	0	96	100	0.92
S - 12	268	32	300	0	95	100	0.9
S - 10	266	34	300	0	94	100	0.89
S - 8	260	40	300	0	93	100	0.87
S - 6	250	50	300	0	92	100	0.85
S - 4	234	66	300	0	89	100	0.8
K - 20	192	108	300	0	82	100	0.69

5.6.5 Scavenge garbage collector collection count

The first dynamic-property investigated for the object leaker is the scavenge garbage collector collection count. This experiment classifies the string builder variant against the string variant. Both examples generate strings, one uses string concatenation (a resource intensive process) while the other uses a string builder which is designed to address the weaknesses of string concatenation. The scavenge garbage collector is responsible for collecting objects in the Eden / survivor spaces and promoting tenured objects to the old generation (Oracle Corporation, 2013f).

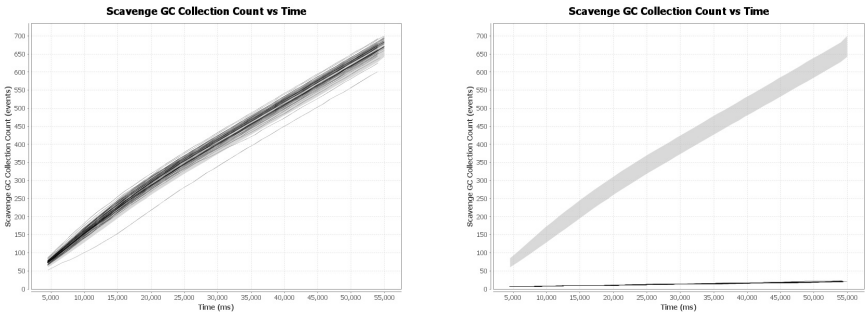
Table 5.22 shows the overall rankings for the scavenge GC collection count property in the object leaker test program. All classifiers in the top 10 performed

similarly with only a couple of false negatives, however after the top 10 the accuracy and precision degrades. The Kalman classifier dominates the top 10 with 8 of the best results. However there is only a difference of few false negatives separating all of the classifiers in the top 10. This is because the scavenge GC collection data sets are obviously different from each other making them simple to classify.

Table 5.22: Scavenge GC collection count rankings for object leaker test program.

Config	TP	FN	TN	FP	Accuracy %	Precision %	MCC
K - 12	298	2	300	0	100	100	0.99
K - 14	298	2	300	0	100	100	0.99
K - 18	298	2	300	0	100	100	0.99
K - 16	298	2	300	0	100	100	0.99
K - 20	298	2	300	0	100	100	0.99
K - 6	298	2	300	0	100	100	0.99
K - 8	298	2	300	0	100	100	0.99
K - 10	298	2	300	0	100	100	0.99
S - 6	290	10	300	0	98	100	0.97
S - 8	290	10	300	0	98	100	0.97

Table 5.5 shows the graphs produced for this experiment. The classifier used in this example is the simple method with deviations of 2. We expect the string example to have a higher rate of scavenge garbage collector activity, this because each string concatenation requires a new string to be built (strings are immutable in Java) leaving a lot of objects in the Eden and survivor spaces to be garbage collected. The results show the string example has significantly more scavenge garbage collector activity than the string builder example. Both examples have a consistent rate of collection events, just the string example has a significantly faster rate starting at a higher value. This example is partly responsible for the good precision rate of the classifiers.



(a) Positive data vs model (b) Negative data vs model

Figure 5.5: Scavenge GC collection count graphs for object leaker test-program.

The results for the classification presented in Figure 5.5 are outlined in Table 5.23. Seven of the positive data was miss-classified as negative and none of the negative data was misclassified. The bounds generated by the classifier was very small but so was the variance of the training data.

Table 5.23: Object leaker scavenge GC collection count experiment result.

	Positive Data	Negative Data
Passed	43	0
Failed	7	50

5.6.6 Marksweep garbage collector collection count

This example shows the same two data sets as the previous example, except the string builder is used as the training data set and the string example is used as negative data. The previous example looked at the scavenge garbage collector, this time the Marksweep garbage collector is looked at. The Marksweep garbage collector is responsible for collecting objects in the old generation memory pool (Oracle Corporation, 2013f). The classifier used in this example is the particle method with 4 deviations.

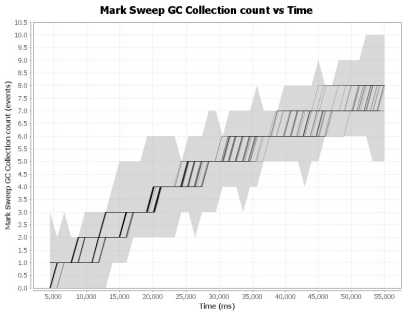
Table 5.24 shows the rankings for the Marksweep GC collection count for the object leaker test program. The results are very similar to the scavenge GC collection time results (5.22), the Kalman filter dominates the top of the rankings board and the top results are only differentiated by a few false negatives. This is to be expected, because, like the scavenge GC collection time results there is significant difference between all the data sets.

Table 5.24: Marksweep GC collection count rankings for object leaker test-program.

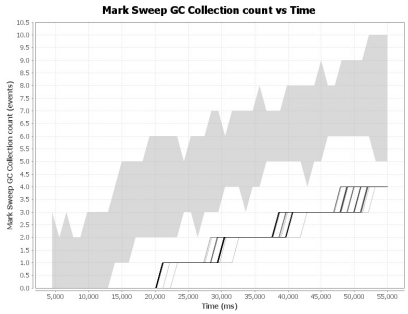
Config	TP	FN	TN	FP	Accuracy %	Precision %	MCC
K - 6	300	0	300	0	100	100	1
K - 2	300	0	300	0	100	100	1
K - 4	300	0	300	0	100	100	1
K - 8	300	0	300	0	100	100	1
K - 10	300	0	300	0	100	100	1
K - 12	300	0	300	0	100	100	1
K - 14	300	0	300	0	100	100	1
K - 18	300	0	300	0	100	100	1
K - 16	300	0	300	0	100	100	1
S - 2	292	8	300	0	99	100	0.97
S - 4	292	8	300	0	99	100	0.97

Table 5.6 shows the graphs produced for this experiment. The Marksweep collector count is expected to increase at a faster rate for the string builder example. This is because temporary buffers for strings used by the string builder often survive collections in the Eden space and make it to the old generation where they are collected when not used. The example shows this by the model and positive data clearly increasing faster than the negative data. Characteristics of the particle filter can be seen in the model with a slight lag in changes of value between the data and the model.

The results for the experiment are presented in Table 5.25. The two data sets are distinctly different and the classifier correctly classified all test instances.



(a) Positive data vs model



(b) Negative data vs model

Figure 5.6: Marksweep GC collection count graphs for object leaker test-program.

Table 5.25: Object leaker Marksweep GC collection count experiment result.

	Positive Data	Negative Data
Passed	50	0
Failed	0	50

5.6.7 Class loading

This section contains the results for the class-loading example, showing overall results and interesting metrics. The class-loading example leaks classes and class-loaders by loading multiple instances of a class and leaking objects of that type. The example was designed to affect the non-heap memory, compilation and class loading aspects of the JVM.

Table 5.26 shows the overall results for the class loading test program. There are six experiments for the classloading test-program, with a test set size of 50 there will be 300 positive and negative classifications, 600 total. The best performing configuration is the simple classifier with 16 deviations. The simple classifier is the most common method in the top 10 results, three instances of the particle classifier made it into the top 10. This table is affected by the negative bias of the overall classifier with no false positives identified by any of the classifiers. The accuracy of all the top classifiers is very high with all configurations with an MCC rating above `0.86` and the three top configurations at `0.99`.

Table 5.26: Class-loading overall rankings.

Classifier	TP	FN	TN	FP	Accuracy %	Precision %	MCC
S - 16	298	2	300	0	100	100	0.99
S - 20	298	2	300	0	100	100	0.99
S - 18	298	2	300	0	100	100	0.99
S - 14	296	4	300	0	99	100	0.99
P - 14	294	6	300	0	99	100	0.98
S - 12	290	10	300	0	98	100	0.97
S - 10	276	24	300	0	96	100	0.92
S - 8	270	30	300	0	95	100	0.9
P - 8	268	32	300	0	95	100	0.9
P - 6	256	44	300	0	93	100	0.86

5.6.8 Compilation Time

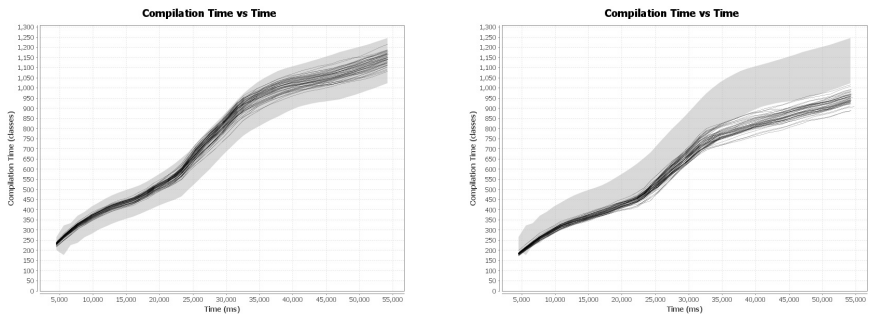
This section explores the compilation time dynamic-property for the class-loading example. The compilation time property measures the approximate elapsed time (in milliseconds) spent in compilation. The examples used load classes at different rates and thus should spend different amounts of time in compilation.

The rankings for the compilation time dynamic-property is outlined in Table 5.27. Each of the classifiers are present in the top four with the Kalman classifier occupying the top two positions. Each of the classifiers are evenly represented in the top 10, with the particle filter being the most common. The data shows good degradation of the precision and accuracy as the rankings decrease, this indicates the data sets are relatively close together, but still far enough to differentiate between them.

Table 5.27: Compilation time rankings for class-loading test program.

Config	TP	FN	TN	FP	Accuracy %	Precision %	MCC
K - 8	300	0	300	0	100	100	1
K - 6	300	0	300	0	100	100	1
S - 6	300	0	300	0	100	100	1
P - 6	300	0	300	0	100	100	1
P - 4	298	2	300	0	100	100	0.99
S - 4	298	2	300	0	100	100	0.99
P - 8	300	0	297	3	100	99	0.99
S - 8	300	0	295	5	99	98	0.98
K - 10	300	0	282	18	97	94	0.94
P - 10	300	0	255	45	93	87	0.86

The graphs for a single experiment are displayed in Figure 5.7. The dataset used to training the classifier generates leaks every `100ms` while the negative dataset leaks at a slower rate, every `1000ms`. The classifier used for the experiment was the Kalman method with deviations of 8. The two datasets look very similar except the positive data is consistently higher than the negative dataset, which is to be expected. This explains the pattern seen in the compilation time rankings shown in Table 5.27, because the two data are definitely different but still are close enough together to generate false positives when the bounds are large.



(a) Positive data vs model

(b) Negative data vs model

Figure 5.7: Compilation time graphs for class-loading test-program.

The results for the experiments are presented Table 5.28. All of the test instances were correctly classified by the Kalman filer, this is good because the test data was similar but still difficult to differentiate.

Table 5.28: Class-loading compilation time experiment result.

	Positive Data	Negative Data
Passed	50	0
Failed	0	50

5.6.9 Permanent generation memory used

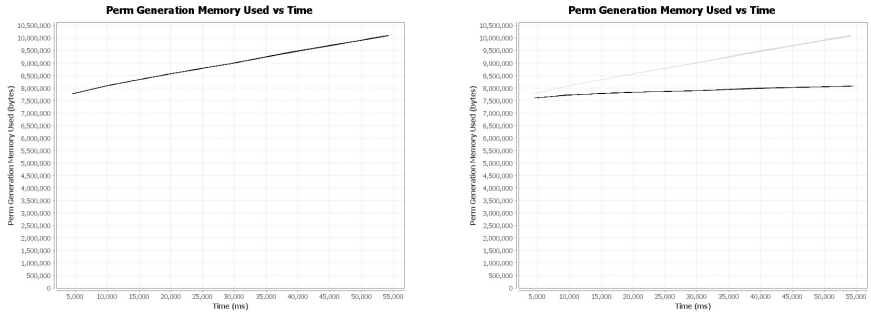
Here an example for the permanent generation memory used dynamic-property is presented. The permanent generation memory pool contains all the reflective data of the virtual machine such as class definitions and method objects. The example leaks classes, which has a direct affect of the utilisation of the permanent generation memory pool.

The results for the permanent generation memory used dynamic-property for the class-loading test program are presented in Table 5.29. All of the classifiers in the top 10 correctly classified all instances, the simple classifier and particle classifier only begins to degrade when the deviations gets below 6. All of the instances of the Kalman filter performed poorly in this situation, with the best ranking configuration having an MCC of `0.71`.

Table 5.29: Permanent generation memory used rankings for class-loading test-program.

Config	TP	FN	TN	FP	Accuracy %	Precision %	MCC
P - 16	300	0	300	0	100	100	1
P - 18	300	0	300	0	100	100	1
P - 20	300	0	300	0	100	100	1
P - 10	300	0	300	0	100	100	1
P - 12	300	0	300	0	100	100	1
P - 14	300	0	300	0	100	100	1
P - 8	300	0	300	0	100	100	1
P - 6	300	0	300	0	100	100	1
S - 6	300	0	300	0	100	100	1
S - 8	300	0	300	0	100	100	1

The experiment is the same as the one in the previous section, except the permanent generation memory pool is illustrated. The classifier that generated the model in Figure 5.8 is the particle classifier with 8 deviations. The training dataset leaks classes at a faster rate than the negative data set, so the permanent generation memory used should leak faster than negative data set. Graphs for the experiment are presented in Figure 5.8. The positive data (5.8(a)) shows a steady increase of the course of the test and not a lot of variance in the data set. The negative data (5.8(b)) shows the same pattern but increasing at slower rate, which is to be expected.



(a) Positive data vs model (b) Negative data vs model

Figure 5.8: Permanent generation used graphs for class loading test program.

The particle with 8 deviations correctly classified all test instances, but this is to be expected with such glaringly different data sets. Experiment results are presented in Table 5.30.

Table 5.30: Class-loading permanent generation used experiment result.

	Positive Data	Negative Data
Passed	50	0
Failed	0	50

5.6.10 Consumer Producer

This section contains the results for the consumer producer example, showing overall results and interesting metrics. The consumer producer example simulates a multi-threaded system, a set of producers generate items for the consumers to used, a shared buffer is used to control production. It is the most complex test-program investigated in this thesis and has some interesting results. Altering the parameters of the test-program should affect the memory utilisation, thread count, and throughput of the example. The two custom metrics, buffer-size and processed job count are explored.

Table 5.31 shows the overall results for the consumer producer test program. There are 14 experiments for the Java references test-program, with a test set size of 50 there will be 700 positive and negative classifications, 1400 total. The simple classifier with 4 deviations is the best performing configuration with an MCC of `0.95`. The simple classifier dominates the rest of the top 10 taking nine of the places, the Kalman filter is the only other method to make it into the top 10, with deviations of 20.

5.6.11 Process job count

The processed job count dynamic-property is a measurement of the throughput of the system. The higher the throughput of the example the better the performance is considered to be. Concurrency can be used to improve the throughput by executing non-dependant tasks in parallel, however concurrency does incur overhead due to scheduling threads and context switching.

Table 5.31: Consumer producer test-program overall rankings.

Config	TP	FN	TN	FP	Accuracy %	Precision %	MCC
S - 4	670	30	694	6	0.97	0.99	0.95
S - 10	698	2	650	50	0.96	0.93	0.93
S - 12	698	2	650	50	0.96	0.93	0.93
S - 14	698	2	650	50	0.96	0.93	0.93
S - 16	698	2	650	50	0.96	0.93	0.93
S - 18	698	2	650	50	0.96	0.93	0.93
S - 20	698	2	650	50	0.96	0.93	0.93
S - 8	693	7	650	50	0.96	0.93	0.92
S - 6	690	10	650	50	0.96	0.93	0.92
K - 20	539	161	650	50	0.85	0.92	0.71

Table 5.32 presents the aggregated processed job count for the consumer producer test program. The top seven classifiers all received an MCC of `1`, with all classifiers represented with deviations between 4 to 8. All the classifiers in the top 10 use smaller deviations, with the Kalman filter at 10 deviations the exception, two configurations with deviations of 2 made the top 10. After the top 10 the MCC ratings of the classifiers drop off dramatically with the configurations with larger deviations towards the end of the table.

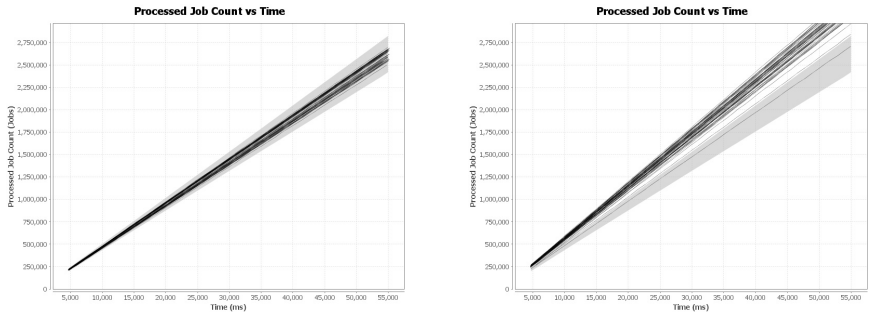
Table 5.32: Processed job count rankings for consumer producer test-program.

Config	TP	FN	TN	FP	Accuracy %	Precision %	MCC
P - 4	700	0	700	0	100	100	1
P - 6	700	0	700	0	100	100	1
S - 4	700	0	700	0	100	100	1
S - 6	700	0	700	0	100	100	1
K - 6	700	0	700	0	100	100	1
K - 8	700	0	700	0	100	100	1
K - 10	700	0	700	0	100	100	1
P - 8	700	0	644	56	96	92	0.91
P - 2	623	77	100	0	95	100	0.9
S - 2	623	77	100	0	95	100	0.9

The experiment investigated for the processed job count demonstrates why the configurations with the larger bounds performed poorly. The training oracle uses 1000 producers, 1000 consumers and a buffer size of 10 while the test oracle uses 10 producers, 10 consumers and a buffer-size of 10. The machine the test was run on was a quad core and is unable to take advantage of a large amount threads, so the overhead introduced by considerably more threads should reduce the throughput of the example and decrease the processed job count.

Graphs for the experiment are presented in Figure 5.9. The bounds were generated using the simple classifier with deviations of 4. All the instances in positive data all correctly fit within the model. The negative dataset is consistently higher than the model, which is expected, however there is one outlier that falls within the bounds of the model. The two data sets are relatively close together but still

are unique enough to differentiate between the two sets, this is why the smaller deviation configurations performed better for the overall rankings.



(a) Positive data vs model (b) Negative data vs model

Figure 5.9: Processed job count graphs for consumer producer test-program.

The results for the experiments are presented in Table 5.9. All the positive instances were correctly classified and only one negative instance was incorrectly classified as positive. This instance can be seen in Figure 5.9(b), there is one outlier in the negative data set that goes directly through the model. Again this example demonstrates the non-deterministic nature of the JVM.

Table 5.33: Consumer producer processed job count example result.

	Positive Data	Negative Data
Passed	50	1
Failed	0	49

5.6.12 Buffer size

The buffer-size dynamic-property is a custom property that monitors how full the shared buffer is in the example. The buffer-size is determined by the production and consumption rates, which is controlled by the number of producers and consumers.

Overall results for the buffer-size dynamic-property in the consumer producer test-program are presented in Table 5.34. The rankings indicate the data-sets in this example are difficult to differentiate, with the classifiers degrading to a neutral MCC after the top six. This is caused by false positives which occur when negative instances are classed as positive. The best performing classifier is the simple method using two deviations, with a MCC rating of `0.82`. Both the Kalman and particle classifiers with two deviations take the 2nd and 3rd positions. The best performing classifiers use small deviations and have a low false positive rate.

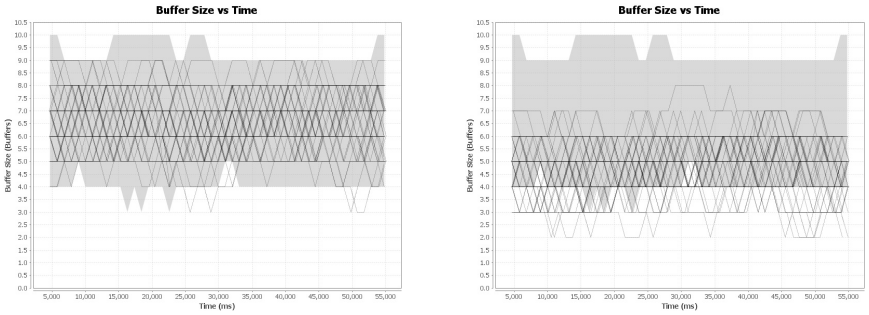
The experiment used, illustrates the buffer-size dynamic-property and compares the use of different producer counts. The training data-set uses 1000 producers while the test dataset uses 10. A higher producer count should yield a faster rate at which objects are generated and keep the buffer fuller.

The graphs produced for the experiment are outlined in Figure 5.10. The classifier used to generate the model is the simple method with 2 deviations. The positive data shows a steady pattern of random oscillation between 4 and 10 items. The model fitted is very tight to the positive data set and there are a few instances

Table 5.34: Buffer size rankings for consumer producer test-program.

Config	TP	FN	TN	FP	Accuracy %	Precision %	MCC
S - 2	523	77	651	49	91	93	0.82
K - 2	539	161	686	14	88	97	0.77
P - 2	686	14	427	273	80	72	0.64
K - 4	700	0	196	504	64	58	0.4
S - 4	700	0	42	658	53	52	0.18
K - 6	700	0	7	693	51	50	0.07
P - 16	700	0	0	700	50	50	0
P - 18	700	0	0	700	50	50	0
P - 20	700	0	0	700	50	50	0
P - 12	700	0	0	700	50	50	0

that fall below the lower bounds of the model. Both of the datasets are very similar with similar distributions and patterns, however the average of the negative dataset is slightly less. Using large deviations will generate models that envelope both data sets and negative instances will be falsely classified as positive. The data sets are different enough to identify the change resource utilisation, but require a good model to be fitted.



(a) Positive data vs model

(b) Negative data vs model

Figure 5.10: Buffer-size graphs for consumer producer test-program.

Results for the buffer-size experiment are presented in Table 5.35. Five of the positive instances were classified as negative and the five of the negative instances classified as positive. This is because the two data sets are very similar to each other and the overlapping instances were misclassified.

Table 5.35: Consumer producer buffer size experiment result.

	Positive Data	Negative Data
Passed	45	5
Failed	5	45

5.7 Notification classifier

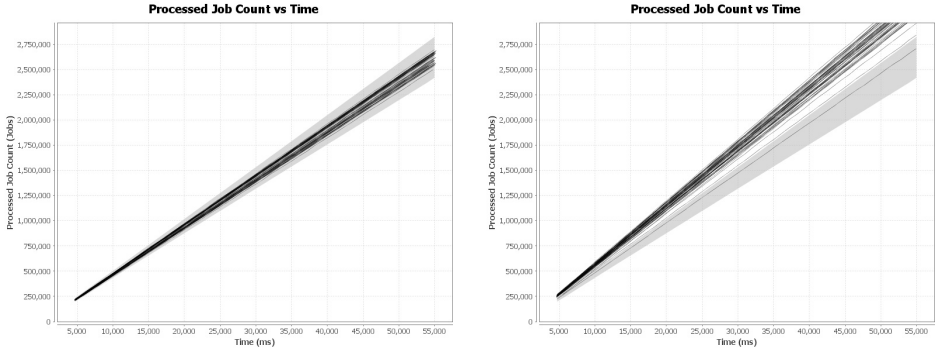
The notification classifier is responsible for building a model for the notification emissions of performance test and testing test-runs against models. There are

no notifications emitted by the the platform MXBeans distributed with the JRE. However the consumer producer test program was designed to emit notifications when the processed job count passes a set log interval. These notifications are used to test the correctness of the notifications classifier.

In the consumer producer test-program the rate at which notifications are emitted is directly proportional to the processed job count. To illustrate the effectiveness of the notification classifier two experiments are analysed here. One with similar processed job counts and one with different processed job counts. This will show how the classifier reacts in both positive and negative situations. Both examples are generated using the simple classifier (mean average type, 4 deviations), mean smoothing window size 9 and linear time-point generation.

5.7.1 Different example

For the different example the training dataset used in the experiment used 1000 producers, 1000 consumers, with a buffer-size of 10. The negative data used 10 producers, 10 consumers, and a buffer-size of 10. The graphs generated from the experiment are shown in Figure 5.11. The data sets show different rates of processing jobs, this is due to the negative dataset having lower overhead due to smaller thread count. An increase in the processed job count should increase the rate at which notifications are emitted by the job counter.



(a) Positive data vs model (b) Negative data vs model

Figure 5.11: Notification classifier processed job count for consumer producer example different processed job rate.

Table 5.36 shows the positive notifications data being tested against the model. Each number represents a training run and how many notifications were flagged as erroneous. Most results are 0 or 1, with a few higher and a definite outlier at 28 misclassified notifications.

Table 5.37 shows the negative notifications data being tested against the model. There is a significant difference between the two data sets, with most of the test-runs misclassifying 33 to 35 notifications. There is one outlier which only misclassified 1 instance, this can be seen as the negative run that goes through bounds in Figure 5.11(b).

The distribution for both the negative and positive data-sets are presented in Table 5.38. Both data sets have a similar distribution with the negative example having a larger standard deviation. The positive dataset has considerably lower misclassification rate with an average of `1.22` compared to `33.44`.

Table 5.36: Positive data classification results from notification classifier different example. Each value is the number of negatively classified notifications from each test-run.

1	0	1	0	4	0	1	1	1	1
0	1	0	0	0	0	1	0	1	0
1	0	1	1	0	0	0	1	0	1
1	1	28	0	1	2	1	1	1	2
0	0	1	0	1	1	1	0	0	1

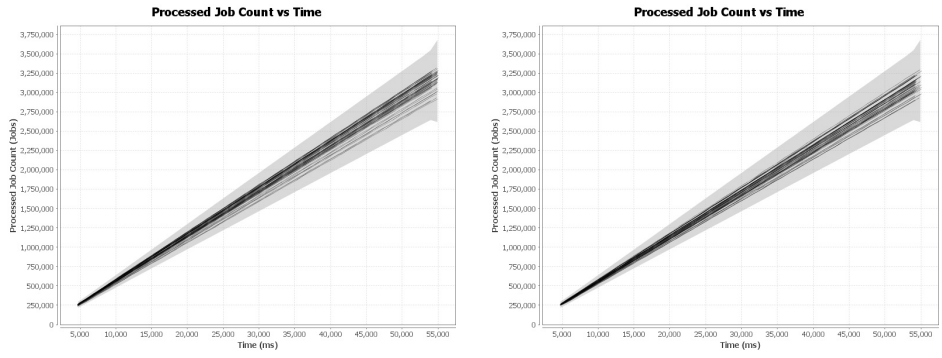
Table 5.37: Notification classifier negative data results from different example.Each value is the number of negatively classified notifications from each test-run.

33	34	34	34	36	1	32	33	35	33
35	35	34	33	35	34	34	35	33	35
34	34	33	33	31	34	35	35	33	33
35	35	35	36	34	34	33	34	35	34
34	34	35	35	34	34	36	33	35	34

5.7.2 Similar example

The similar example compares 10 producers, 10 consumers and a buffer size of 10 against 10 producers, 10 consumers and a buffer-size of 1000. Because there is a balanced rate between consumption and production altering the buffer-size does not have a large affect on the throughput of the example. Notifications should be emitted at a similar rate and the classifier should not detect a significant difference between them.

The graphs for the experiments are presented in Figure 5.12. Both the negative and positive data sets look similar, however there is some variation between the two. Both data sets fit within the model, which was generated using the simple classifier with deviations of 4.



(a) Positive data vs model

(b) Negative data vs model

Figure 5.12: Processed job count for consumer producer example showing similar processed job rate.

Table 5.38: Notification classifier results for different notifications rate.

	Average	Stdev
Positive	1.22	3.93
Negative	33.44	4.79

Table 5.39: Positive data results from notification classifier similar example.

2	0	1	1	0	2	2	2	0	2
2	1	1	2	0	1	1	1	1	4
7	1	2	7	2	0	2	2	5	2
1	2	3	2	3	4	1	1	1	1
1	3	1	2	0	1	1	2	1	7

Notification classification results from the positive data set are presented in Table 5.39. Only a few of the instances were correctly classified with no differences, most instances had misclassified notifications with the average being around 2.

Table 5.40: Negative data results from notification classifier similar example.

2	3	3	1	0	3	1	2	2	1
2	1	2	5	3	3	3	2	2	3
3	2	4	1	1	2	5	1	7	3
2	7	3	2	2	2	2	1	5	2
1	1	1	2	1	3	3	7	3	1

Table 5.41: Notification classifier results for similar notifications rate.

	Average	Stdev
Positive	1.88	1.65
Negative	2.48	1.57

Results from the negative dataset are presented in Table 5.40. These results appear to slightly higher than the positive dataset with more misclassified notifications. Only one data fitted the model correctly, this is counter intuitive to the data shown in the graphs where the classifier could not distinguish between the two data sets.

The spread of the data is similar with standard deviations of `1.65` and `1.57`, however the average for the positive dataset is `0.6` lower than the negative dataset.

Chapter 6

Evaluation

This research has investigated the viability of resource utilisation regression testing, specifically for JVM based applications. To demonstrate this, a set of test programs and a set of analysis tools were developed and experiments conducted to investigate the robustness of the method. Section 1.4 outlines the research questions the thesis aimed to address.

Identify machine learning techniques that are suitable for generating robust statistical test oracles.

The first research question aimed to identify a set of techniques that are suitable for generating statistical oracles. A simple method was developed that used the distribution of training runs to calculate bounds. Two more complex solutions were also explored. These were the Kalman and particle filters, which are two methods commonly applies to time-series data. Both algorithms are used in tracking and predicting systems with noisy measurements. These methods were then implemented as classifiers in the Buto framework. Data pre-processing methods including smoothing, time-point generation were explored as well as two bounds matching methods.
The methods identified in this thesis work well and demonstrated good promise for use in real world situations.

Identify a good set of benchmarking scenarios that illustrate important runtime features.

The next research questions regards identifying test programs that are configurable to show different resource utilisation profiles. Four test programs were developed that exhibited common characteristics of a Java programs resource consumption. A set of experiments were developed based on these test program. Configurations of the test programs affected all but two of the metrics monitored. These benchmarking scenarios were not overly complicated but are sufficient for the purposes of this thesis.

The results of the classifiers is heavily dependant on the data sets used for evaluation. If datasets are glaringly different, like the object leaker GC collection count example (Figure 5.5), they are easy to classify and will not provide a lot of insight into how well a classification method performs. An example of good datasets is the compilation time graphs for the class loading example shown in

Figure 5.7. The negative data in Figure 5.7(b) is relatively similar to the model, but still different enough to differentiate. This forces the classifier to fit tight bounds to the training data, otherwise the false positive rate will increase. Improving the quality of the data sets used for evaluation will provide better insight into the robustness of the classification methods.

Determine the robustness of the machine learning techniques relative to the set of benchmarking scenarios.

The final research question involved assessing how robust the tool is relative to the chosen test programs. Overall, the best performing classifier was the simple classifier with an MCC `0.94` (Table 5.14). The worst performing metric presented was the Eden space memory used for the Java test program (Section 5.6.2), with the best classifier performing with a MCC rating of `0.65` However, there are examples where the Kalman classifier or the particle classifier outperforms the simple classifier. Analysis of the overall results showed a bias towards towards the negative. Investigating individual metrics provided better insight into the robustness of each classification method. The notification classification method showed showed promise and worked well on the examples used. Relative to the experiments conducted the tool shows good promise at identifying resource utilisation changes in performance tests.

This thesis aimed to test the notion of performing resource utilisation regression testing. A set of machine learning techniques were explored against a set of test programs. The results from the experiments showed the classifier can identify a range of changes in applications and shows good promise for use in software development process.

6.1 Future Work

This thesis presents a proof of concept for resource utilisation regression testing and has many ways in which to extend the concept. Improvements can be made by developing more classification methods, writing more complex test projects to test the framework against and integrating with other development tools to make the process easier to adopt by engineers. The ideas presented below were outside the scope of this project but were uncovered as potential ways to extend the project.

The immediate future work of this thesis is to turn the proof of concept into a usable framework for Java developers. The work done needs to be packaged in a way that engineers can download the tool, immediately begin using it and gain feedback about their software's resource utilisation. Mostly this will involve creating a usable API which is lightweight and easily configurable.

The obvious way to extend the project is by developing more classification methods. The methods investigated in this thesis treat each monitored property separately even though some are related to each other. For example, garbage collector data is highly dependant on the state of the heap. Knowledge of this dependency could used to detect anomalies between the two properties.

Further investigation of suitable test projects is key to future work. The test projects used in this thesis are simple by design, however, other larger and more complex test projects will be required for further evaluation. Also, more effort needs to be placed on designing experiments that use datasets that are similar

but still noticeably different A common use case for performance testing is web servers and developing test programs that demonstrate anomalies in web server performance should be considered.

For resource utilisation regression testing to be adopted as common practice there needs to be integration with other development tools. Version control is the obvious tool for integration, allowing engineers to design performance tests to be run after each commit so regressions can be identified as soon as they are added to the code base. Integration with version control will also allow engineers to track the runtime signatures of their programs as development progresses and useful insight into the performance of their program could be extracted. JUnit is the standard for testing practices in Java and integration with JUnit will increase the likelihood of adopting the practice. There have already been attempts at integrating benchmarking with JUnit through Perfidix (University of Konstanz, 2012) JUnitPerf (Clarkware Consulting, 2009) and ContiPerf (Databene, 2013). Automated build scripts are a common practice for engineers and integration with tools like Ant and Maven will allow engineers to execute performance tests using build scripts in the same way as unit tests.

More in depth reporting and visualisations of the data collected by Buto will allow for more insight to be gained by programmers. More in-depth reporting of detected anomalies can be developed and visualisations of where the program reaches abnormal states could be developed. An interesting idea that could be developed alongside integration with version control systems would be to visualise the programs development over time and see how the runtime of their code has changed as development has progressed.

A common practice of performance testing is testing the system under a workload. This is a common practice for teams developing servers or websites which need to process requests from an outside source. Developing a mechanism to simulate workloads for Buto would greatly benefit theses engineers. Engineers can design performance tests that stress the program in different ways and build baselines for their execution under different levels of load. A workload simulation component would be of great use to engineers and would also speed up the adoption of runtime performance testing. The JMX API allows for operations of an MBean to be invoked remotely, allowing a JMX client to affect a running performance test. Invoking operations can allow the test-runner to influence the test as it is running, this could be change used to change aspects of the VM at certain points in the test or drive a test's execution.

Heger et al. (2013) propose a technique for identifying the root causes of performance regressions in software. Their method involves version control integration and performance aware unit tests to trace performance regressions in the version control history to determine what code changes caused the regression in the software. This technique was tested on the Apache Commons Math library and was able to detect the code changes which caused a major performance regression. This is an interesting technique that could be used to extend the capabilities of Buto.

This methodology for assessing software performance could be used in automatic tuning of applications. Implementing a comparison classifier that could differentiate between better and worse resource utilisation profiles will allow configurations to be ranked. This can allow software to adapt to their runtime environment based on their current resource utilisation profile.

Java was chosen as the implementation language for Buto because of the al-

ready existing technologies that allowed for easier development of the proof of concept. But the concept can be applied to any other language that has a mechanism to collect data about the runtime of the application. Another obvious implementation would be using C and C++ languages and Windows Management Instrumentation (WMI) to monitor their execution. WMI is an operating system interface that allows instrumented components to provided information and notifications regarding their state. Oracle also provides and alternative implementation of the JVM known as JRockit (Oracle Corporation, 2013a) JVM, this implementation is designed for high performance and monitoring. The conception of resource utilisation regression testing can be applied to any domain where relevant performance counters are available.

This thesis is a proof of concept for resource utilisation regression testing and there are many extension points available. This area of software engineering is still underdeveloped but is gaining more attention. As engineers begin to adopt more performance testing methods these extension points are expected to be developed.

Bibliography

F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2499-0. doi: 10.1109/CGO.2006.37. URL http://dx.doi.org/10.1109/CGO.2006.37.

Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.*, 32 (5):85–96, May 1997. ISSN 0362-1340. doi: 10.1145/258916.258924. URL http://doi.acm.org/10.1145/258916.258924.

AppPerfect. Appperfect java profiler, 2013. URL http://www.appperfect.com/products/java-profiler-features.html. Accessed: 9-8-2013.

Davide Arcelli, Vittorio Cortellessa, and Catia Trubiani. Antipattern-based model refactoring for software performance improvement. In *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures*, QoSA '12, pages 33–42, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1346-9. doi: 10.1145/2304696.2304704. URL http://doi.acm.org/10.1145/2304696.2304704.

LB Arief and Neil A Speirs. A uml tool for an automatic generation of simulation programs. In *Proceedings of the 2nd international workshop on Software and performance*, pages 71–76. ACM, 2000.

M. Arnold and D. Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 51–62, 2005. doi: 10.1109/CGO.2005.9.

Vlastimil Babka, Petr Tuma, Pma, and Lubomir Bulej. Validating model-driven performance predictions on random software systems. In *Research into Practice–Reality and Gaps*, pages 3–19. Springer, 2010.

Jose Baiocchi, Bruce R. Childers, Jack W. Davidson, Jason D. Hiser, and Jonathan Misurda. Fragment cache management for dynamic binary translators in embedded systems with scratchpad. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '07, pages 75–84, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-826-8. doi: 10.1145/1289881.1289898. URL http://doi.acm.org/10.1145/1289881.1289898.

Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.

S. Balsamo, A. di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *Software Engineering, IEEE Transactions on*, 30(5):295–310, 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.9.

K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mallor, Ken Shwaber, and Jeff Sutherland. The agile manifesto. Technical report, The Agile Alliance, 2001.

Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004. ISBN 0321278658.

Marina Biberstein, Vugranam C. Sreedhar, Bilha Mendelson, Daniel Citron, and Alberto Giammaria. Instrumenting annotated programs. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 164–174, New York, NY, USA, 2005. ACM. ISBN 1-59593-047-7. doi: 10.1145/1064979.1065002. URL `http://doi.acm.org/10.1145/1064979.1065002`.

Walter Binder, Jarle Hulaas, and Philippe Moret. Advanced java bytecode instrumentation. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, PPPJ '07, pages 135–144, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-672-1. doi: 10.1145/1294325.1294344. URL `http://doi.acm.org/10.1145/1294325.1294344`.

Borland. Devpartner, 2013. URL `http://www.borland.com/products/devpartner/default.aspx`. Accessed: 19-8-2013.

Robert Goodell Brown. *Smoothing, forecasting and prediction of discrete time series*. Courier Dover Publications, 2004.

Michael G Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J Serrano, Vugranam C Sreedhar, Harini Srinivasan, and John Whaley. The jalapeno dynamic optimizing compiler for java. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141. ACM, 1999.

Yuhong Cai, John Grundy, and John Hosking. Synthesizing client load models for performance engineering via web crawling. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 353–362. ACM, 2007.

Pierre Caserta and Olivier Zendra. A tracing technique using dynamic bytecode instrumentation of java applications and libraries at basic block level. In *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '11, pages 6:1–6:4, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0894-6. doi: 10.1145/2069172.2069178. URL `http://doi.acm.org/10.1145/2069172.2069178`.

Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S Bharadwaj Yadavalli, and John Yates. A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, 1998.

Clarkware Consulting. Junitperf, 2009. URL `http://www.clarkware.com/software/JUnitPerf.html`. Accessed: 28-8-2013.

R Clint Whaley, Antoine Petitet, and Jack J Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1):3–35, 2001.

Guojing Cong, I-Hsin Chung, Hui-Fang Wen, David Klepacki, Hiroki Murata, Yasushi Negishi, and Takao Moriyama. A systematic approach toward automated performance analysis and tuning. *Parallel and Distributed Systems, IEEE Transactions on*, 23(3):426–435, 2012.

Vittorio Cortellessa, Antinisca Di Marco, Paola Inverardi, Fabio Mancinelli, and Patrizio Pelliccione. A framework for the integration of functional and non-functional analysis of software architectures. *Electronic Notes in Theoretical Computer Science*, 116:31–44, 2005.

Databene. Contiperf 2, June 2013. URL `http://databene.org/contiperf.html`. Accessed: 22-9-2013.

Markus Debusmann and Kurt Geihs. Efficient and transparent instrumentation of application components using an aspect-oriented approach. In *Self-Managing Distributed Systems*, pages 209–220. Springer, 2003.

Luiz DeRose and Heidi Poxon. A paradigm change: from performance monitoring to performance analysis. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD'09. 21st International Symposium on*, pages 119–126. IEEE, 2009.

Luiz DeRose, Bill Homer, Dean Johnson, Steve Kaufmann, and Heidi Poxon. Cray performance analysis tools. In *Tools for High Performance Computing*, pages 191–199. Springer, 2008.

Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 353–364, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. doi: 10.1145/1854273.1854318. URL `http://doi.acm.org/10.1145/1854273.1854318`.

Mikhail Dmitriev. Design of jfluid: A profiling technology and tool based on dynamic bytecode. In *Instrumentation, Sun Microsystems Technical Report, Mountain View, CA, available at http://research.sun.com*, 2004.

Adam Donlin. An analysis of the producer-consumer problem, June 2002. URL `http://www.dcs.ed.ac.uk/home/adamd/essays/ex1.html`.

D. Draheim, John Grundy, J. Hosking, C. Lutteroth, and G. Weber. Realistic load testing of web applications. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 11 pp.–70, 2006. doi: 10.1109/CSMR.2006.43.

Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *SIGOPS Oper. Syst. Rev.*, 34(5):202–211, November 2000. ISSN 0163-5980. doi: 10.1145/384264.379241. URL `http://doi.acm.org/10.1145/384264.379241`.

Eclipse Foundation. Eclipse memory analyzer, June 2013. URL `http://www.eclipse.org/mat/`. Accessed: 30-8-2013.

EJ Technologies. Jprofiler, 2012. URL `http://www.ej-technologies.com/products/jprofiler/overview.html`. Accessed: 28-9-2013.

Sean P Engelson and Drew V McDermott. Error correction in mobile robot map learning. In *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*, pages 2555–2560. IEEE, 1992.

H. Erdogmus, Maurizio Morisio, and Marco Torchiano. On the effectiveness of the test-first approach to programming. *Software Engineering, IEEE Transactions on*, 31(3):226–237, 2005. ISSN 0098-5589. doi: 10.1109/TSE.2005.37.

King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E Hassan, Ying Zou, and Parminder Flora. Mining performance regression testing repositories for automated performance analysis. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 32–41. IEEE, 2010.

Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun. Monte carlo localization: Efficient position estimation for mobile robots. *AAAI/IAAI*, 1999: 343–349, 1999.

Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007a.

Andy Georges, Dries Buytaert, and Lieven Eeckhout. Javastats, September 2007b. URL `http://www.elis.ugent.be/JavaStats`.

S. Ghaith, Miao Wang, P. Perry, and J. Murphy. Profile-based, load-independent anomaly detection and analysis in performance regression testing of software systems. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 379–383, 2013. doi: 10.1109/CSMR.2013.54.

Shadi Ghaith. Analysis of performance regression testing data by transaction profiles. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 370–373, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2159-4. doi: 10.1145/2483760.2492399. URL `http://doi.acm.org/10.1145/2483760.2492399`.

Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. *SIGPLAN Not.*, 39(4):49–57, April 2004. ISSN 0362-1340. doi: 10.1145/989393.989401. URL `http://doi.acm.org/10.1145/989393.989401`.

Edward James Hannan. *Multiple time series*, volume 38. Wiley. com, 2009.

Lee Rainie Harrison Rainie and Barry Wellman. *Networked: The New Social Operating System*. MIT Press, 2012.

Christoph Heger, Jens Happe, and Roozbeh Farahbod. Automated root cause isolation of performance regressions during software development. In *Proceedings of the ACM/SPEC international conference on International conference on performance engineering*, pages 27–38. ACM, 2013.

IBM Developer Works. Ibm heap analyzer - a graphical tool for discovering possible java heap leaks, Jun 2013. URL `https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=4544bafe-c7a2-455f-9d43-eb866ea60091`. Accessed: 21-9-2013.

Zhen Ming Jiang. Automated analysis of load testing results. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 143–146, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-823-0. doi: 10.1145/1831708.1831726. URL `http://doi.acm.org/10.1145/1831708.1831726`.

M.J. Johnson, E.M. Maximilien, Chih-Wei Ho, and L. Williams. Incorporating performance testing in test-driven development. *Software, IEEE*, 24(3):67–73, 2007. ISSN 0740-7459. doi: 10.1109/MS.2007.77.

Junit-team. Junit, August 2013. URL `http://junit.org/`. Accessed: 28-9-2013.

Rudolph Emil Kalman et al. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.

Thomas Karcher and Victor Pankratius. Run-time automatic performance tuning for multicore applications. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6852 of *Lecture Notes in Computer Science*, pages 3–14. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-23399-9. doi: 10.1007/978-3-642-23400-2_2. URL `http://dx.doi.org/10.1007/978-3-642-23400-2_2`.

Thomas Karcher, Christoph Schaefer, and Victor Pankratius. Auto-tuning support for manycore applications: perspectives for operating systems and compilers. *ACM SIGOPS Operating Systems Review*, 43(2):96–97, 2009.

Torsten Kempf, Kingshuk Karuri, and Lei Gao. *Software Instrumentation*. John Wiley & Sons, Inc., 2007. ISBN 9780470050118. doi: 10.1002/9780470050118.ecse386. URL `http://dx.doi.org/10.1002/9780470050118.ecse386`.

Marc Kramis, Alexander Onea, and Sebastian Graf. *PERFIDIX: a Generic Java Benchmarking Tool*. Bibliothek der Universität Konstanz, 2007.

Donghun Lee and Jong-Jin Park. Software performance monitoring using aggregated performance metrics by z-value. In *Convergence and Hybrid Information Technology*, pages 708–715. Springer, 2012.

Donghun Lee, Sang K Cha, and Arthur H Lee. A performance anomaly detection and analysis framework for dbms development. *Knowledge and Data Engineering, IEEE Transactions on*, 24(8):1345–1360, 2012.

C. Lutteroth and G. Weber. Modeling a realistic workload for performance testing. In *Enterprise Distributed Object Computing Conference, 2008. EDOC '08. 12th International IEEE*, pages 149–158, 2008. doi: 10.1109/EDOC.2008.40.

A.J. Maalej, M. Hamza, M. Krichen, and M. Jmaiel. Automated significant load testing for ws-bpel compositions. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 144–153, 2013. doi: 10.1109/ICSTW.2013.25.

Lech Madeyski. *Test-Driven Development - An Empirical Evaluation of Agile Practice*. Springer, 2010. ISBN 978-3-642-04287-4.

HAROON Malik. Automated analysis of load tests using performance counter logs. 2013.

Bronis R. de Supinski Martin Schulz. egprof: Extending gprof for comparative analysis, 2008. URL `https://computing.llnl.gov/code/egprof/`. Accessed: 21-9-2013.

Moreno Marzolla and Simonetta Balsamo. Uml-psi: The uml performance simulator. In *Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings. First International Conference on the*, pages 340–341. IEEE, 2004.

Paul E. McKenney. Differential profiling. *Software: Practice and Experience*, 29(3):219–234, 1999. ISSN 1097-024X. doi: 10.1002/(SICI)1097-024X(199903)29:3⟨219::AID-SPE230⟩3.0.CO;2-0. URL `http://dx.doi.org/10.1002/(SICI)1097-024X(199903)29:3<219::AID-SPE230>3.0.CO;2-0`.

Nathan Meyers. Jperfanal - a java performance analyzer, June 2013. URL `http://jperfanal.sourceforge.net/`. Accessed: 12-9-2013.

Ian Molyneaux. *The Art of Application Performance Testing - Help for Programmers and Quality Assurance.* O'Reilly, 2009. ISBN 978-0-596-52066-3.

Michael Montemerlo, Sebastian Thrun, and William Whittaker. Conditional particle filters for simultaneous mobile robot localization and people-tracking. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 1, pages 695–701. IEEE, 2002.

P. Moret, W. Binder, D. Ansaloni, and A. Villazon. Visualizing calling context profiles with ring charts. In *Visualizing Software for Understanding and Analysis, 2009. VISSOFT 2009. 5th IEEE International Workshop on*, pages 33–36, 2009a. doi: 10.1109/VISSOF.2009.5336425.

Philippe Moret, Walter Binder, and Alex Villazon. Cccp: complete calling context profiling in virtual execution environments. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '09, pages 151–160, New York, NY, USA, 2009b. ACM. ISBN 978-1-60558-327-3. doi: 10.1145/1480945.1480967. URL `http://doi.acm.org/10.1145/1480945.1480967`.

Nagy Mostafa and Chandra Krintz. Tracking performance across software revisions. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 162–171. ACM, 2009.

Colin J Neill and Philip A Laplante. *Antipatterns: Identification, Refactoring, and Management*. CRC Press, 2005.

Net Beans. Net beans profiler, 2013. URL `https://profiler.netbeans.org/`. Accessed: 2-9-2013.

Thanh H.D. Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Automated detection of performance regressions using statistical process control techniques. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 299–310, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1202-8. doi: 10.1145/2188286.2188344. URL `http://doi.acm.org/10.1145/2188286.2188344`.

NovaTec. inspectit, 2013. URL `http://www.inspectit.eu/en/home/`. Accessed: 17-8-2013.

Object Management Group consortium. Ml profile for marte: Modeling and analysis of real time and embedded systems, March 2003a. URL `www.omg.org/spec/MARTE/`.

Object Management Group consortium. Uml profile for schedulability, performance, and time specification, March 2003b. URL `http://www.dcl.hpi.uni-potsdam.de/teaching/EES03/UML-Profile.pdf`.

Oracle Corporation. Java virtual machine tool interface (jvmti). URL `http://docs.oracle.com/javase/6/docs/technotes/guides/jvmti/index.html`. Accessed: 30-9-2013.

Oracle Corporation. Using jconsole - java se monitoring and management, 2011a. URL `http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html`. Accessed: 16-9-2013.

Oracle Corporation. jhat - java heap analysis tool, 2011b. URL `http://docs.oracle.com/javase/6/docs/technotes/tools/share/jhat.html`. Accessed: 9-9-2013.

Oracle Corporation. Oracle jrockit jvm, 2013a. URL `http://www.oracle.com/technetwork/middleware/jrockit/overview/index.html`. Accessed: 6-9-2013.

Oracle Corporation. java.lang.management docs (java platform se 1.7), November 2013b. URL `http://docs.oracle.com/javase/7/docs/api/java/lang/management/package-summary.html`.

Oracle Corporation. Java management extensions (jmx) techology, 2013c. URL `http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`. Accessed: 18-9-2013.

Oracle Corporation. Jmx specification 1.4, 2013d. URL http://docs.oracle.com/javase/7/docs/technotes/guides/jmx/JMX_1_4_specification.pdf. Accessed: 23-9-2013.

Oracle Corporation. Overview of the jmx technology, November 2013e. URL http://docs.oracle.com/javase/tutorial/jmx/overview/index.html.

Oracle Corporation. The java virtual machine specification java se 7 edition, November 2013f. URL http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf.

Jiantao Pan. Software testing. *Dependable Embedded Systems*, 1999.

Zhelong Pan and Rudolf Eigenmann. Fast, automatic, procedure-level performance tuning. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 173–181. ACM, 2006.

Trevor Parsons. *Automatic detection of performance design and deployment antipatterns in component based enterprise systems*. PhD thesis, Citeseer, 2007.

J Perry. *Java Management Extensions:" Managing Java Applications with JMX"–Cover.-Includes Index*. O'Reilly, 2002.

Project Kenai. Visualvm - all-in-one java troublshootingtool, June 2013. URL http://visualvm.java.net/. Accessed: 23-9-2013.

Shah Faizur Rahman, Jichi Guo, and Qing Yi. Automated empirical tuning of scientific codes for performance and power consumption. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 107–116, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0241-8. doi: 10.1145/1944862.1944880. URL http://doi.acm.org/10.1145/1944862.1944880.

Raspberry Pi Foundation. Raspberry pi, 2013. URL http://www.raspberrypi.org/. Accessed: 13-9-2013.

Harald Röck. Survey of dynamic instrumentation of operating systems, 2007.

Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929–948, 2001.

M. Sanjeev Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A tutorial on particle filters for online nonlinear non-gaussian bayesian tracking. *Signal Processing, IEEE Transactions on*, 50(2):174–188, 2002. ISSN 1053-587X. doi: 10.1109/78.978374.

D. Schulz, W. Burgard, D. Fox, and A.B. Cremers. Tracking multiple moving targets with a mobile robot using particle filters and statistical data association. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 2, pages 1665–1670 vol.2, 2001. doi: 10.1109/ROBOT.2001.932850.

Martin Schulz and Bronis R de Supinski. Practical differential profiling. In *Euro-Par 2007 Parallel Processing*, pages 97–106. Springer, 2007.

Mauricio Serrano and Xiaotong Zhuang. Building approximate calling context from partial call traces. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 221–230, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3576-0. doi: 10.1109/CGO.2009.12. URL http://dx.doi.org/10.1109/CGO.2009.12.

Justin Seyster, Ketan Dixit, Xiaowan Huang, Radu Grosu, Klaus Havelund, Scott A Smolka, Scott D Stoller, and Erez Zadok. Aspect-oriented instrumentation with gcc. In *Runtime Verification*, pages 405–420. Springer, 2010.

Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 167–177, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337243.

Jon Skeet. Jbench - simple java benchmarking, 2013. URL http://jbench.sourceforge.net/index.html. Accessed: 21-9-2013.

C. U. Smith and L. G. Williams. *Performance Solutions: a practical guide to creating responsive, scalable software*. Addison-Wesley, 2002.

Connie U Smith and Lloyd G Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *Computer Measurement Group Conference*, pages 717–725, 2003.

Software Pearls. Collectionspy - the java profiler for the collections framework, 2009. URL http://www.collectionspy.com/. Accessed: 28-8-2013.

Marina Sokolova, Nathalie Japkowicz, and Stan Szpakowicz. Beyond accuracy, f-score and roc: A family of discriminant measures for performance evaluation. In Abdul Sattar and Byeong-ho Kang, editors, *AI 2006: Advances in Artificial Intelligence*, volume 4304 of *Lecture Notes in Computer Science*, pages 1015–1021. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-49787-5. doi: 10.1007/11941439_114. URL http://dx.doi.org/10.1007/11941439_114.

Steve Souza. Jamon (java application monitor), July 2011. URL http://jamonapi.sourceforge.net/. Accessed: 15-9-2013.

Tagtraum Industries. Gcviewer, 2011. URL http://www.tagtraum.com/gcviewer.html. Accessed: 11-9-2013.

Cristian Ţăpuş, I-Hsin Chung, Jeffrey K Hollingsworth, et al. Active harmony: towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Computer Society Press, 2002.

S. Thrun. Particle filters in robotics. In *Proceedings of the 17th Annual Conference on Uncertainty in AI (UAI)*, 2002.

TIOBE Software. Tiob programming community index for october 2013, October 2013. URL http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

Bhuvan Unhelkar. *The Art of Agile Practice: A Composite Approach for Projects and Organizations*. Auerbach Publications, 2012.

Distributed Systems Group University of Konstanz. Perfidix, 2012. URL http://disy.github.io/perfidix/. Accessed: 1-9-2013.

Poongavanam Vasanthanathan, Olivier Taboureau, Chris Oostenbrink, Nico PE Vermeulen, Lars Olsen, and Flemming Steen Jørgensen. Classification of cytochrome p450 1a2 inhibitors and noninhibitors by machine learning techniques. *Drug Metabolism and Disposition*, 37(3):658–664, 2009.

Qingping Wang, Sameer Kulkarni, John Cavazos, and Michael Spear. A transactional memory with automatic performance tuning. *ACM Trans. Archit. Code Optim.*, 8(4):54:1–54:23, January 2012. ISSN 1544-3566. doi: 10.1145/2086696.2086733. URL http://doi.acm.org/10.1145/2086696.2086733.

Greg Welch and Gary Bishop. An introduction to the kalman filter, 1995. URL http://clubs.ens-cachan.fr/krobot/old/data/positionnement/kalman.pdf.

Alexander Wert, Jens Happe, and Lucia Happe. Supporting swift reaction: automatically uncovering performance problems by systematic experiments. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 552–561, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486861.

John Whaley. A portable sampling-based profiler for java virtual machines. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 78–87, New York, NY, USA, 2000. ACM. ISBN 1-58113-288-3. doi: 10.1145/337449.337483. URL http://doi.acm.org/10.1145/337449.337483.

M. Woodside, G. Franks, and D.C. Petriu. The future of software performance engineering. In *Future of Software Engineering, 2007. FOSE '07*, pages 171–187, 2007. doi: 10.1109/FOSE.2007.32.

G. Xu and A. Rountev. Precise memory leak detection for java software using container profiling. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 151–160, 2008. doi: 10.1145/1368088.1368110.

Guoqing Xu. Finding reusable data structures. *SIGPLAN Not.*, 47(10):1017–1034, October 2012. ISSN 0362-1340. doi: 10.1145/2398857.2384690. URL http://doi.acm.org/10.1145/2398857.2384690.

Jing Xu. Rule-based automatic software performance diagnosis and improvement. *Performance Evaluation*, 67(8):585–611, 2010.

Dacong Yan, Guoqing Xu, and Atanas Rountev. Uncovering performance problems in java applications with reference propagation profiling. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 134–144, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL `http://dl.acm.org/citation.cfm?id=2337223.2337240`.

Changjiang Yang, Ramani Duraiswami, and Larry Davis. Fast multiple object tracking via a hierarchical particle filter. In *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, volume 1, pages 212–219. IEEE, 2005.

YourKit. Yourkit jprofiler, 2013. URL `http://www.yourkit.com/overview/index.jsp`. Accessed: 9-9-2013.

Appendix A

Repeating experiments

The experiments in this thesis can be repeated by checking out this repository using Mecurial.

https://fergus-hewson@bitbucket.org/fergus-hewson/buto-test-data-repository.

The readme file has instructions to repeat all of the tests that are presented in this thesis. This is mercurial is repository hosted at `www.bitbucket.com`. The repository contains an executable and all the test data used in the experiments presented in this thesis. The data is contained in the following zips, which need to extracted before analysis.

- Classloading.zip
- Consumer producer.zip
- Java references.zip
- Object leaker.zip

It contains a jar file `Buto-demo.jar` which is used to run the experiments. The following experiments can be reproduced.

- training-set-size
- smoothing
- time-point-generation
- bounds-matcher
- simple-classifier-tuning
- kalman-classifier-tuning-averageMethod
- kalman-classifier-tuning-processNoiseMethod
- particle-classifier-tuning-iterations
- particle-classifier-tuning-resamplePercentage
- particle-classifier-tuning-propagate
- particle-classifier-tuning-weightingMethod
- particle-classifier-tuning-particleCount
- overall

Results are output to a new folder in the root directory of the repository.

Appendix B

Test program code

This section contains the test-program code omitted from Section 3.1.

B.1 Java references

```
public class WeakObjectManager implements ObjectManager{
  private List<WeakReference<Object>> weakList = new ArrayList<WeakReference<
    Object>>();
  @Override public void addObject(Object obj) {
    weakList.add(new WeakReference<Object>(obj));
    if(weakList.size() % 10000 == 0){
      Iterator<WeakReference<Object>> iter = weakList.iterator();
      while(iter.hasNext()){
        WeakReference<Object> ref = iter.next();
        if(ref.get() == null){
          iter.remove();
        }
      }
    }
  }
  @Override public List<Object> getObjects() {
    List<Object> outputList = new ArrayList<Object>();
    for(WeakReference<Object> weakRef : weakList){
      Object obj = weakRef.get();
      if(obj != null)
        outputList.add(obj);
    }
    return outputList;
  }
}
```

Listing B.1: Weak object manager implementation. Uses weak references to hold onto objects,.

B.2 Object Leaker

```
public class Leaker {
  private static List<Object> objectList = new ArrayList<Object>();
  public static void main(String[] args) {
    new ShutdownThread(Integer.parseInt(args[0]) * 60 * 1000);
    final int leakCount = Integer.parseInt(args[1]);
    final int leakInterval = Integer.parseInt(args[2]);;
    final String objectName = args[3];
    while(true){
      for(int i = 0 ; i < leakCount ; i++)
        objectList.add(generateObject(objectName));
      try {Thread.sleep(leakInterval);} catch (InterruptedException e) {}
    }
  }
  private static Object generateObject(String objectName){
    Object obj = null;
    switch(objectName){
    case "thread":
      Thread thread = new Thread() {
        public void run() {
          while (true) {
            System.out.println("Hellow World");
            try {Thread.sleep(100);} catch (InterruptedException e) {}
          }
        }
      };thread.start();
      obj = thread;
      break;
    case"string":
      String str = "";
      for(int index = 0 ; index < 1000; index++)
        str = str + " " + System.currentTimeMillis();
      obj = str;
      break;
    case"stringBuilder":
      StringBuilder stringBuilder = new StringBuilder();
      for(int index = 0 ; index < 1000; index++)
        stringBuilder.append(" " + System.currentTimeMillis());
      obj = stringBuilder;
    }
    return obj;
  }
}
```

Listing B.2: Object leaker test program, simulates memory leaks by keeping strong references to objects so they are not garbage collected.

B.3 Classloading

```
public class Runner {
  private static IDomainObject mainObject;
  public static void main(String[] args) throws Exception {
    new ShutdownThread(Integer.parseInt(args[0]) * 60 * 1000);
    int leakInterval = Integer.parseInt(args[1]);
    int index = 0;
    mainObject = DomainObjectFactory.newInstance();
    mainObject.setName("Domain Object " + index++);
    while (true) {
      IDomainObject newObj = DomainObjectFactory.newInstance();
      newObj.copy(mainObject);
      mainObject = newObj;
      mainObject.setName("Domain Object " + index++);
      Thread.sleep(leakInterval);
    }
  }
}
```

Listing B.3: Class loading test program runner

Figure B.4 outlines the interface for the domain object for the application. It contains a name field accessed through getters and setters, a `ConfiguraitonObject` property and a clone method that copies the state of the provided `IDomainObject` into the current object.

```
public interface IDomainObject {
  String getName();
  void setName(String name);
  void copy(IDomainObject domainObject);
  ConfigurationObject getConfiguration();
}
```

Listing B.4: Class loading test domain object interface

```
public class DomainObject implements IDomainObject {
  private String name;
  private ConfigurationObject configObject;
  public String getName(){
    return name;
  }
  public void setName(String name){
    this.name = name;
  }
  public ConfigObject getConfiguration(){
    // Create copy of configuration object
    return new ConfigObject(configObject);
  }
  public void copy(IDomainObject domainObject) {
    setName(domainObject.getName());
    configObject = domainObject.getConfiguration();
  }
}
```

Listing B.5: Class loading test program domain object

```
public class DomainObjectFactory {
  public static IDomainObject newInstance() throws Exception{
    URL classpathUrl = new URL("file:" + DomainObjectFactory.class.
    getProtectionDomain().getCodeSource().getLocation().getPath());
    URLClassLoader tmp = new URLClassLoader(new URL[] { classpathUrl }) {
      public Class<?> loadClass(String name)throws ClassNotFoundException {
        if ("nz.ac.massey.buto.testprofiles.classloading.DomainObject".equals(
    name))
          return findClass(name);
        return super.loadClass(name);
      }
    };
    return (IDomainObject) tmp.loadClass("nz.ac.massey.buto.testprofiles.
    classloading.DomainObject").newInstance();
  }
}
```

Listing B.6: Class loading test program domain object factory

```
public class ConfigObject  {
  // LEAK - keep reference to parent configuration
  private ConfigObject configObject;
  public ConfigObject(ConfigObject configObject) {
    this.configObject = configObject;
  }
}
```

Listing B.7: Classloading configuration object

B.4 Consumer Producer

This section outlines code used in the consumer producer test program that was included in section 3.1.1. It is broken into two subsections, one for the JMX related code and one for the domain objects used in the example.

```
public class Runner implements ProductionMXBean{
  public static void main(String[] args) throws Exception{
    if(args.length != 4){return;}
    new ShutdownThread(Integer.parseInt(args[0]) * 60 * 1000);
    int producerCount = Integer.parseInt(args[1]);
    int consumerCount = Integer.parseInt(args[2]);
    int bufferSize = Integer.parseInt(args[3]);
    new Runner(producerCount, consumerCount, bufferSize);
  }
  private BlockingQueue <List<Student>> queue;
  private Counter jobCounter = new Counter();
  public Runner(int producerCount, int consumerCount, int bufferSize) throws
    Exception {
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
    ObjectName name = new ObjectName("nz.ac.massey.buto.testprofiles.:type=
    Production,id=1");
    mbs.registerMBean(this, name);
    queue = new ArrayBlockingQueue<List<Student>>(bufferSize);
    // start producers
    for(int index = 0; index < producerCount; index++){
      Thread thread = new Thread(new Producer(queue), "producer - " + index);
      thread.start();
    }
    // start consumers
    for(int index = 0; index < consumerCount; index++){
      Thread thread = new Thread(new Consumer(queue, jobCounter), "consumer - " +
     index);
      thread.start();
    }
  }
  @Override public ProductionMBeanInfo getProductionMBeanInfo() {
    return new ProductionMBeanInfo(getMaxBufferSize(), queue.size(), jobCounter.
    getProcessedJobs());
  }
}
```

Listing B.8: Consumer Producer runner, producers make objects which are put into a buffer for consumer use.

```
public class Consumer implements Runnable {
  private BlockingQueue <List<Student>> queue;
  private Counter counter;
  private boolean stopped = false;
  public Consumer(BlockingQueue<List<Student>> queue, Counter counter) {
    this.queue = queue;
    this.counter = counter;
  }
  public void stop() {stopped = true;}
  @Override public void run() {
    while (!stopped) {
      try{
        new WebSiteBuilder().buildWebSite(queue.take());
```

```
14          counter.increase();
15        } catch (InterruptedException e) {}
16      }
17    }
18  }
```

Listing B.9: Consumer object definition.

```
1  public class Producer implements Runnable {
2    private BlockingQueue <List<Student>> queue;
3    private boolean stopped = false;
4    public Producer(BlockingQueue<List<Student>> queue) {
5      this.queue = queue;
6    }
7    public void stop() {stopped = true;}
8    @Override public void run() {
9      while (!stopped) {
10       List<Student> list = new ArrayList<Student>();
11       for (int index = 0; index < 100; index++)
12         list.add(StudentFactory.createRandomStudent());
13       try {
14         queue.put(list);
15       } catch (InterruptedException e) {}
16     }
17   }
18 }
```

Listing B.10: Producer object definition.

B.4.1 JMX

This section presents the JMX related objects used in the consumer producer test program. Interfaces for the `ProductionMXBean` and `CounterMXBean` are defined in listings B.12 and B.11. An composite data object for the `ProductionMXBean` is defined in Listing B.13. The counter used to monitor the consumption of data items and emit notifications when intervals have been met is defined in B.14.

```
1  public interface ProductionMXBean {
2    public ProductionMBeanInfo getProductionMBeanInfo();
3    public void stop();
4  }
```

Listing B.11: Interface for ProductionMXBean

The `ProductionMXBean` interface is exposes important information about the state of the test. The size of the buffer, how fill the buffer is and the number of processed jobs is exposed through the methods `getBufferSize`, `getMaxBufferSize` and `getProcessedJobCount`. A composite data item `ProductionMBeanInfo` is also made available, this object is outlined in Listing B.13 and contains the same data already exposed in the bean. A stop method is included for debugging purposes.

```
1  public interface CounterMXBean {
2    public int getProcessedJobs();
3  }
```

Listing B.12: Interface for CounterMXBean

The interface for the `CounterMXBean` is outlined in Listing B.12. It exposes one property, the number of jobs processed by the consumers. This data is made available through the `ProductionMXBean` but to emit notifications the job counter needed to be a valid MXBean.

```
public class ProductionMBeanInfo {
  private int bufferSize, queueSize, processedJobCount;
  @ConstructorProperties({"bufferSize", "queueSize", "processedJobCount"})
  public ProductionMBeanInfo(int bufferSize,  int queueSize, int
    processedJobCount){
    this.bufferSize = bufferSize;
    this.queueSize = queueSize;
    this.processedJobCount = processedJobCount;
  }
  public int getBufferSize(){
    return bufferSize;
  }
  public void setBufferSize(int bufferSize){
    this.bufferSize = bufferSize;
  }
  public int getQueueSize(){
    return queueSize;
  }
  public void setQueueSize(int queueSize){
    this.queueSize = queueSize;
  }
  public int getProcessedJobCount(){
    return processedJobCount;
  }
  public void setProcessedJobCount(int processedJobCount) {
    this.processedJobCount = processedJobCount;
  }
}
```

Listing B.13: ProductionMBeanInfo object definition.

The object `ProductionMBeanInfo` is used to test the data collectors ability to handle complex data items outlined in the JMX API. The object exposes the same data outline in the `ProductionMXBean` but is handled different by agents who monitor the object. It contains getters and setters for its properties `bufferSize`, `queueSize` and `processedJobCount`.

```
public class Counter extends NotificationBroadcasterSupport implements
    CounterMXBean {
  private int processedJobs = 0;
  private long sequenceNumber = 0;
  private long timestamp = System.currentTimeMillis();
  private long startTime =  System.currentTimeMillis();;
  private static final int LOG_INTERVAL = 100000;
  public Counter() throws Exception{
    super();
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
    ObjectName oname = new ObjectName("nz.ac.massey.buto.testprofiles:type=
    Counter,id=1");
    mbs.registerMBean(this, oname);
```

```
12    }
13    public synchronized void increase () {
14      processedJobs++;
15      if (processedJobs%LOG_INTERVAL==0) {
16        long now = System.currentTimeMillis();
17        long diff = now-timestamp;
18        timestamp = now;
19        String logString = "job counter: it took " + diff + " ms to process " +
      LOG_INTERVAL + " jobs";
20        Notification n = new Notification("Processed Job MileStone", this,
      sequenceNumber++, now - startTime, logString);
21        sendNotification(n);
22      }
23    }
24    @Override public synchronized int getProcessedJobs() {
25      return processedJobs;
26    }
27  }
```

Listing B.14: Counter object definition.

A `Counter` object is passed to all consumers to track the number of data items consumed. After consumption consumers call the increase method which increments the count and if an interval been met a notification is broadcast to using the JMX API. The data collector will asynchronously receive the notifications, storing for later analysis. The counter is also an MBean which exposes the number of processed in it's interface (Listing B.12).

B.4.2 domain

This section defines code from the consumer producer test program that comprise the domain of the example. The data item that is at the center of the example is a Student and is defined in Listing B.15. Data items are produced in a student factory outlined in Listing B.16. Data items are consumed by a small website builder outlined in Listing B.17.

```
1  public class Student {
2    private int id;
3    private String firstName, name, degree;
4    public int getId() {
5      return id;
6    }
7    public void setId(int id) {
8      this.id = id;
9    }
10   public String getName() {
11     return name;
12   }
13   public void setName(String name) {
14     this.name = name;
15   }
16   public String getFirstName() {
17     return firstName;
18   }
19   public void setFirstName(String firstName) {
20     this.firstName = firstName;
```

```
21    }
22    public String getDegree() {
23      return degree;
24    }
25    public void setDegree(String degree) {
26      this.degree = degree;
27    }
28    @Override
29    public String toString() {
30      return "Student [id=" + id + ", name=" + name + ", firstName=" + firstName +
        ", degree=" + degree + "]";
31    }
32  }
```

Listing B.15: Student object definition.

The `Student` object is the data item used in the consumer producer test program. It is a simple Java bean with properties for `id`, `firstName`, `lastName` and `degree`. Listing B.15 defines the `Student` object.

```
1   public class StudentFactory {
2     private static final String[] lastNames = {
3       "Smith","Miller","MacDonald","Key","Taylor","O'Brien","Bush","Gupta","Wang","
        Sanchez","Mueller"
4     };
5     private static final String[] firstNames = {
6       "John","Jim","Bob","Kate","James","Brian","Basil","George","Liz","Linda","
        Anna","Maria"
7     };
8     private static final String[] degrees = {
9       "BInfSci SE","BSC CS","BE CSE","BSc Mathematics","BE Food Technology","BSc
        Chemistry"
10    };
11    private static int lastId = 0;
12    private static String getRandomValue(String[] values) {
13      Random random = new Random();
14      return values[random.nextInt(values.length)];
15    }
16    public static Student createRandomStudent() {
17      Student s = new Student();
18      s.setId(++lastId);
19      s.setName(getRandomValue(lastNames));
20      s.setFirstName(getRandomValue(firstNames));
21      s.setDegree(getRandomValue(degrees));
22      return s;
23    }
24  }
```

Listing B.16: StudentFactory object definition.

The student factory is used to simulate production of data-items for the consumers to consume. The main method is `produceRandomStudent()` which generates new student with a unique `ID` and a random `name`, `firstName` and `degree`.

```
1   public class WebSiteBuilder{
2     public String buildWebSite(List<Student> students) {
3       StringBuilder builder = new StringBuilder();
4       builder.append("<html>\n")
```

```
5       .append("<body>\n")
6       .append("<h3>Student List</h3>\n")
7       .append("<table border=\"1\">\n")
8       .append("<tr><th>ID</th><th>FIRST NAME</th><th>NAME</th><th>DEGREE</th></tr>\
        n");
9
10      for (Student s:students) {
11        builder.append("<tr>")
12        .append("<td>").append(s.getId()).append("</td>")
13        .append("<td>").append(s.getFirstName()).append("</td>")
14        .append("<td>").append(s.getName()).append("</td>")
15        .append("<td>").append(s.getDegree()).append("</td></tr>\n");
16      }
17      builder.append("</table>\n").append("</body>\n").append("</html>\n");
18
19      return builder.toString();
20    }
21  }
```

Listing B.17: WebSiteBuilder object definition.

The website builder is used to simulate consumption of the data-items. A simple website builder is consumes a list of student objects and generates a web page display their properties.

B.5 Shut-down thread

A shut-down thread is used in all test programs to control the execution time of the test. Listing B.18 defines the shut-down thread. No clean up of resources is performed before using `System.exit()` to close the VM.

```
1  public class ShutdownThread extends Thread{
2    public int executionTime;
3    public ShutdownThread(int executionTime){
4      this.executionTime = executionTime;
5      start();
6    }
7    public void run() {
8      try {
9        Thread.sleep(executionTime);
10     } catch (InterruptedException e){}
11     System.exit(0);
12   }
13 }
```

Listing B.18: Shutdown Thread object definition.

Appendix C

Analysis code

This section contains the analysis code omitted from Section 4.

C.1 Kalman Filter

```
kalmanClassifier(timepointList){
  boundsList = []
  q = 0
  firstTimePoint = true
  // Process time-points
  for timepoint in timepointList{
    // Process first time-point - init filter near first measurement
    if firstTimePoint{
        // Get measurement
      measuredState = timepoint.mean
      measuredVariance = timepoint.stdev * deviations
            // Set predicted state
            predictedState = measuredState
            predictedVariance = measuredVariance
            // Calculate bounds
      lowerBound = predictedState - predictedVariance
      upperBound = predictedState + predictedVariance
      boundsList.append(new Bounds(timepoint.timestamp, lowerBound, upperBound))
      // Continue processing
      firstTimePoint = false
      continue
    }
    // Get the measured distribution
    measuredState = timepoint.mean
    measuredVariance = timepoint.stdev
    // Calculate the Kalman gain
    kalmanGain = predictedVariance / (predictedVariance + measuredVariance)
    // Calculate the predicted state
    predictedState = kalmanGain * (measuredState - predictedState) +
    predictedState
    // Calculate the predicted variance
    predictedVariance = predictedVariance - (kalmanGain * predictedVariance)
    predictedVariance = (predictedVariance + q) * (1 - kalmanGain)
    // Calculate allowed deviation
    deviation = predictedVariance * deviations
    // Calculate bounds and add to bound list
```

```
    lowerBound = predictedState - deviation
    upperBound = predictedState + deviation
    boundsList.append(new Bounds(timepoint.timestamp, lowerBound, upperBound))
  }
  return boundsList
}
```

Listing C.1: The Kalman Filter classification algorithm.

```
private double calcProcessNoise(List<TimePoint> data, String processNoiseMethod){
  switch(processNoiseMethod){
  case"averageDifference":
    double[] differences = new double[data.size() -1];
    double prevVal = data.get(0).valueMean;
    for(int index = 1 ; index < data.size(); index++){
      TimePoint timepoint = data.get(index);
      double val = timepoint.valueMean;
      differences[index - 1] = Math.abs(prevVal - val);
      prevVal = val;
    }
    return StatUtils.mean(differences);
  case"averageVariance":
    double[] values2 = new double[data.size() ];
    for(int index = 0 ; index < data.size(); index++){
      TimePoint timepoint = data.get(index);
      values2[index] = timepoint.valueStdev;
    }
    return StatUtils.mean(values2);
  }
  return 0.0;
}
```

Listing C.2: Method used to calculate the process noise for the Kalman filter.

C.2 Particle filter

```
pf (timePointList, particleCount){
  particleList = []
  bounds = []
  // Initialise particles
  for i = 0 : particleCount{
    particle.x = getInitialState()
    particle.w = 1 / particleCount
    particleList.add(particle)
  }
  previousPredictedState = null
  velocity = 0
  // Iterate over time points
  for timepoint in timePointList{
    // Propagate particles
    propagateParticles(velocity);
    // Get the measured distribution
    measuredState = timepoint.mean
    measuredVariance = timepoint.stdev
    // Update particles state and weightings
```

```
    weightSum = 0
    for particle in particleList{
      particle.w = updateWeight(measuredState,measuredVariance)
      weightSum += particle.w
    }
    // Normalise Weightings
    for particle in particleList
      particle.w = particle.w / weightSum
    // If effs below re-sample threshold then re-sample
    if effectiveSampleSize(particleList) < resampleThreshold
      particles = resampleParticles(particleList)
    // Calculate the new predicted distribution
    predictedState = mean(particleList.x)
    predictedStdev = sqrt(variance(particleList.state))
    // Calculate allowed deviation
    deviation = predictedStdev * deviations
    // Calculate bounds
    lowerBound = predictedState - deviation
    upperBound = predictedState + deviation
    // Add new bounds
    boundsList.append(new Bounds(timepoint.timestamp, lowerBound, upperBound))
    // Update previous predicted state and velocity
    if previousPredictedState == null{
      previousPredictedState = predictedState
    }else{
      velocity = predictedState - prevPredictedState
      previousPredictedState = predictedState
    }
  }
}
```

Listing C.3: The Particle Filter classification algorithm.

```
effectiveSampleSize(particleList){
  // Sum weights
  sumSqrd = 0
  // Iterate over particles adding weights ^ 2 to sumSqrd
  for particle in particleList{
    sumSqrd += particle.w ^ 2
  }
  // Return inverse of sumsqrd
  return 1 / sumSqrd
}
```

Listing C.4: Effective sample size evaluation method.

```
resampleParticles(particleList){
  // Sort particles from smallest for largest weighting
  sort(particleList)
  // Build cumulative weightings array
  cw[0] = particleList[0].w
  for i = 1 : particles.size{
    cw[i] = cw[i-1] + particleList[i].w
  }
  // The new distribution
  newDistribution = []
  // Sample new particles
  for i = 0 : particleList.size{
```

```
    // Generate random value
    randomValue = random.next()
    // Search for insertion point of random value in cw
    selectedIndex = binarySearch(cw, randomValue)
    // Clone the sampled particle
    newDistribution.add(particleList[selectedIndex].clone())
  }
  // Return new distribution
  return newDistribution
}
```

Listing C.5: Method used to re-sample particles with respect to their weightings.

```
updateState(particleList, velocity, processNoiseDist, propagateMethod){
  for particle in particleList{
    // Calculate noise
    noise = processNoiseDist.sample()
    // Update state
    switch(propagateMethod){
    case"none":
      particle.x = particle.x +  noise
    case"velocity":
      particle.x = particle.x + velocity + noise
    }
  }
}
```

Listing C.6: Method used to update the particles weight after each iteration.

```
updateWeight(particle, measuredState, measuredStdev){
  switch(weightMethod){
  case"distance":
    distance = Abs(particle.x - measuredState)
    // Inverse the distance, so closer particles have a higher weighting
    weight = 1 / distance;
  case"gausian":
    distribution = new NormalDistribution(measuredState, measuredStdev)
    weight = distribution.desnity(particle.x)
  }
  return weight
}
```

Listing C.7: Method used to update particle weights.

C.3 Bounds matcher

```
// List of bounds
boundList
getBounds(timeStamp, collectionInterval){
  // The bound object to return
  outputBound = null
  // Calculate approx difference between timestamps
  timeStampDifference = boundList.get(1).timestamp -  boundList.get(0).timestamp
  // Iterate over list of bounds to find the closest one
  for(bound : boundList){
```

```
    // Calculate difference between bound and timestamps
    diff = Math.abs(bound.timestamp - timestamp)
    // Test if difference less than current timestamp difference
    if(diff < collectionInterval){
      outputBound = bound
      timeStampDifference = diff
    }
  }
  // Return collected bound
  return outputBound
}
```

Listing C.8: Simple Bounds Matcher Algorithm, matches time-stamps to the nearest bounds.

```
// List of bounds
boundList
getBounds(timeStamp, collectionInterval){
  // Test if time-stamp less than first bounds
  if(timestamp < boundList.get(0).timestamp)
    return boundList.get(0)
  // Pointers to keep track of previous and current bounds
  previousBounds = boundList.get(0)
  currentBounds = null
  // Iterate over the list of bounds
  for(index = 1; index < boundList.size(); index++){
    // Get the current bounds
    currentBounds = boundList.get(index)
    // Test if found the higher bound
    if(timestamp < currentBounds.timestamp){
      // Lower bound
      lowerBound = linearInterpolate(prevBounds.timeStamp, prevBounds.lowerBound,
     currentBounds.timeStamp, currentBounds.lowerBound, timeStamp)
      // Upper bound
      upperBound = linearInterpolate(prevBounds.timeStamp, prevBounds.upperBound,
     currentBounds.timeStamp, currentBounds.upperBound, timeStamp)
      // Create new bounds object and return it
      return new Bounds(timestamp, lowerBound, upperBound)
    }
    // Update previous bounds
    previousBounds = currentBounds
  }
  // Test if within range of last bound
  if(timeStamp - currentBounds.timeStamp < boundList.get(0).timeStamp)
    return currentBounds
  // Timestamp is out of range of bounds, return null
  return null
}
```

Listing C.9: Interpolation Bounds Matcher Algorithm, uses interpolation to generate bounds for a given time-stamp based on the bounds the time-stamp falls between.

```
linearInterpolate(x1, y1, x2, y2, x3){
  // Check if x3 falls between x1 and x2, if not return null
  if(x3 < x1 || x3 > x2){return null}
  // Calculate the x and y distance between points
```

```
  width = x2 - x1
  height = y2 - y1
  // Calculate the offset of x3 from the first point, on the x axis
  x3Offset = x3 - x1
  // Calculate the ratio of the x3offset value to the x distance between points
  xRatio = x3Offset / width
  // Calculate the offset of the interpolated y3 value
  y3Offset = height * xRatio
  // Calculate the total value of y3
  y3 = y3Offset + y1
  return y3
}
```

Listing C.10: Method to interpolate a Y value between two points given an X value.

C.4 Meta-data classifier

```
metaDataModel
train(trainingData, filterList){
  // Removed filtered items
  for map in trainingData{
    map.filter(filterList)
  }
  // Single training-run, set model to that data
  if trainingData.size == 1{
    model = trainingData[0]
    return
  }
  // Set a base map to compare other maps to
  baseMap = trainingData[0]
  // Compare other maps to base map
  for i = 1 : trainingData.size(){
    currentMap = trainingData[i]
    // Iterate over keys and compare
    for key in baseMap.keys{
      baseObj = baseMap.get(key)
      currentObj = currentMap.get(key)
      if currentObj == null{
        throw Error()
      }else if baseObj != currentObj{
        throw Error()
      }
    }
    // Check for unprocessed keys
    currentMap.filter(baseMap.keys)
    if currentMap.size != 0{
      throw Error()
    }
  }
  // Set model as the base map
  metaDataModel = baseMap
}
```

Listing C.11: Meta-data training algorithm, checks consistency of training meta-data before allowing model to be built.

```
metaDataModel
filterList
classify(testrun){
  // list of differences
  differences = []
  // filter
  testrun.filter(filterList)
  // Compare keys in model to the test run
  for key in metaDataModel.keys{
    modelObj = model.get(key)
    testObj = testrun.get(key)
    // Compare
    if testObj == null{
      differences.add("Missing")
    }else if model != testObj{
      differences.add("Different")
    }
  }
  // Remove processed keys from test-run
  testrun.filter(metaDataModel.keys)
  for key in testrun.keys{
    differences.add("Extra")
  }
  return differences
}
```

Listing C.12: Meta-data classification algorithm, compares meta-data from training-run against meta-data model.

C.5 Notification classifier

```
notificationModel = {}

train(trainingData, deviations){
  // Map for grouping notifications
  groupingMap = {}
  // Add all notifications in training data to list
  for notificationList in trainingData{
    for notification in notificationList{
      // Generate key for the notifications
      key =  getKey(notification)
      // If list exists, add to it, else create new list
      if groupingMap.contains(key){
        groupingMap.get(key).append(notification)
      }else{
        keyList = [notification]
        groupingMap.put(key, keyList)
      }
    }
  }
```

```
  //Test all groupings are the same size
  groupingSize = groupingMap.get(0).size
  for grouping on groupingMap{
    if(grouping.size != groupingSize)
      throw Error()
  }
  // Generate bounds for groupings of notifications
  for key in groupingMap.keys{
    // Get list of notifications
    notificationList = groupingMap.get(key)
    // Calculate distribution of points
    mean = average(notificationList.timestamps)
    stdev = stdev(notificationList.timestamps)
    // Calculate bounds for grouping
    lowerBound = mean - (stdev * deviations)
    upperBound = mean + (stdev * deviations)
    // Add grouping to model
    notificationModel.put(key, new Bounds( lowerBound, upperBound))
  }
}
```

Listing C.13: Notification Training Algorithm, checks training-run notifications for consistency before building model.

```
notificationModel

classify(testRun){
  // List of differences to return
  differences = []
  // List of bounds to process
  unprocessedBounds = notificationModel.values
  //Iterate over data
  for notification in testRun{
    // Generate key for the notification
    key = getKey(notification)
    // Test if the model contains a grouping with that key
    if notificationModel.contains(key){
      //Get bounds and test if notification with bounds.
      bounds = notificationModel.get(key)
      // Check for duplicate
      if(unprocessedBounds.contains()){
        differences.append("Duplicate notification")
      }
      if(notification.timestamp < bounds.lowerBound || notification.timestamp >
    bounds.upperBound){
        differences.append("Out of bounds")
      }
      // Removed bounds from list of unprocessed bounds
      unprocessedBounds.remove(bounds)
    }else{
      differences.append("No matching bounds")
    }
  }
  // Add all unmatched groupings as errors
  for bounds in unprocessedBounds{
    differences.append("Bounds not matched")
  }
  return differences
```

```
}
```

Listing C.14: Notification Classification Algorithm, compares test-run notifications against model.

C.6 Test-run classifier

```
dynamicPropertyModelMap = {}
metaDataClassifier
notificationClassifier
buildClassifier(trainingData, butoPropertyList, dataSmoother, timePointGenerator,
     classificationAlgorithm, boundsMatcherFactory, metaDataFilterList){
  // Create dynamic-property models
  forEach property in butoPropertyList{
    // Get property data from training runs
    dataList = []
    for trainingRun in trainingData{
      //Get data smooth and add to data list
      data = trainingRun.get(property)
      smoothedData = dataSmoother.smooth(data)
      dataList.append(smoothedData)
    }
    // Generate time-point list
    timePointList = timePointGenerator.generate(dataList)
    // Generate model and add to map of models
    model = classificationAlgorithm.generateBounds(timePointList)
    dynamicPropertyModelMapo.put(property, model)
  }
  // Create meta-data model, extract data from training-runs
  metaDataList = []
  for trainingRun in trainingData{
    metaDataList.append(trainingRun.getMetaData())
  }
  // Build meta-data classifier
  metaDataClassifier = new MetaDataClassifier(metaDataList, metaDataFilterList )
  // Create notification model, extract notification data from training-runs
  notificationList = []
  for trainingRun in trainingData{
    notificationList.append(trainingRun.getNotificaitons())
  }
  // Build notification classifier
  notificationClassifier = new NotificationClassifier(notificationsList)
}
```

Listing C.15: Test-Run Classifier Training Algorithm, builds models for Meta-Data, Notifications and each Buto Property.

```
dynamicPropertyModelMap
metaDataClassifier
notificationClassifier
boundsMatcherFactory

classifiy(testRun){
  // Map of dynamic-property results
```

```
8   dynamicPropertyResultMap = {}
9   // Classify each property in the dynamic model map
10  for property in dynamicPropertyModelMap.keys{
11    // Get data from test-run
12    data = testRun.get(property)
13    // Smoothe test-data
14    smoothedData = dataSmoother.smooth(data)
15    // Get model from dynamic-property map
16    model = dynamicPropertyModelMap.get(property)
17    // Create bounds matcher from model
18    boundsMatcher = boundsMatcherFactory.createBoundsMatcher(model)
19    // Classify test-run data against model
20    results = boundsMatcher.classify(smoothedData)
21    dynamicPropertyResultMap.put(property, results)
22  }
23  // Get and classify meta-data
24  metaData = testRun.getMetaData()
25  metaDataResults = metaDataClassifier.classify(metaData)
26  // Get and classify notifications
27  notifications = testRun.getNotifications()
28  notificationResults = notificationClassifier.classify(notifications)
29  // Return results from each classifier
30  return new ClassificationResult(dynamicPropertyResultMap, metaDataResults,
      notificationResults)
31 }
```

Listing C.16: Test-Run Classification Algorithm, compares a test-run against a trained classifier.

Acronyms

ADL Architecture Description Language

AI Artificial Intelligence

ANOVA Analysis of Variance

AOP Aspect-Oriented programming

API Application Programming Interface

CCT Calling Context Tree

CPU Central-Processing-Unit

DBMS Database Management System

EJB Enterprise Java Beans

GC Garbage Collector

HPC High Performance Computing

IDE Integrated Development Environment

JDK Java Development Kit

JHAT Java Heap Analysis Tool

JIT Just-In-Time Compiler

JMX Java Management Extensions

JRE Java Runtime Environment

JVM Java Virtual Machine

JVMPI Java Virtual Machine Profiler Interface

JVMTI Java Virtual Machine Tool Interface

LQN Layered queueing network

MCC Mathews correlation coefficient

MDD Model Driven Development

MDE Model-driven engineering

ML Machine Learning

OMG Object Management Group

ORM Object-relational mapping

QoS Quality of Service

SPE Software Performance Engineering

TDD Test-Driven-Development

UML Unified Modelling Language

VM Virtual Machine

WMI Windows Management Instrumentation

XP eXtreme Programming