Evaluating Library Configurations

by

Kimiisa Oshikoji

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2012

Abstract

A wide range of libraries are available for a developer to choose from when building a software system, but once the library is chosen, the developer must determine which version of the library to use. Is there some characteristic that can identify the optimal version of a library to use? Even if a library compiles correctly, there could be a better version of that library that will provide better error handling, or improved security. In particular, the developer would prefer to avoid poor configurations: that is, sets of libraries that perform poorly, or not at all.

This paper describes a method by which a sub performing version of a library can be identified from the behavior observed from different configurations of the library. Each library is measured by performance and trace analysis. During the course of these runs, different configurations of the library are substituted in and the results are collected to be analyzed.

The results of this analysis shows that there is no quick way to identify a sub performing library. However such a library can be determined through concentrated efforts to collect and analyze time-based data.

Acknowledgements

I would like to thank my mom and dad, my two brothers, my grandmother, my cousins, my aunts and uncles, my friends and my supervisors.

Dedication

This is dedicated to my family and my passion of reading, including my love of Star Wars and Star Trek.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Deciding what version of a library to use is not an easy decision to make. A great deal of thought must be taken in answering the following questions. Do I want to go with a recent version? Do I want to use an older version? Do any of my fellow programmers have a preference? Are there any compatibility issues that necessitate a certain library version? Which version provides the best performance? The answers to these questions play an important role in determining what library version to use.

A project contains source code that, when executed, performs some action (or behavior). Developers often make use of programs, written by other developers, that they treat as black-boxes. These programs are referred to as libraries. A set of libraries is referred to as a configuration.

Determining which version of a library will lead to the best performance is a nontrivial task. Executing every version of a library in conjunction with the base project takes a large amount of resources to accomplish. In addition, not every version of a library uses the same Application Programming Interface (otherwise known as an API) as all the other versions of that library. For instance, systems that use the Version 4.0 of the jUnit[1] test library are incompatible with jUnit Version 3.0. The work needed to make the base project and library compatible is a time and resource sink that could be better spent elsewhere.

For most popular commercial software, upgrades are automatic[2]. In fact, the popular Microsoft Windows Operating System has the ability to download and install an upgrade without any user input. Matters are not so straightforward for a library. A library can be incorporated in a wide range of projects that have nothing in common. This leads

[1] http://www.junit.org/

to situations where a change in the library can be beneficial for one project and hinder another project. An example of this can be found in the situation where the same MongoDB driver is being implemented in two different ways[2]. The first implementation focuses on a lightweight approach that is fast and stable[3]. The second approach strives to be heavy weight by supporting additional functionality[4]. Both of these MongoDB drivers would be considered useful to different projects with different needs. Another example of this is when a library chooses an upgrade path that emphasizes speed over efficiency to support a databases. A project that wants quick responses when it queries a database would prefer the latest version of this library. Another project that runs in a mobile environment, where memory is at a premium, would stick with an older versions of the library. The issue at hand is to determine if there is a method by which the performance and behavior of a project can be used to decide between library versions that emphasize different features.

The research question of this case study is: is it possible to detect problematic configurations of libraries for a statically constant system? When the base code is changed, the system behavior will also change; however, the result is not known when the system itself is unchanged but the support libraries are changed.

The use of libraries by programmers is important, yet there is not much research that focuses on libraries and how a programmer can go about selecting the proper combination of library versions. The result of this case study will pinpoint what library version should not be selected in the final library configuration.

This problem cannot be solved by updating each library to the latest version. This is due to the fact that whenever an upgrade to a newer version is initiated there is always the risk that the new version is not compatible with the project from both a software and behavior perspective. An example of this is when a library call is depreciated[5]. Another form of organizing the system, through components, is lacking in ways to classify a library properly [3].

What little concern has been taken for libraries is focused on finding actual problems with those libraries (e.g., [10]). Developers take steps to ensure that a library is compatible with the base system (i.e., when the library is called the system does not crash). However, there is now a move towards acknowledging the important role libraries play. In the past, most developers took steps to ensure that deadlocks did not occur in their projects. There is now even research in determining if the interaction between libraries can cause deadlocks

[2]http://www.mongodb.org/display/DOCS/Drivers
[3]https://bitbucket.org/rumataestor/emongo
[4]https://github.com/wpntv/erlmongo
[5]https://github.com/voldemort/voldemort/commit/79a6660f96ef74450734e79860b98c3796e7dc14

to occur [10].

Gathering social data is the new trend in today's world. One group of researchers has even begun gathering social data regarding library usage [7]. Their idea is to use the user's preferences to guide how developers select library versions.

The goal of this thesis is to determine which version of a library is worst, performance wise, autonomously. One factor to consider is the behavior of a project when a library is changed. Does the resulting project do what the developer wants? Is there some characteristic that can be used to rank the library versions with each other? For instance, if a library has versions from 1.0 to 1.99 is there a method that can be used to prioritize which version should be used before another? Normally the latest version is the best, but rollbacks happen all the time. A rollback occurs when a library is upgraded to a newer version but some issue causes the developers to choose to return to a previous version of that library. In this case study, a developer message was encountered that indicated a library version rollback due to a performance issue[6].

Chapter 2 provides an overview of related work. Chapter 3 describes the framework for evaluating the different library configurations. Chapter 4 describes the criteria used to select the test system, the algorithms, the methodology and the analysis. Chapter 5 discusses threats to validity and the steps taken to mitigate them along with different ways this case study can be built upon and extended. Chapter 6 summarizes the work and discusses possible future directions.

[6]https://github.com/voldemort/voldemort/commit/e52f4911b83ca2e29427b104e129afee40cc5cfc

Chapter 2

Related Work

This section is divided into six parts:

1. What is Maven?
2. Dividing a system into components.
3. Static analysis of the system.
4. Mining a software repository.
5. Using a trace to analyze the system.
6. Analyzing a system through Software Reconnaissance.

2.1 Maven

Maven is an automated build tool for projects. This automation is evident at compile time when Maven accesses the Internet to use the latest version of the libraries called in the project. Maven seems to solve the problem of choosing library versions by forcing the use of the latest version automatically [1]. By doing this, the project is always updated to the latest version of each library. This drive towards automation is also useful in that during the build stage, any forgotten step can lead to catastrophe [11].

There are many problems associated with updating a library. There might not be enough time to test the new library for compatibility with the project. The new library might have an updated API that is not compatible with the current project. There is also an issue with security considerations (newer software inevitably requires steps to be taken to provide security). Another issue is that the performance of the library might have changed. If a user had chosen a particularly lightweight and efficient version of the library, they would not be interested in the latest feature-filled version. By automatically updating a library, unforeseen problems can arise.

Maven ignores issues related to upgrading a library by making optimistic assumptions about the newest version of that library. Projects that uses Maven must always be supported by active developers. There is a strong possibility that legacy projects, without developers, that use Maven will encounter a problem when run. This problem is due to the fact that the legacy project might not be compatible with the library versions Maven downloads. The reason this happens is because even if a project is no longer being updated, many of the libraries it uses could still be updated. Since Maven always uses the latest version of a library, legacy projects may be executed with library versions they are not compatible with. There is also an issue that arises when a library is updated but does not give any syntax or run time errors. Instead, the use of the new library version could lead to logical errors. These errors can be hard to diagnose for large projects that use and update multiple libraries at once. Being forced to run compatibility checks every time one of the libraries their project depends on is changed could be burdensome to the developer.

This case study in different library configurations is complementary to Maven. It is not in the best interest of a developer to be forced to update their project every time one of their libraries releases a new version. This is because mistakes can be introduced in the rush to ensure compatibility. There is the issue with legacy project dependencies being broken because their libraries are still being updated while the project is not. Legacy systems should not use Maven and should instead choose Ant, whose purpose is to assist in the build process for software projects [8].

2.2 Software Components

A way to conceptualize a project is through thinking of it as a set of components. Then a method discussed by Bauml and Brada, which focuses on component type reconstruction, can be used [4]. They studied the component dependencies that were created at build time and then checked these components for type compatibility. This was necessary because there is the potential for components to contain incompatible classes that are bound together, which results in run-time errors. Their solution was based on run-time compatibility checks. Instead of breaking down a program into the base project and associated libraries, projects are grouped into components. This grouping allows both behavior and dependencies to be measured between components [4].

The problem with this approach stems from trying to determine the resulting behavior when a library is updated. Sometimes there is a direct correspondence between a library change and a change in a component's behavior. At other times, instead of one component, multiple components can be influenced by a single library update [3]. Checking the behavior of each of these components can distort the results. This problem becomes clearer when two different library updates happen at the same time. Both of these updates result in direct behavior changes and both of these libraries are associated with a single component. The resulting behavior change might be wrongly characterized by the developer.

It is important to separate the base code and associated libraries to promote the easy characterization of behavior. Developers know their project's code better than code found in libraries. This case study focuses on changes in library versions to help developers better understand how libraries can affect their program's behavior.

2.3 Static Analysis

Static analysis of libraries can be used to determine if the interaction between the project and libraries can create a deadlock situation. The purpose of Shanbhags research in this topic is to find deadlocks situations that arise through library code changes [10]. This is in contrast to the purpose of this case study, which is to determine the behavior of different library configurations.

Deadlock situations are hard to find. A balance must be struck between over and under reporting potential deadlocks. Shanbhag uses the static analysis of libraries to determine if a potential deadlock exists within a combination of libraries. This case study uses static and dynamic analysis to determine what kind of behavioral changes are associated with changes in library version combinations. Both approaches attempt to find interesting behavior from examining libraries, however Shanbhag uses code analysis while this case study analyses the size and performance metrics from using different library configurations.

2.4 Mining Software Repositories

Another way to determine how to find the best combination of library versions is through mining the library usage patterns among users [7]. This case study is similar to to topic in that both try to answer similar questions but get answers through different data sources.

Two groups of stakeholders want to determine which is the most popular library version. The first is the library user. There are many reasons why the library user would want to go with the general trend. The trendy version might have the least number of bugs associated with it (i.e., it is stable). It could also be faster or more efficient. The other stakeholder is the library developer. The developer wants feedback on which version of their library is most popular so they can focus their efforts on that version.

Mining library usage is done through examining the different ways libraries are used among users. This is done through looking at the history of the library, what versions were used at what time and how long each version was used for. Other metrics that are kept track of are the most current library version and rollback information.

Rollbacks are characterized when large numbers of users switch to a specific library version and then switch back to their old version in a short time period. The number of users who switch to and away from that version is a good indicator of how reliable that specific version is.

This methodology groups users into two different categories: early adopters and late adopters. Early adopters prioritize using the latest version over stability while late adopters prefer stability over features. The mining techniques described in this paper benefits late adopters because early adopters tend to ignore trends and just go with the latest version.

Both approaches are measuring different characteristics. This case study tracks the:

- Execution of the libraries.
- Size of the trace.
- Test performance.

The mining approach tracks how:

- Long a library is used.
- Popular a library is (download wise).
- Often a rollback occurs.

At first glance, both approaches look similar in that they are trying to select combination of library versions. However, this is not true. This case study focuses on analyzing test artifacts while the mining approach focuses on analyzing the social aspects of library usage.

2.5 System Tracing

One way to determine what the effects of a code change in a project is through static and dynamic analysis of the code. This can be used to match code change with behavior changes. This approach was taken by Holmes and Notkin when they analyzed a project's trace and then grouped the results of their analysis into different categories [6]. This case study is similar to their research but differs in the data being collected and measured.

What Holmes and Notkins did is that they had two versions of a project and compared the versions to each other. The versions differed in that one version remained unchanged while the other had some code changes. What they were trying to do is determine what kind of behavioral changes resulted from a code change between these two systems. They had two different sources of data: static and dynamic traces [5]. They determined what was statically added or removed and then did the same for the dynamic trace by executing the system's test suite. This case study keeps the base project statically constant while changing the library configurations.

The aim of Holmes and Notkins is to associate behavioral changes from code changes by analyzing the base system while this case study looks at code changes in the associated libraries.

2.6 Software Reconnaissance

Rohatgi and Hamou-Lhadj proposed a method that uses static and dynamic analysis to find features in a system [9]. This technique is based on Software Reconnaissance [12]. Though both Rohatgi and Hamou-Lhadj and this case study analyze the trace, different questions are being answered.

Rohatgi and Hamou-Lhadj first get the feature trace based on the execution of the code. They perform impact analysis based on the static and dynamic data. The impact analysis step is critical to the success of feature identification. Impact analysis is used to determine how much of an effect a specific part of the system has on the rest of the system [9]. By doing this, they are able to rank each component based on how much it interacts with the rest of the system. The less interaction the component has with the rest of the system, the more strongly that component is associated with a feature.

Software reconnaissance, a technique proposed by Wild and Scully, is a method that utilizes traces to identify features [12]. The steps are to:

- Gather the different traces.
- Analyze those traces.
- Identify different components
- Associate each component with a feature using a ranking system.

This case study focuses on identifying behavior changes associated with code changes while software reconnaissance attempts to identify components that make up a feature.

2.7 Summary

Each of the related works brought up in the different sections of this chapter have been shown to not invalidate this case study. These works show that this topic is important because they demonstrate the need for a focus on problems associated with upgrading a library. Some of these problems are:

- Deciding which version to upgrade to.
- Selecting criteria for upgrading.
- Matching library changes to behavioral changes.

Chapter 3

Evaluation Framework

This chapter describes a framework for evaluating a case study. This framework is comprised of 10 steps and is divided into three sections:

- **Select baseline and alternate library configurations:**

S1 Determine the baseline.

S2 Select candidate library versions.

S3 Ensure compatibility of all libraries.

- **Execute candidate configurations:**

E1 Execute the baseline.

E2 Execute the different configurations.

E3 Execute the configurations from step E2 with a dynamic tracer[1].

E4 Ensure the integrity of all executions.

- **Analyze the collected data:**

A1 Compare configuration performance.

A2 Compare size of trace artifacts.

[1]https://bitbucket.org/rtholmes/dynamictracer

A3 Compare static and dynamic traces.

This case study focuses on steps S1 to A2. Step A3 is left for future work. Each of the three sections along with their inputs/outputs are shown in Figure 3.1.

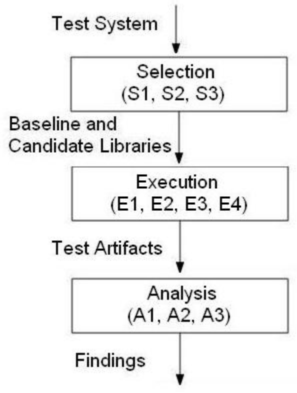


Figure 3.1: Framework overview with each section and their inputs/outputs.

3.1 Avoiding Incorrect Configurations

Once a system has been selected, the libraries that will be evaluated need to be identified. The purpose of step S1 is to specify a default configuration of library versions that will be compared against alternate configurations. This default configuration is often a known state (e.g., the current configuration), and will be referred to as the baseline.

Step S2 determines the candidate library versions which the alternate configurations are based on. Each of the candidate libraries need to be compiled with the test system to ensure that there are no broken static dependencies. This is to ensure that the candidate libraries do not introduce any build errors.

In step S3 the candidate libraries are checked to ensure that no behavior changes are introduced in the project. This is accomplished by individually substituting each candidate library with its alternate version located in the baseline. The project is run with the resulting library configuration and the test artifacts from that run are checked for errors. If the artifact contains an error this signifies the candidate library changes the behavior of the project and needs to be rejected.

3.2 Execute Configurations

After the baseline and candidate libraries are selected, alternate library configurations need to be created from the combination of baseline and candidate libraries. These configurations are then run and test artifacts are stored in different repositories so that each of these configurations can be traced back to the library versions they were created from.

In step E1 there are two separate sets of trial runs that are spawned from the baselines. The first set of runs corresponds to a baseline for the performance metrics. This is found by collecting the test artifacts associated with running the project using the baseline library configuration. The second set of runs corresponds to a baseline for the trace metrics. These runs use the same baseline library configuration as the previous set of runs with the addition of a dynamic tracer, called Inconsistency Inspector, being active.

In step E2 every possible library configuration that can be made from the combination of baseline and candidate library versions is run and the test artifacts associated with these runs are stored.

In step E3 each of the library configurations from Step E2 are rerun with a dynamic tracer active. The resulting trace artifacts are grouped into two categories: static trace artifacts and dynamic trace artifacts.

In step E4 all of the runs in steps E1-E3 are validated with respect to behavior. This step ensures that invalid runs are not included in the analysis section. Runs that contain errors in their test artifacts are invalidated because step S3 has demonstrated that the candidate libraries do not introduce any behavior changes in the project. If the interaction between libraries causes a behavioral change, step E4 verifies this behavior change was not a fluke. This is accomplished by having the invalidated run be redone and validated again. If this behavior is not a fluke, the associated configuration will be examined, separately, in the next section.

3.3 Data Analysis

After the artifacts from the set of runs have been validated, the analysis section begins. The initial analysis in Step A1 is done with respect to the performance metrics compiled from steps E1 and E2. Performance metrics can be used to identify which library version causes the project to run slower.

In step A2, analysis is based on the size of each of the trace artifacts from steps E1 and E3. Artifacts from the two trace categories, static and dynamic, are only compared to artifacts within the same category. A direct comparison between artifacts in different categories is not possible because static traces are based on all possible execution paths while dynamic traces are based solely on the test runs. The size of the trace artifact is important because this can be used to gauge the resource consumption related to each library. An assumption is made that there is a direct correlation between the size of the trace artifact and the number of calls recorded in that artifact. From step S3 and E4, no dependencies or behaviors were changed so step A2 is examining if the number of calls to accomplish the same result makes a difference.

In step A3, left to future work, the analysis would be based on the trace artifacts collected from steps E1 and E3. The reason this is not examined in this case study is because the result of the calls has the same outcome on the behavior of the project. Examining these calls does not provide useful information from a behavior standpoint. A sample of these calls can be seen in section D of the Appendix.

3.4 Goal

This case study aims to help developers identify problematic libraries. To evaluate these libraries, a project needs to be selected that uses multiple libraries and possesses an extensive test suite. This project also needs to have some distinguishing library version that can be detected using the proposed framework.

3.5 Voldemort

The developers of Voldemort[2] describe it as a key-value storage system. Voldemort was written and tested in Java and the project source code is contained in a Git repository. Voldemort's history shows that development first began on January 13, 2009. Over the course of three years, Voldemort's library folder grew to 30 different libraries while the test suite contains 768 tests.

Voldemort can be characterized as one of the more successful open source projects. It has a large and active user base that has downloaded it over ten thousand times. One of the big selling points for Voldemort is that it drives the LinkedIn social network[3]. Voldemort also has multiple volunteer developers. There are already several independent Voldemort

[2]http://project-voldemort.com/
[3]https://github.com/voldemort/voldemort/wiki/Powered-By

developers and when one of the key developers was contacted they mentioned how LinkedIn employees actively contributed to the project. For support, Voldemort possesses a wiki page, documentation, mailing list, IRC and forums.

3.5.1 Select Voldemort

Voldemort Version 0.90 snapshot 344 fulfilled all key characteristics needed for testing this framework. The main criteria for selecting a test system was that the system had to be written in Java, possess an extensive group of libraries and an accessible development history.

Voldemort is the ideal test system for this case study. It is written in Java and is compatible with Ant. Its ease of building minimizes problems with constantly changing library configurations and rebuilding the system.

The need for a large set of library dependencies stems from the fact that this case study is on library configurations. Inevitably, some library versions will not work. This happens because when a developer updates a library, they begin to code their project around the new library and stop supporting the old library so there is a good chance that the project is no longer compatible with older library versions

The system has to possess a long history. Since Voldemort's development is hosted on Github, its history of updates for associated libraries is accessible. Perusing the library history will lead to determining which libraries and what version of those libraries to select as candidates. Through examination of Voldemort's history one can determine that there had been a rollback on the `je` library[4]. This case study will focus on detecting this problematic library version.

3.5.2 Executing Voldemort

The process of setting up Voldemort to work was quite arduous and there were many undocumented issues that cropped up when trying to run Voldemort on a dedicated system with Ubuntu 10.04 LTS and an AMD Athlon 7750 with 3.25 GB of RAM. This can be seen in section B of the Appendix.

[4]https://github.com/voldemort/voldemort/commit/e52f4911b83ca2e29427b104e129afee40cc5cfc

3.5.3 Identifying Candidate Libraries

Once Voldemort was selected as the test system, the selection section of the framework was ready to begin. A criterion was needed for selecting the candidate libraries. The libraries needed to have been upgraded or downgraded inside of the project history and still worked with the 0.90 implementation of Voldemort.

The Voldemort library folder has 30 different libraries. The first task was to find which of these libraries had their version upgraded or downgraded by the developer. To figure this out, the Github history for the library folder had to be checked.

By examining the Voldemort library folder, 14 of the 30 libraries were pinpointed as having been changed at one time or another. Each of these libraries had their versions changed once, twice or even three times. This added up to 17 different changes. However not all of these 14 libraries would be suitable for the tests because the library:

- Was renamed for clarification.
- Is no longer compatible with the project.

Once the list of libraries had been compiled, each library had to be verified as compatible with the base project. To do this, each of these libraries had to be downloaded from the Apache repository. During this download step it was observed that one of the 14 libraries that had been switched was actually not an upgrade or downgrade but in reality merely a renaming to ensure that the version of the library was preserved in the name of the library. The Voldemort developers had decided to have a standard where the name of the library contains the library name and version. Then each library was individually switched into the Voldemort library folder and the system was built to ensure library compatibility. This process left nine different libraries but since one of the libraries had been upgraded twice there were 10 different candidate libraries that could be tested with. Table 3.1 shows the list of 17 changes and whether or not that change's corresponding library version was selected.

3.6 Methodology

Voldemort was used as the test system to gather data through executing the test suite with different library configurations. This data was then analyzed to find the sub-optimal library. This library had been determined through examination of Voldemort's history. The following is a list of all the programs needed to accomplish this purpose:

Table 3.1: Available Voldemort libraries

Base Library	Inserted Library	Outcome
avro-1.4.0.jar	avro-modified-jdk5-1.3.0.jar	✓
avro-1.4.0.jar	avro-1.3.0.jar	✓
colt.jar	colt-1.2.0.jar	RENAMED
google-collect-1.0.jar	google-collect-1.0-rc2.jar	✓
je-4.0.92.jar	je-4.1.7.jar	✓
je-4.0.92.jar	je-4.0.103.jar	X
commons-io-1.4.jar	commons-io-1.3.2.jar	✓
commons-lang-2.4.jar	commons-lang-2.1.jar	X
jetty-util-6.1.18.jar	jetty-util-6.1.6rc0.jar	X
jetty-6.1.18.jar	jetty-6.1.6rc0.jar	✓
junit-4.6.jar	junit-4.1.jar	X
libthrift-0.5.0.jar	libthrift-0.2.0.jar	X
libthrift-0.5.0.jar	libthrift-20080411p1.jar	X
slf4j-log4j12-1.5.6.jar	slf4j-log4j12-1.4.3.jar	✓
slf4j-api-1.5.6.jar	slf4j-api-1.4.3.jar	✓
log4j-1.2.15.jar	log4j-1.2.13.jar	✓
velocity-1.6.2.jar	velocity-1.5.jar	✓

- **Autotest-Single:** Gathers data for baseline.
- **Autotest-Multiple:** Gathers data for all library configurations.
- **Autotest-Multiple-Tracer:** Gathers trace data for different library configurations.
- **Fill in the Blank:** Re-execute any library configuration that resulted in failure.
- **Analyze Test Results:** Classifies the data into different buckets and compute statistics from those buckets.

When the test runs were being planned, library versions were initially switched manually. However this method is too cumbersome especially when there is a large number of configurations to test. A method was needed that would be able to gather and store large amounts of data from Voldemort test runs. This necessitated automation.

3.6.1 Autotest-Single

The goal of step S1 was to establish a baseline that could be compared against other configurations. A baseline is an initial configuration that all other configurations can be compared to[5]. For this purpose, the baseline would be the latest library configuration for Voldemort. Table 3.2 shows the candidate libraries and their versions that make up the baseline. This was chosen as the baseline because an assumption was made that the newest version would be the best version, performance-wise[6]. To create a baseline, a program would need to be coded that would be able to execute the Voldemort program multiple times without any user input. This program will fulfill step E1 and can be seen in Figure 3.2.

Table 3.2: Baseline Configurations

Library	Version
avro	1.4.0
google	collect-1.0
je	4.0.92
commons-io	1.4
jetty	6.1.18
slf4j-log4j12	1.5.6
slf4j-api	1.5.6
log4j	1.2.15
velocity	1.6.2

There are a few setup steps before the test artifacts can be stored from multiple runs. The location of where to store the artifacts and Voldemort installation had to be specified.

First Voldemort had to be built. This was done by using the `exec` command to send command line arguments. In this case the `exec` command sent the `ant build buildtest` argument into the command line. This would cause Ant to build the project. The algorithm then ensures that the project built without returning any errors.

Once the build has been verified as successful, the actual testing phase could begin. This was done by wrapping `ant junit` inside the `exec` command. This command would cause Ant to begin testing Voldemort with its jUnit test suite. The issues encountered with using `exec` are documented in section ?? of the Appendix.

[5]http://en.wikipedia.org/wiki/Baseline_(configuration_management)
[6]http://en.wikipedia.org/wiki/Upgrade

After the tests were done, the data had to be stored in preparation for the next run. This was done with the use of the `copy` command which was used to copy the test results folder inside the Voldemort distribution into the storage structure.

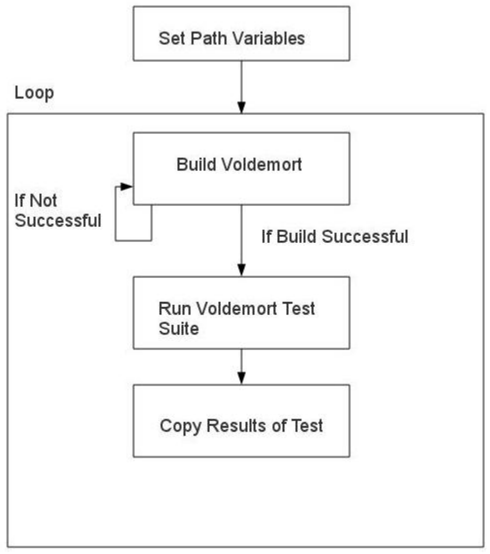


Figure 3.2: Structure of the Autotest-Single program.

3.6.2 Autotest-Multiple

Step S2's goal was to collect test artifacts for the runs that resulted from the different configurations of library versions that had been selected which can be seen in Table 3.3. Step E2 was fulfilled by the execution of the following algorithm, which will collect test artifacts from the different library configurations. This algorithm is shown in Figure 3.3.

To specify the configuration that had to be tested, there needed to be an array structure that would be able to represent this configuration. This array would have the following three cell pattern repeated ten times. The first cell in this pattern would contain either a zero or one. The second cell would contain the new library version that was being switched in. The third cell would contain the library version that was being switched out. The way the array worked is that each pattern began with a number and if that number was zero then the pattern would be considered off, which meant that library version would not be switched. However if the pattern began with a one the pattern would be considered on

Table 3.3: Library Configurations

Library	Version
avro	modified-jdk5-1.3.0
avro	1.3.0
google	collect-1.0-rc2
je	4.1.7
commons-io	1.3.2
jetty	6.1.6rc0
slf4j-log4j12	1.4.3
slf4j-api	1.4.3
log4j	1.2.13
velocity	1.5

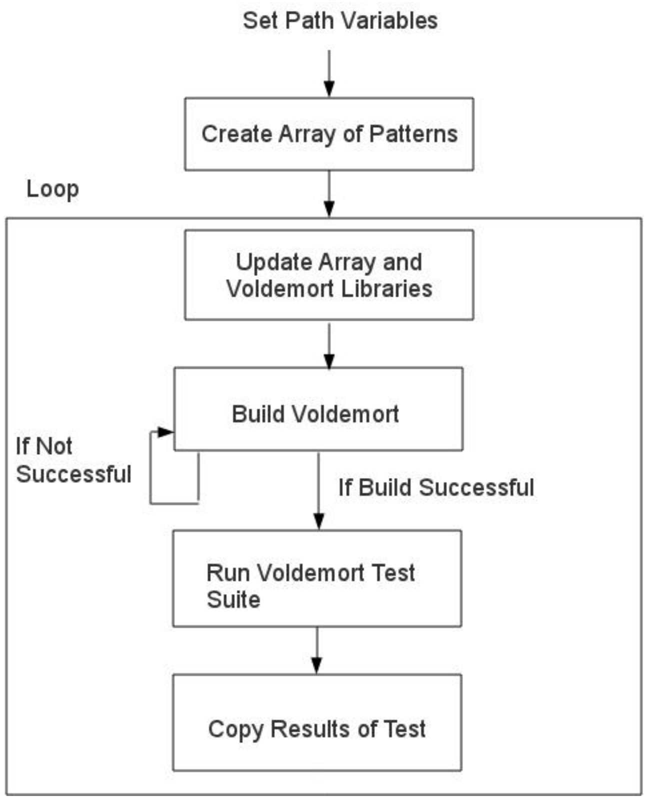


Figure 3.3: Structure of the Autotest-Multiple program.

so that library would have its version upgraded (or downgraded) to the specified version inside the pattern. Figure 3.4 shows this structure.

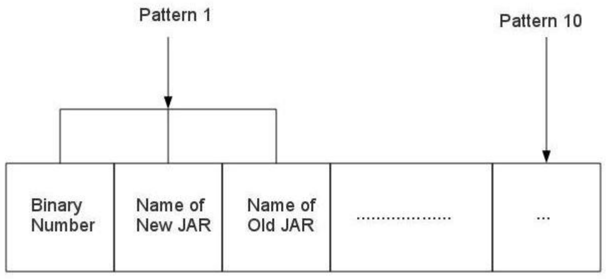


Figure 3.4: Structure of the array pattern described.

To collect test artifacts for all the possible library configurations of Voldemort, some sort of counting algorithm had to be specified and then implemented. There needed to be a loop to count, starting from the baseline case of zero all the way up to the configuration that contains all the candidate libraries which corresponds to 1023. This way each of the configurations could be tested in turn by counting from 0 to 1023. This algorithm started at zero and then stored a representation of zero in the array full of patterns. The libraries that make up each configuration can be either the candidate version or the baseline version. There are exactly 1024 configurations because there are 10 candidate libraries to test and 2^{10} is 1024.

Configurations would be chosen by converting an integer number into its corresponding binary number. The length of this binary number would then be calculated and padded on the left with 0's. If the binary number only had two digits then eight more 0's would be added so that this new binary number of length 10 would properly match up with the array. Each specific digit in this new length 10 binary number corresponds to one of the 10 patterns located in the array. The binary number would be parsed out and each pattern would now have a new digit at the start that was taken from the binary number. In the next run, the original integer number would be incremented by one and this process would start again. In this way, each run of the `Autotest-multiple` program would correspond to a different combination and all of the combinations would be represented. Based on the configuration specified in the array, the correct set of libraries were copied to the Voldemort folder. Issues with this program are documented in section C of the Appendix. Figure 3.5 shows the program used to ensure that a properly passed test suite was recorded and fulfills the goal for step E4.

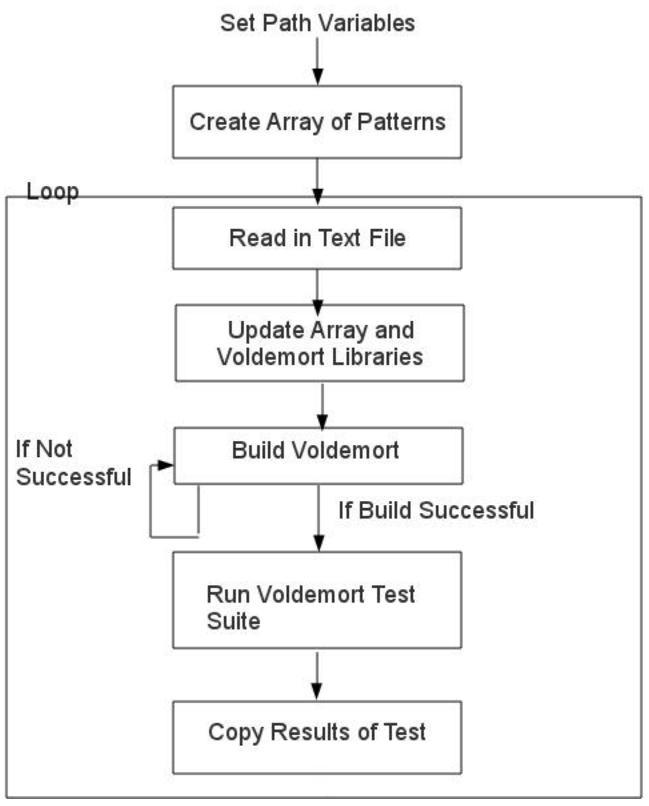


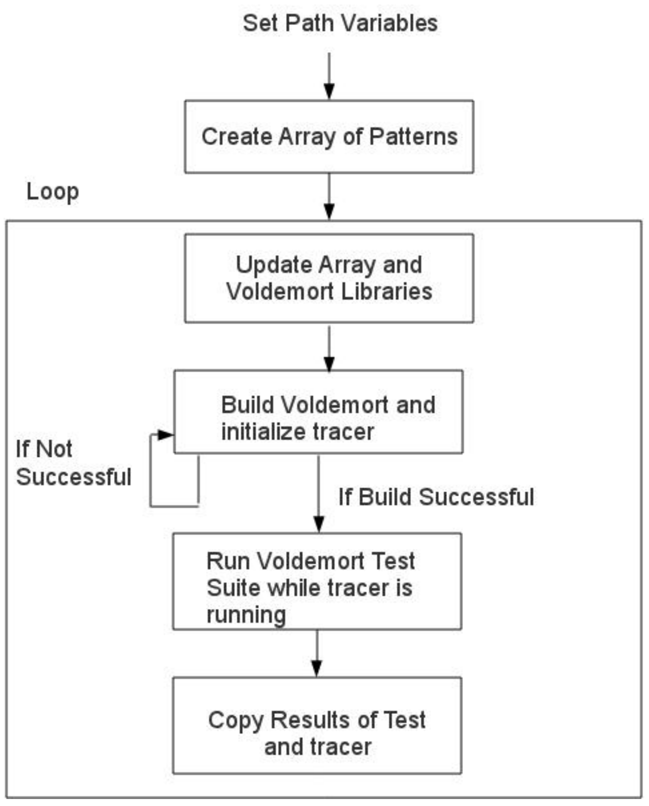
Figure 3.5: Structure of the Autotest program for ensuring run integrity.

3.6.3 Autotest-Multiple-Tracer

The candidate libraries that were selected in step S2 need to have a tracer collect trace artifacts from each of the combinations of the Voldemort test runs. After all the additions required by the previous program, the additions to include a tracer to fulfill step E3 were not as extensive (as can be seen in Figure 3.6).

Instead of building and testing Voldemort directly, now the tracer harness would be used to call Voldemort. Instead of issuing an `exec` command for `ant build buildtest` and `ant junit` these would be replaced by calls to the tracer with `ant clean build buildtest iiStatic iiWeave -Dii=` and `ant iiDynamic -Dii=`. The artifacts from this trace were then included when the Voldemort test folder was copied over.

The trace was separated into two XML files. This can be seen in how the files were named:



Set Path Variables
Create Array of Patterns
Loop
Update Array and Voldemort Libraries
Build Voldemort and initialize tracer
If Not Successful
If Build Successful
Run Voldemort Test Suite while tracer is running
Copy Results of Test and tracer

Figure 3.6: Structure of the Autotest-Multiple program with a tracer.

- `-static.xml`: Contains data from statically tracing the system.
- `-dynamic.xml`: Contains data from executing the test suite.

From this list, one can see that both a static and dynamic trace were stored for each library configuration.

3.6.4 Autotest-Size

Step A2 needed an analysis of the trace results which required iterating through each separate static and dynamic XML file and compare their size within the set of static or dynamic artifacts. The reason this was done was to see if different library configurations led to drastic differences in the number of method calls made for the duration of each run. This task was shortened by calling upon the code used to specify the counting protocol in the previous Autotest programs. The structure of this newly modified program can be seen in Figure 3.7.

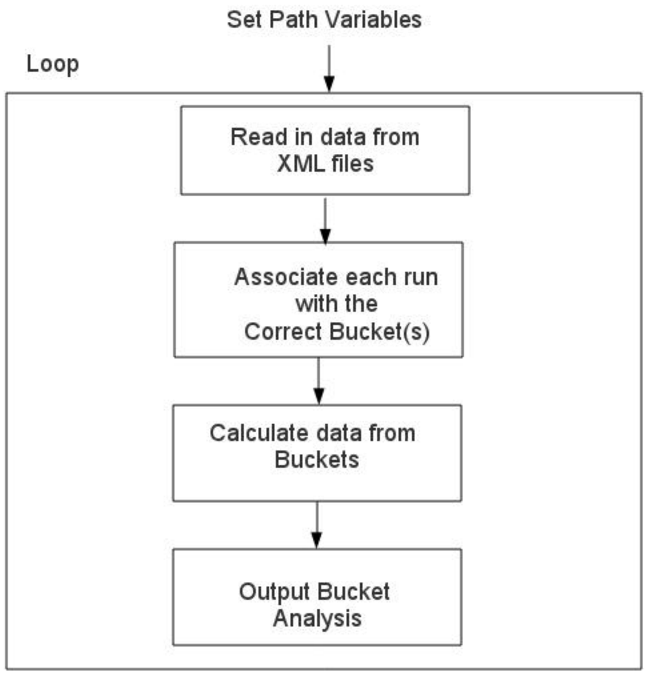

Set Path Variables
Loop
Read in data from XML files
Associate each run with the Correct Bucket(s)
Calculate data from Buckets
Output Bucket Analysis

Figure 3.7: Structure of the Autotest-Size program.

The program had to iterate through all of the static and dynamic trace artifacts and store the size of each artifact along with the label for the corresponding run. Separate arrays would be used to store the static and dynamic traces along with their identifying labels. Each trace was associated with the candidate libraries that made up each run configuration. The next step would be to compute the average size of the trace for each of the libraries.

3.6.5 Analyze Test Results

By adapting the Devulge program[7], whose structure can be seen in Figure 3.8, three things could be accomplished:

- Show the performance breakdown by library.
- Calculate the minimum, maximum, mean and standard deviation of the breakdown.
- Determine the location of jUnit test results that contained errors.

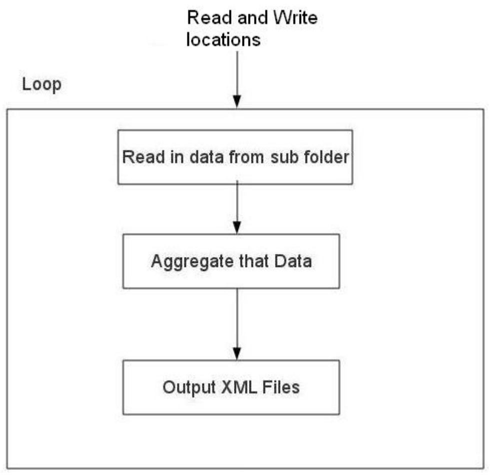


Figure 3.8: Structure of the Devulge program.

The Devulge program works by specifying a folder structure and then recursively goes through that folder hierarchy and catalogs all of the XML files it encounters. This is done by taking in a command line argument that specifies the location of the folder and where the output should be stored. Then the program goes through each of the subfolders which correspond to a different combination of library versions and consolidates that data into an XML file. The result is that there is an XML file corresponding to each combination of libraries. The structure of the modified Devulge program can be seen in Figure 3.9.

The additions made to the Devulge program fulfilled the goal of step A1 and involved being able to classify each run by the libraries that made up that specific configuration. The data had to be displayed in a graphical format so storage arrays and counters were

[7] https://bitbucket.org/divamjain/devulge/overview

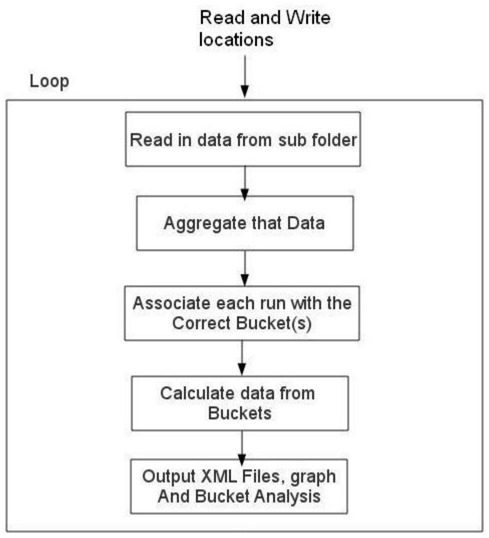

Read and Write locations
Loop
Read in data from sub folder
Aggregate that Data
Associate each run with the Correct Bucket(s)
Calculate data from Buckets
Output XML Files, graph And Bucket Analysis

Figure 3.9: Structure of the modified Devulge program.

created that would correspond to each library, as can be seen in Figure 3.13. When a run was first read in, that run was associated with the arrays that correspond to the libraries which had their versions changed. What the array specifically stored was the total time it took to execute that specific combination of libraries. This was possible because the location the run was stored in contained this data in the name of the folder itself. After all the runs were finished being cataloged, the program then looped over the resulting arrays filled with the times and cataloged them in a graph. The following calculations were made for each of the libraries:

- **Minimum:** Time for the fastest executing run.
- **Maximum:** Time for the slowest executing run.
- **Mean:** Average of all the run times.
- **Standard Deviation:** Range most data runs were in relation to the mean.

The last step was to output both the graph and the different time metrics that were computed.

To determine if an error had occurred in one of the test runs, the Devulge program was modified so that it would display the error and specify the run associated with that error so the correct library configuration can be rerun.

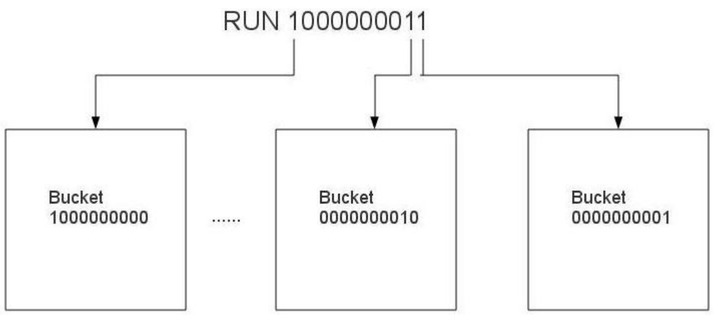


Figure 3.10: Structure of the buckets used to contain run data.

3.6.6 Storage Structure of Data

The sheer volume of data that was collected required an explicit organizational scheme to store everything so that the data could be accessible and categorize able. The issue is running the same tests repeatedly, so test from one run to another had to be distinguishable. The test artifacts being stored originated in the Voldemort test folder. This directory structure is shown in Figure 3.11.

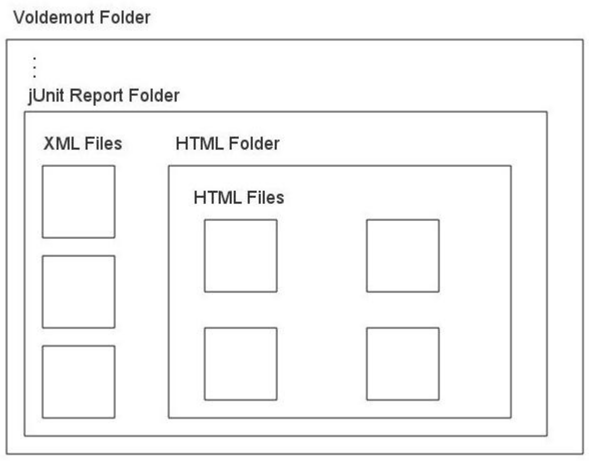


Figure 3.11: Directory structure of the Voldemort reports folder.

Each run has a structure containing both XML and HTML files. The reason why the information is duplicated is that the HTML file provides an easy way to review the results of the test run while the XML file contains the information in an easily parsable format.

Each run had to be uniquely identified so that when the time came to analyze the data in terms of libraries, the data could be associated with each library it is related to. This directory structure is shown in Figure 3.12.

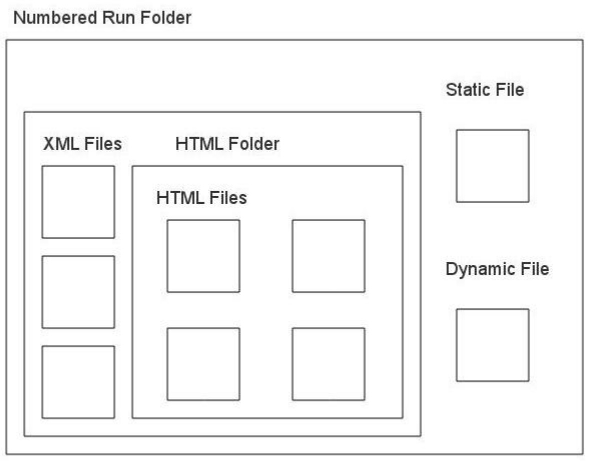


Figure 3.12: Directory structure used to store the run data.

A base folder was created for each test run using a naming convention to represent what library configuration is stored in that folder. Each time the data collection program ran, it iterated through the different library configurations by using an array with 10 patterns composed of three cells. The naming convention used specified that the binary number be taken from the first cell of each pattern and concatenated together. This new 10 digit binary number would be the name of the base folder for that specific run. Since each one digit binary number represented whether a specific library version was switched during the run, this new 10 digit binary number would be a unique identifier for each run (since every possible combination of library versions only happened once).

The trace artifacts also had to be stored. While running the tracer, all the results were stored into two XML files, each of which was associated with either static or dynamic traces. The way they were sorted was to extend the naming convention for the base folder to each of the xml files so that the file name would contain a 10 digit binary number followed by either static or dynamic to differentiate the two trace types.

The trace files were taken from their location in the Voldemort folder and copied into the storage structure. All of the static and dynamic trace files originated inside of the report folder that was created in the Voldemort base directory. These specific trace files were then copied directly into the folder that was associated with the current configuration.

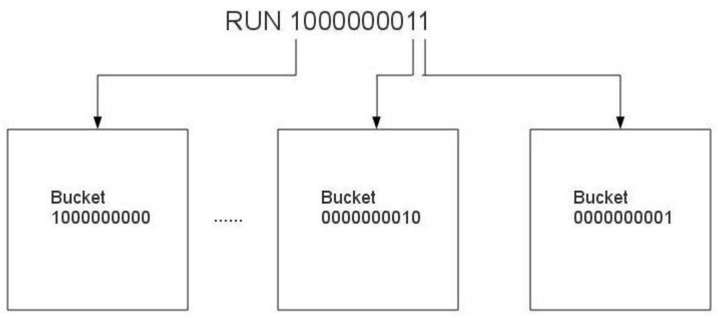


Figure 3.13: Structure of the buckets used to contain run data.

3.7 Data Gathering

The four different goals that needed to be fulfilled while collecting data were:

- Identify tests that needed to be removed.
- Compute a baseline.
- Collect test artifacts from all possible library configurations.
- Collect trace artifacts from all possible library configurations.

3.7.1 Identify Tests for Removal

Once Voldemort and its test suite were running, the tests that would be used in the experiment had to be identified. All the tests were not used because many of the Voldemort tests returned errors. This is not surprising considering that Voldemort is still under development. The developers of Voldemort do not have the resources to track down every bug nor test on every system. Some of the tests did not even work with the current Voldemort installation. However this did not necessarily mean that they should be removed. For example, consider the MySQL associated tests. Originally, all of the tests that were for MySQL did not work because the MySQL library was not included in Voldemort. After talking to the developer, it was discovered that MySQL was included for one of the developers side projects that was not documented in the main Voldemort release. The only way for these tests to work was for the MYyQL library to be included. This was the only addition made onto the Voldemort base library configuration. The Voldemort release was continuously run and tests that fluctuated in their results were removed. As the Voldemort test suite was run over and over again, other tests were identified and removed that hanged or returned errors.

The use of jUnit allowed for the quick removal of all necessary Voldemort tests. The way this was accomplished was through the use of the `@Ignore` annotation with the tests that needed to be ignored. Occasionally, an entire test class had to use the `@Ignore` annotation when the class reached the point where all of the tests inside of it had been identified as returning unstable results. The reason why a class was not immediately identified for the `@Ignore` annotation is that some classes did not have all their tests included with the `@Ignore` annotation until several hundred runs had happened from when the first test was ignored to the last.

Approximately 13% of the test cases from the Voldemort test suite were removed. This means out of 768 tests, 670 remained for the remaining three steps. Table 3.4 shows the test class and shows the total number of tests in that class along with the number of tests removed.

Table 3.4: Tests Removed

Class	Original	Removed
AdminRebalanceTest	12	10
AdminServiceFilterTest	6	6
RebalanceTest	12	12
AdminServiceBasicTest	32	32
StoreSwapperTest	4	4
EndToEndTest	8	6
RebootstrappingStoreTest	4	3
ServerStoreVerifierTest	2	2
GossiperTest	2	2
RedirectingStoreTest	6	6
StreamingSlopPusherTest	3	3
DataFileChunkSetIteratorTest	1	1
ReadOnlyStorageEngineTest	78	9
ServerSideRoutingTest	4	2

3.7.2 Default Library Configuration

For step E1 in the framework, a baseline was needed for analysis so the Voldemort library configuration that was specified in Table 3.2 was run. This configuration was run with and without a tracer so that there would be a baseline for the performance and trace metrics. This resulted in two different repositories:

- One only contained jUnit test reports.
- The other had both jUnit test reports and trace files.

3.7.3 Different Library Configurations

To collect data on the different library configuration needed for step E2, the algorithm substituted a different configuration each time the Voldemort test suite ran. The resulting jUnit test folder was then stored. This process can be seen in Figure 3.3.

3.7.4 Combination with Tracer

To collect all the trace data associated with the different library configurations for step E3, hooks had to be placed into the test framework. This would result in the execution of the test suite returning both a jUnit test folder and trace files. This is shown in Figure 3.6.

3.7.5 Consolidation of Results

Since Voldemort is a work in progress, not all aspects of it are stable and work 100 % of the time, which is why step E4 is necessary. Many of the tests kept returning errors every so often. The baseline library configuration in Table 3.2 was used to identify these faulty tests.

Once the tests that had been identified as unstable were found, it was time to begin collecting test artifacts associated with the baseline configuration. The next phase was to collect test data on every possible library configuration. During this step, some tests were still found to fail that had not been identified in the previous iteration, but steps had to be taken to ensure that this failure was not from the combination of the candidate library versions. To do this, those runs were identified and then that specific configuration was rerun. The results were examined again to determine if there were any errors. This ensured that every combination of library versions was viable and did not fail any of the tests. The final data runs to be collected had a tracer active while each library combination was tested.

3.8 Summary

The framework described in this chapter shows the way in which a sub-optimal library version can be found. The preliminary step for this process is to identify candidate and baseline library versions to work with. The next step is to execute different configurations based on a mixture of candidate and baseline libraries. The resulting test artifacts are stored for future analysis.

Chapter 4

Analysis

This Analysis has four sections:

- Characterize the baseline and candidate libraries.
- Specify the baselines that will be compared against.
- Analyze performance of the other configurations with respect to the performance baseline.
- Analyze the size of the trace artifacts generated by the other configurations with respect to the trace size baseline.

4.1 Versions to Compare

This analysis in this section corresponds to the artifacts gathered from steps S1 and S2. This provides an overview of the baseline configuration along with all of the different candidate library versions that make up the possible configurations. Table 4.1 shows this relationship by having the baseline library listed before its candidate library counterpart. Something to note is that there are three entries for `avro`. The first entry is the baseline while the other two entries are candidates.

Table 4.1: Overview of Library Versions

Library	Version	Size {bytes}
avro	1.4.0	568,780
avro	jdk5-1.3.0	328,427
avro	1.3.0	338,945
google-collect	1.0	577,311
google-collect	1.0-rc2	575,468
je	4.0.92	2,132,019
je	4.1.7	2,146,304
commons-io	1.4	109,043
commons-io	1.3.2	87,776
jetty	6.1.18	525,497
jetty	6.1.6rc0	492,432
slf4j-log4j12	1.5.6	22,338
slf4j-log4j12	1.4.3	15,345
slf4j-api	1.5.6	22,338
slf4j-api	1.4.3	15345
log4j	1.2.15	391,834
log4j	1.2.13	358,180
velocity	1.6.2	420,975
velocity	1.5	392,124

4.2 Baseline Analysis

There are three different sets of baselines, which came from step E1, that are being used:

- **Time Baseline**
- **Static Size Baseline**
- **Dynamic Size Baseline**

The time baseline will be compared with the data runs that correspond to the different library configurations. The results of this baseline were analyzed to determine the maximum, minimum, mean and standard deviation. They are shown in Table 4.2.

The static size baseline will be compared to the static files that correspond to the different library configurations. An analysis of the static baseline metrics can be found in Table 4.3.

Table 4.2: Time Baseline

Function	Seconds
max	192
min	139
mean	147
standard deviation	4

Table 4.3: Static Size Baseline

Function	Size {bytes}
max	13,437,178
min	13,437,178
mean	13,437,178
standard deviation	0

The dynamic size baseline will be compared to the dynamic files that correspond to the different library configurations. An analysis of the dynamic baseline metrics can be found in Table 4.4.

Table 4.4: Dynamic Size Baseline

Function	Size {bytes}
max	4,469,558
min	4,371,402
mean	4,434,848
standard deviation	27256

4.3 Time Analysis

Step A1, the time analysis, will come from the performance data that was computed from the results of step E2 and will be compared against the baseline data from step E1. The results were grouped into ten different buckets that corresponded to each of the ten different libraries that were tested with. For instance, every time that a configuration was executed, that run could be associated with up to ten different buckets depending on whether or not the run used the candidate libraries that corresponded to each of the ten buckets.

The performance of the candidate libraries was uniform for every case except one. Each of these libraries had a mean of 148 seconds. The time baseline mean clocked in at 147 seconds. There was not a drastic difference in performance between the means of the candidate and baseline libraries. In fact this one second slowdown can be attributed to the inferior time performance of the one outlying candidate library, `je`. The `je` library had a mean of 151 seconds which was most likely a large factor as to why the mean for the candidate runs were one second slower than the time baseline runs. The candidate runs used a combination of different library versions in executing Voldemort. Half of these combinations included the `je` library which gave slower performance so this would lower the perceived performance of all the other libraries. The run times for the candidate configurations can be seen in Table 4.6

Taking a closer look at `je`, it is interesting to note that its performance is still within the standard deviation of 4 seconds. This means that from a performance viewpoint, the `je` library blends in with the other libraries making it difficult to detect as the sub-optimal library. This explains why the Voldemort developers did not realize their mistake until two months had passed[1][2]. In contrast, the time it took to run this portion of the case study was approximately 42 hours.

The other case to consider is for runs that fall outside the standard deviation range. One of the reasons why data for the longest and slowest execution of Voldemort was collected is to look for interesting edge cases in terms of configurations that led to interesting behaviors. What was found was slightly disheartening and is a testament to Voldemort still being in the development stages. For the configuration runs, the slowest run clocked in at 174 seconds while the fastest run came in at 137 seconds as can be seen from Table 4.5. When compared to the time baseline where the slowest was 192 and fastest was 139 this means two things. For the slowest configuration, the time baseline was actually slower so one cannot really look at this as an interesting development. However the fastest configuration was two seconds faster than the fastest time baseline. One thought to consider is that this combination would not include the slowest library, `je`. However this was a mistake because the fastest configuration was 1001110111. Since `je` is the fifth library in the array of patterns, since there is a one in the fifth position. On the other end of the spectrum, the slowest combination was 1111110001. This also contained the `je` library which is interesting because this means that both the fastest and slowest configurations contained the, on average, worst performing library. This leads to the belief that since Voldemort's performance in terms of time was so unstable, as one can see from both the time baseline

[1]https://github.com/voldemort/voldemort/commits/8fe2c0826d4b2e06ea265c763632586288eba8d1/lib/je-4.1.7.jar

[2]https://github.com/voldemort/voldemort/blob/master/lib/je-4.0.92.jar

and the combination run results, one cannot read too much into any of the data points outside of the standard deviation.

Table 4.5: Time Metrics

Function	Seconds
max	174
min	137
mean	148
standard deviation	4

Table 4.6: Time Analysis

Library	Seconds
avro-modified-jdk5-1.3.0.jar	148
avro-1.3.0.jar	148
google-collect-1.0-rc2.jar	148
je-4.1.7.jar	**151**
commons-io-1.3.2.jar	148
jetty-6.1.6rc0.jar	148
slf4j-log4j12-1.4.3.jar	148
slf4j-api-1.4.3.jar	148
log4j-1.2.13.jar	148
velocity-1.5.jar	148

4.4 Trace Size Analysis

Step A2, analyzing the size of the trace artifacts, was computed using the runs from step E3 and compared them with the baseline trace runs from step E1. The reason these runs had to be separate from the original combination runs was that the tracer added quite a bit of time for each run and this was not a constant amount of time that could be computed. Instead of ten buckets, one bucket per library, the tracer runs were grouped into twenty buckets. These buckets corresponded to the static and dynamic trace associated with each of the ten libraries.

Across all the buckets for static traces, the mean size is between 13.2-13.4 million bytes. At first glance at Table 4.7, none of the libraries distinguish themselves. However, it is known from the time analysis that the candidate `je` library had the worst performance but here the static trace show that it is average in terms of the number of calls made. This means that calls made with the candidate version of `je` take longer to accomplish rather than just being more numerous, which cannot be computed from the trace artifacts. When the static trace artifacts for the candidate libraries are compared against the baseline it is interesting to note that the baseline static artifacts are stable in size with a standard deviation of 0. Upon further examination of this phenomenon, it was discovered that each configuration which corresponded to only a single candidate library being switched in (ie 0000000001, 0000000010...etc) had the same size except the one corresponding to `google1.0rc2`. It is only during configurations which include the candidate version of the Google library that the static trace sizes begin to vary. Potentially, this instability in the number of calls made with the `google1.0rc2` library led to the developers choosing another version. Checking the Voldemort repository[3] showed that this library had been upgraded because of issue 215. However upon further examination of the Voldemort bug database[4], it was discovered that issue 215 was an upgrade due to API changes. This means that the `google1.0rc2` library was upgraded because of documentation issues, not because it was problematic.

The dynamic trace file sizes range from 4.3-4.4 million bytes as can be seen in Table 4.8. The dynamic trace artifacts associated with the candidate `je` library are 4,428,101 bytes large, which is average. The artifacts from the other candidate libraries range in size from 4.3-4.5 million bytes. When compared back to the baseline of 4,406,145 bytes there is no distinguishing attribute among the dynamic traces.

The three different methods used for analyzing the artifacts were time, static and dynamic analysis. From these three, time analysis was able to indicate the problematic candidate library. Measuring the number of static calls was not able to pinpoint the faulty library, in fact, this pinpointed the wrong library. Measuring the size of the dynamic artifacts also did not provide any enlightenment. Only through time analysis was the `je` library found to be problematic.

4.5 Summary

The analysis in this chapter shows that determining a sub-optimal library requires extensive performance-based testing. The reason why executing a small number of test runs (less

[3]https://github.com/voldemort/voldemort/commit/b6b77655f69b61195b64b654317cac36fe7f593c

[4]http://code.google.com/p/project-voldemort/issues/detail?id=215&can=1&sort=-id&start=100

Table 4.7: Static Trace Analysis

Library	Size {bytes}
avro-modified-jdk5-1.3.0.jar	13,279,415
avro-1.3.0.jar	13,436,878
google-collect-1.0-rc2.jar	13,305,371
je-4.1.7.jar	13,253,179
commons-io-1.3.2.jar	13,436,878
jetty-6.1.6rc0.jar	13,358,149
slf4j-log4j12-1.4.3.jar	13,253,174
slf4j-api-1.4.3.jar	13,279,418
log4j-1.2.13.jar	13,226,934
velocity-1.5.jar	13,358,155

Table 4.8: Dynamic Trace Analysis

Library	Size {bytes}
avro-modified-jdk5-1.3.0.jar	4,387,997
avro-1.3.0.jar	4,436,866
google-collect-1.0-rc2.jar	4,397,005
je-4.1.7.jar	4,382,622
commons-io-1.3.2.jar	4,437,100
jetty-6.1.6rc0.jar	4,409,894
slf4j-log4j12-1.4.3.jar	4,375,510
slf4j-api-1.4.3.jar	4,392,212
log4j-1.2.13.jar	4,371,189
velocity-1.5.jar	4,410,547

than 100) will not suffice is that the wide variance in the performance cannot be properly normalized without a large number of runs. The performance based tests identified the sub-optimal candidate library, `je`, even though its performance drop was less than the standard deviation found in the test runs. This was done through collecting and measuring test artifacts found from the large number of data runs recorded.

Chapter 5

Scenarios

This chapter describes possible threats to validity of our results; that is, circumstances that could lead to inaccurate results and the steps taken to mitigate them. There are also additional case studies that are needed to cover other experimental conditions.

5.1 Internal Validity

Internal threats to validity are inherent issues with decisions the researcher takes within their experiment. Potentially, these issue could distort the results [13]. This distortion could potentially happen with:

- The test suite.
- Decisions on what candidate libraries to select.
- The baseline.
- Performance metrics.

One of the biggest impacts upon the data results was the fact that approximately 13 % of the tests within the Voldemort test suite were removed. If all of the tests had remained, potentially this could have led to more accurate performance metrics.

Another issue with using the Voldemort test suite is that it might not be testing what is needed to be tested. The test suite was written to test the Voldemort project and not the libraries used by Voldemort. This means that potentially there could be bias against some of the libraries. For example, if Voldemort relies heavily on one library over another, then potentially this could bias the results in favor of that library.

There needed to be a method to select which library versions would be evaluated as candidates. Instead of evaluating every possible version of the Voldemort libraries, only versions used previously by Voldemort were considered. Evaluating more library versions could have led to a wider range of results.

Selecting a good baseline is crucial because the baseline is what comparisons will be made against. The way that the baseline was chosen was to use the developer library versions that were included in the Voldemort installation. This method allowed the removal of any built in bias that could have been introduced by selecting library versions that favored the framework. However there could have been a better baseline to choose from.

Another internal threat had to do with time measurement. While doing the trial runs, executing the test suite inside Eclipse lengthened how long it took to run the test suite. The problem is that each program running takes up resources in the environment and this has the potential to slow down the execution of tests. The way the time was recorded does not take this into consideration. If the system became overwhelmed with tasks needing to be executed and hung during testing, this hang time would be included in how long it takes that specific test to be run. One step was to limit the usage of the system while collecting data results. This was done by only running the Voldemort test suite at night when the system was not in use. This would limit resource consumption and ensure most system resources were being devoted towards Voldemort.

The decisions taken to mitigate the internal threats were couched in trying to preserve the developer vision of the system while at the same time taking limited resources into consideration. Having all of the tests that the developer used would have been ideal. Though potentially these tests could have favored one library over another. Testing every possible library version could have led to interesting results. There could have also been a better baseline to choose from. Several factors could have led to distortions in the performance metric.

5.2 Additional Case Studies

Alternate case studies are needed to fully examine the issue of problematic libraries. These case studies will involve different:

- Types of applications.
- Operating Systems.
- Test suites.
- Combination of libraries.

Chapter 6

Conclusions

6.1 Findings

This case study was able to autonomously determine the sub-optimal `je` library, Version 4.1.7, through performance based time analysis. This version was newer than the baseline of 4.0.92 which goes to show that a newer version is not necessarily the best version. This means that the Maven solution of newer is better is not necessarily the best approach for a developer to take.

Performance analysis cannot be done in half measures. Programs do not run the same every time they are executed. Performance metrics must be measured after a pattern is established. For Voldemort Version 0.90 snapshot 344, this pattern was established after hundreds of runs. In fact, the configuration approach used in this case study meant that Version 4.1.7 was executed in 512 different configurations. Another advantage to using configurations is that multiple libraries can be tested at the same time.

6.2 Future Work

There are many other topics that this research can lead to. They can be grouped into two different fields: libraries and tests. Future work into the libraries field would be focused on how to extend the conclusions drawn towards different library configurations. The tests field extends this research to consider different test scenarios. Both fields have great potential.

6.2.1 Libraries

Within the library field one must consider how to organize a library and what kinds of comparisons to make between different versions of a library made up of the same libraries. Would it be beneficial to make distinctions beyond the relationship between base system and associated library? Could a more effective organization be based on what a certain subset of libraries did and then compare within or between groups of libraries? One of the big threats to the usage of this research was Maven which automatically selected the latest version of a library. Could there be some truth to always using the latest version? It might be beneficial to delve into this matter by comparing old versions of libraries to their newer counterparts and seeing if there is a trend. A limitation encountered while designing the experiment was being able to select only a few libraries to be tested. It is not possible to test all versions of every library used in Voldemort. However this left out a great deal of potentially useful and interesting data that should be looked at. Finally, the conclusions were only for libraries used in Java but there are other popular programming languages out there that deserve consideration.

6.2.2 Tests

There are many test scenarios that were not run. The first would have been to run an in-depth analysis of the static and dynamic trace, step A3 of the framework, with a specific focus on the `je` and `google` library. It would have been useful to collect data based on the results of executing tests for multiple systems. Though one could not necessarily compare the data directly between systems, it would have been useful to test whether the conclusions held under the alternate systems. Another variable is that library usage differs between small and large systems. It could have been interesting to determine if the size of the project could be used to draw conclusions. Another issue is the makeup of the project. Is there a difference in systems that were written by multiple developers verses one written by a single developer?

APPENDICES

Appendix A

Issues with running Voldemort

This research requires running tests on the Voldemort system, something that normal users do not do. Many problems were encountered that only someone with a knowledge of the actual Voldemort system would be able to solve.

There were also some problematic system issues. One of these was that Voldemort required a larger number of open files that was set by default by Ubuntu so this number had to be increased. Another problem was that since Voldemort was constantly run, every run of Voldemort left behind artifacts inside the temporary folder. After a few hundred runs, this folder became filled and would no longer allow any more files to be created inside of it. The solution was to do a clean of that folder every so often or restart the computer.

While executing the entire Voldemort test suite, the Mysql tests were failing every time. This was because Voldemort did not come with the necessary Mysql library, so these had to be separately installed. Another issue was that even after installing the Mysql library the tests were still failing because the Mysql database was not initialized[1]. After this was done, all of the Voldemort tests ran and passed at least once. Many of the remaining tests had to be commented out because they failed every so often.

[1] http://groups.google.com/group/project-voldemort/browse_thread/thread/1be4d88c73a10742

Appendix B

Issues with Exec

An issue encountered during this process is that every so often the `exec` command hangs. The way to solve this is by ensuring the buffers are all flushed including input, output and error buffer. This is done by having a while loop iterate and print everything within these buffers while the program waits for the `exec` command to finish.

Appendix C

Issues with Autotest-Multiple

This process was not without issues. Every couple hundred runs, a newly created library proved to be corrupt so steps had to be taken to ensure that data from the corrupt library was not recorded. The way this was tested for is through the `ant build buildtest` command. This was done by parsing the results of the command and looking for a line that contained BUILD SUCCESSFUL. If this line was not in the output then that meant for some reason or another the build failed so that build would then be rebuilt and this process would be attempted all over again until the success string was successfully detected. Another case to consider was when library corruption did not lead to build failure. This was dealt with by comparing the md5 hash of the individual libraries in the current library configuration to their original counterparts where they were copied from. If these hashes were not identical, the libraries would be copied again until the hashes matched. The last case to consider was when there would be a system hiccup that affected the Voldemort test data. Two examples of this were when the system ran out of memory and when the temporary folder Voldemort did its computations in was full. This case is covered through detection of the success string. These failures, though not common, would have led to drastically different results.

Appendix D

Static Trace File

...

```
    call s="voldemort.server.socket.SocketServerSession.run()"
    t="java.lang.StringBuilder.init()"/
    call s="voldemort.server.socket.SocketServerSession.run()"
    t="org.apache.log4j.Logger.info(java.lang.Object)"
    call s="voldemort.server.socket.SocketServerSession.run()"
    t="voldemort.server.protocol.StreamRequestHandler.handleRequest
    (java.io.DataInputStream, java.io.DataOutputStream)"
    call s="voldemort.server.socket.SocketServerSession.run()"
    t="java.net.Socket.close()"
    call s="voldemort.server.socket.SocketServerSession.run()"
    t="java.lang.Long.valueOf(long)"
    call s="voldemort.server.socket.SocketServerSession.run()"
    t="java.io.DataOutputStream.flush()"
    call s="voldemort.server.socket.SocketServerSession.run()"
    t="java.lang.StringBuilder.append(java.lang.Object)"
    call s="voldemort.server.socket.SocketServerSession.run()"
```

```
t=”org.apache.log4j.Logger.isTraceEnabled()”
call s=”voldemort.server.socket.SocketServerSession.run()”
t=”java.net.Socket.isClosed()”/
...
```

References

[1] T.J. Andersen and L.E. Amdor. Leveraging maven 2 for agility. In *Agile Conference, 2009. AGILE '09.*, pages 383–386, Aug. 2009.

[2] Microsoft Corporation. Windows Update. http://windowsupdate.microsoft.com.

[3] F. Barbier and N. Belloir. Component behavior prediction and monitoring through built-in test. In *Engineering of Computer-Based Systems, 2003. Proceedings. 10th IEEE International Conference and Workshop on the*, pages 17–22, April 2003.

[4] Jaroslav Bauml and Premek Brada. Reconstruction of type information from java bytecode for component compatibility. *Electronic Notes in Theoretical Computer Science*, 264(4):3– 18, 2011. Proceedings of the Fifth Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2010).

[5] Reid Holmes and David Notkin. Identifying Opaque Behavioral Changes, 2011. In Proceedings of the International Conference on Software Engineering (ICSE). 2011. 371-380.

[6] Reid Holmes and David Notkin. Identifying program, test, and Environmental changes that affect behaviour, 2011. In Proceedings of the International Conference on Software Engineering (ICSE). Research Demonstration. 2011. 995-997.

[7] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. Mining trends of library usage. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, IWPSE-Evol '09, pages 57–62, 2009. ACM.

[8] The Apache Ant Project. Apache Ant, 2012. http://ant.apache.org/index.html.

[9] Abhishek Rohatgi, Abdelwahab Hamou-Lhadj, and Juergen Rilling. An approach for mapping features to code based on static and dynamic analysis. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 236–241, 2008.

[10] V.K. Shanbhag. Deadlock-detection in java-library using static-analysis. In *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*, pages 361–368, Dec. 2008.

[11] D. Spinellis. Software builders. *Software, IEEE*, 25(3):22–23, May-June 2008.

[12] Norman Wilde and Michael C. Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[13] Dr. Patricia A. Zapf. Validity: Internal and external, 2005. http://web.jjay.cuny.edu/p̃zapf/classes/CRJ70000/Internal and External Validity.htm.