ABSTRACT

Title of dissertation: COLLABORATIVE TESTING ACROSS SHARED SOFTWARE COMPONENTS

Teng Long, Doctor of Philosophy, 2015

Dissertation directed by: Professor Alan Sussman
Department of Computer Science

Large component-based systems are often built from many of the same components. As individual component-based software systems are developed, tested and maintained, these shared components are repeatedly manipulated. As a result there are often significant overlaps and synergies across and among the different test efforts of different component-based systems. However, in practice, testers of different systems rarely collaborate, taking a test-all-by-yourself approach. As a result, redundant effort is spent testing common components, and important information that could be used to improve testing quality is lost.

The goal of this research is to demonstrate that, if done properly, testers of shared software components can save effort by avoiding redundant work, and can improve the test effectiveness for each component as well as for each component-based software system by using information obtained when testing across multiple components. To achieve this goal I have developed collaborative testing techniques and tools for developers and testers of component-based systems with shared components, applied the techniques to subject systems, and evaluated the cost and

effectiveness of applying the techniques.

The dissertation research is organized in three parts. First, I investigated current testing practices for component-based software systems to find the testing overlap and synergy we conjectured exists. Second, I designed and implemented infrastructure and related tools to facilitate communication and data sharing between testers. Third, I designed two testing processes to implement different collaborative testing algorithms and applied them to large actively developed software systems.

This dissertation has shown the benefits of collaborative testing across component developers who share their components. With collaborative testing, researchers can design algorithms and tools to support collaboration processes, achieve better efficiency in testing configurations, and discover inter-component compatibility faults within a minimal time window after they are introduced.

COLLABORATIVE TESTING ACROSS SHARED SOFTWARE COMPONENTS

by

Teng Long

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2015

Advisory Committee:
Professor Alan Sussman, Chair/Advisor
Professor Adam Porter, Co-Advisor
Professor Atif Memon, Co-Advisor
Professor Ilchul Yoon
Professor Donald Yeung, Dean's Representative

Table of Contents

List of Tables iv

List of Figures v

1 Introduction 1
1.1 Thesis Statement and Contributions 9

2 Related Work 11
2.1 Distributed Software Development 11
2.2 Regression Testing 12
2.3 Continuous Integration and Testing 13
2.4 Software Product Lines Testing 15

3 Background 16
3.1 Component-based Software Systems 16
3.2 Existing Testing and Collaboration methods 18
3.3 Annotated Component Dependency Model 20
3.4 Rachet Automatic Testing Framework 21
3.5 Testing Regression Configurations 23

4 Exploring Overlaps and Synergies 25
4.1 Modeling How Components are Exercised 25
4.2 Research Questions 27
4.3 Metrics 28
4.4 Subject Components 30
4.5 Study Procedure 33
4.6 Data and Analysis 34
4.6.1 Build Testing 34
4.6.2 Line/Branch Coverage 36
4.6.3 Parameter Value Coverage 40
4.6.4 Fault Detection 47
4.7 Summary 51

5 Collaborative Testing Infrastructure 52
5.1 Environment Model . . . 53
5.2 Conch Data Sharing Repository . . . 55
5.3 Sharing Virtual Machines with Environment Differencing . . . 58
5.4 Ad-hoc Collaborative Testing Process . . . 61
5.4.1 Testing Procedures for Component-based Systems . . . 61
5.4.2 Collaborative Build and Functional Testing . . . 63
5.5 Evaluation . . . 66
5.5.1 Collaborative Build and Functional Testing . . . 67
5.5.2 Continuous Collaborative Regression Testing . . . 71
5.6 Summary . . . 75

6 Coordinated Collaborative Testing Process 77
6.1 Coordinated Collaborative Testing Process . . . 78
6.1.1 Notification Scheme for Coordinated Collaborative Testing . . . 78
6.1.2 Strategy for Coordinated Collaboration . . . 80
6.1.3 Regression Testing based on Cross-Component Coverage . . . 83
6.2 Experiments . . . 84
6.2.1 Subject Components . . . 84
6.2.2 Testing Strategies . . . 85
6.2.3 Experimental Setup . . . 89
6.2.4 Experimental Results . . . 91
6.2.4.1 Comparing Cumulative Test Execution Time . . . 92
6.2.4.2 Comparing Maximum Fault Exposure Time . . . 93
6.2.4.3 Analyzing Cross-Component Compatibility Faults . . . 95
6.3 Summary . . . 97

7 Conclusions and Future Work 98
7.1 Thesis and Contributions . . . 98
7.2 Future Work . . . 101

Bibliography 104

List of Tables

4.1 Configurations tested by component developers 35
4.2 Induced Coverage of APR and MPICH2(%) 37
4.3 Induced Coverage Distribution of APR and MPICH2's users (%) . . . 37
4.4 Number of Numeric Parameter Values 40
4.5 Summary of Fault Detection Results 49

5.1 Subject Components 69
5.2 Configuration Preparation Cost(hours) and Benefits(%) 72
5.3 Regression Test Selection Results 75

6.1 Subject Components 85
6.2 Cumulative Time in Testing Strategies (in hours) 93

List of Figures

1.1 Systems with Common Components 4

3.1 An Example System Model 21
3.2 Example Regression Configurations of `A` 24

4.1 Induced Coverage Example 27
4.2 APR CDG 31
4.3 MPICH2 CDG 32
4.4 Induced Coverage of APR 38
4.5 Induced Coverage of MPICH2 39
4.6 APR Parameter Value Distribution 43
4.7 MPICH2 Parameter Value Distribution 45
4.8 Test results from all components on 112 faults seeded into MPICH2. . . 49

5.1 Simplified example environment description for a VM 55
5.2 The Conch Data Sharing Repository 56
5.3 Request for *SQLite* dependency data 57
5.4 *Conch* response with *SQLite* dependency data 57
5.5 High-level Design of Ede Infrastructure 59
5.6 Subject Systems on Ubuntu for collaborative Testing 66
5.7 Subject Systems on Debian for Collaborative Testing 68

6.1 Subject Components for Continuous Collaborative Testing 86
6.2 Subject Components Update History 87
6.3 Maximum Fault Exposure Time 94

Chapter 1: Introduction

Over the years, the practice of software development has changed. Instead of developing software from scratch, many developer groups and organizations rely on third-party software components, knitting them together to implement their system. Each component in a component-based system may have multiple versions, thus there can be a large number of version combinations (*configurations*) for a single software system. Also, as these components evolve independently during the life cycle of the systems, new versions are continuously released, and new end-user machine configurations that contain the new versions are added for the potential user base. If developers use Agile or DevOps methodologies [1, 2], which is prevalent in the software development community, the version (or build) release cycle can be very short, and the number of configurations can increase rapidly. As a result, component developer groups are challenged with testing a large amount of configurations in a timely manner.

However, today's component developers still continue to follow the old school "test-by-yourself" approach. This approach can be time-consuming and overwhelming for a single tester, because the total number of configurations grow exponentially with the numbers of components in a system [3], and isolated developers need to test

these configurations all by themselves. As a trade-off, in practice, developers have performed *compatibility testing* [4] by selecting a set of configurations and testing whether each configuration builds and functions correctly. This method leaves large number of configurations untested, and important compatibility faults might be left uncovered.

In this research, we posit that the paradigm shift to component-based software development has created numerous opportunities for sharing test effort by collaboration. Collaborative testing can not only boost test efficiency comparing to testing in isolation, but also provide opportunities to improve the quality of individual components. Our supposition is based on two characteristics of component-based systems that we discuss via the example shown in Figure 1.1 (we will discuss the nomenclature of the figure later in Chapter 3).

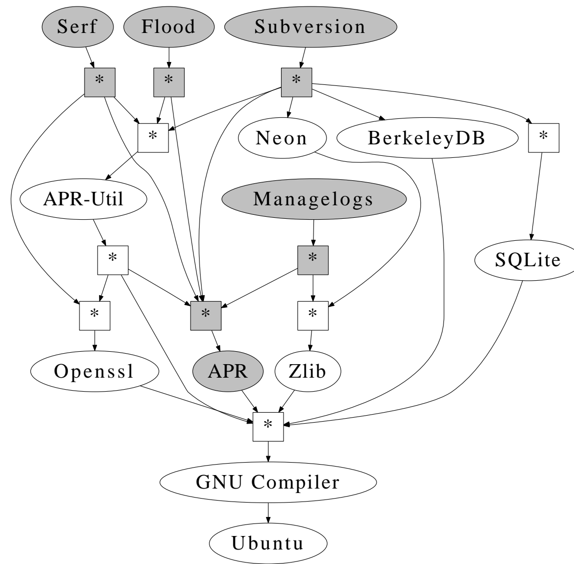
The first characteristic is that components in component-based software systems have **dependency** relationships between them, i.e., some components *use* or depend on other components. Consider the top-level right-most shaded node in Figure 1.1 labeled *Subversion*, which relies on other "lower-level" components or *provider* components, in this case *APR-util*, *SQLite*, *APR*, *Neon*, and *BerkeleyDB*, also shown as nodes connected directly to *subversion* via "$*$" connector boxes. **Opportunity 1: Exploit such provider-user relationships to share test effort and improve local tests of individual components.** More specifically, testers of the *user* components can inform the *provider* components about the context in which they are being used. Similarly, the *provider* components can inform their *user* components of the latest code changes and the latest test efforts and results.

This bi-directional flow of information can help to avoid overlaps in testing and also enables testers to focus their efforts where it can do the most good.

The second characteristic is that many components are commonly used or **shared** by multiple software systems. Consider, for example, that the *Apache Portable Runtime* library (*APR*) in Figure 1.1 is used not only by *Subversion*, but also by other systems, such as *Serf* and *Flood*. Building and testing any of these systems necessarily involves building *APR*, and therefore exercises *APR* as well. **Opportunity 2: Distribute test effort and share results for common components to lower cost and improve test quality.** More specifically, when two or more component-based systems use at least one common component, developers of the systems can collaborate in the testing of the common component, for instance, when pooling their test cases would help to achieve some desired coverage criteria. Alternatively, in the case where two or more low-level components implement the same interface and functionality, and could therefore be used interchangeably by the same high-level component [5], tests run on one low-level component could be extracted and applied to the other low-level components.

Based on these opportunities observed, I support the following thesis in this dissertation: **automated collaborative testing between developer groups of shared software component can i) improve the quality of compatibility testing of component-based systems; and ii) boost the efficiency of testing software system configurations.** The goal of this research is to develop automated collaborative testing theories and tools for individual developers of shared software components, so that their testing practice can be more efficient and with

Figure 1.1: Systems with Common Components



higher quality comparing to the paradigm of testing their components all by themselves.

This research involves three major parts: first, investigate the current procedures of testing component-based software systems to find opportunities of removing redundant work between testers as well as information that can be shared across testers to improve the test quality of each other. Second, design and build an infrastructure and related tools to facilitate communication and data sharing between testers, so that collaboration between them can be possible. Third, develop different collaborative testing processes upon the infrastructure, so that testers can rely on information shared by others to coordinate their own testing and improve the efficiency and effectiveness of their local tests.

As the initial step, we conjectured that overlap and synergy exist in testing functionally related components [6]. This conjecture is based on the fact that a component usually relies on one or multiple *provider component(s)* to support some of

the component's functionality, thus when exercising the tests of a component, parts of its provider component(s) are also being tested. We conducted an empirical study on two sets of components that functionally depend on two common components (*base components*) whose developers are currently testing them independently. Two research questions were addressed in this study: i)**to what extent do component developers duplicate their test effort when they are sharing provider components;** and ii) **and to what extent does testing by component users go beyond testing by the providers?** We exercised the test suites of some example *provider components* and their user component sets who functionally depend on them. Experiments suggested that: **first**, redundant test efforts are usually made when different testers are testing a component and/or its provider/user components; **second**, test cases designed and run by component users can exhibit new behaviors that are not covered by the original provider component's test cases, which may provide synergistic data to help improving the original provider component's testing. Through this step, we conclude that **it is worthwhile to build an infrastructure to support sharing of test results and artifacts between testers in order to eliminate the overlapped test effort, and to design techniques utilizing shared information to improve the quality of local tests of individual components.**

Next, we started building a prototype infrastructure for individual testers to exchange their test metadata, results and artifacts. The core component of this infrastructure is the **Conch** [7] data repository provided to testers as a set of web services. Automated testing tools can submit data to the repository as well as query

the repository to coordinate their own testing processes, or to speed up their testing by reusing results and prebuilt artifacts stored in the repository. To support scalable caching and sharing of rebuilt artifacts(virtual machine images), we also developed a tool called *Ede* [8] (Environment Differencing Engine). With *Ede*, *Conch* shares not full virtual machine images, but only the incremental environment differences from pristine operating systems. **The infrastructure provided by *Conch* and *Ede* connects isolated component developers, enables the developers to share testing results and artifacts automatically, and allows sophisticated collaborative testing strategies and processes to be implemented upon it.**

Relying on *Conch* and *Ede*, I further developed two collaborative testing processes that coordinate the local testing procedures of component developers.

The first process is called *ad-hoc collaborative testing*, which requires minimal modification to the current practice of isolated component developers conducting their testing. In this process, automated tools of isolated developers will query *Conch* before building any configuration, or running any functional test. If there are any prebuilt configurations or results shared for the same testing task, the developers can just reuse them and avoid redundant effort. Otherwise, they can continue with their original procedure, and share their prebuilt artifacts and/or test results afterward.

To evaluate the *ad-hoc collaborative testing process*, I developed a simulator that simulates individual testers of two sets of component assemblies using our collaborative testing process to coordinate their regression test. The simulation was based on the historical development data of the components during 2 years. Our

results showed that: **first**, by reusing shared functional test results and prebuilt system environments, individual testers can save enormous amount of time on preparing their test configurations when a provider component gets updated; **second**, by analyzing code coverage information shared in the repository, testers can save time by selecting only the affected regression tests instead of the whole regression test suite to exercise; and **third**, by analyzing the cases when updating a provider component causes regression test failure of a user component, testers can reveal faults in either provider or user components. As a conclusion, **the prototype data sharing infrastructure and the *ad-hoc* collaborative testing process have shown their merit supporting collaborative regression test of component assemblies with overlapping components by both eliminating redundancy of test effort and improving test quality of individual tests.**

The *ad-hoc* collaborative testing process can benefit component developers the most if reuse is maximized in this process. However, in a regression process where components are continuously updated and new configurations are introduced constantly, multiple developers may start to conduct regression testing if their components are affected by the update. This testing strategy increases redundancy in test effort spent by the groups, especially if there are inter-component dependencies or if the component is shared by multiple groups.

Thus, I developed a *coordinated collaborative regression testing process* for multiple developer groups, with the objectives of reducing the overall test redundancy across the groups as well as minimizing the time in which compatibility faults are exposed to the user community. The process involves a test scheduling and notification

mechanism via *Conch* across developer groups, so that each group is made aware of the configurations under test by other groups, enabling the groups to avoid performing redundant tests that could not be avoided by *ad-hoc* collaborative testing. We apply this process to a set of software systems with shared components in an Ubuntu distribution, emulate the application of this process over the 2-year history of the component development, and evaluate the cost and effectiveness of the process. Our experimental results show that **comparing to *ad-hoc* collaborative testing, *coordinated* collaborative testing can further reduce test redundancy across developers of shared software components, and still maintain the ability to discover cross-component compatibility faults within a minimum time window.**

Through the two example collaborative testing processes implemented on the data sharing infrastructure, this dissertation has shown the benefits of collaborative testing across component developers who share their components. With the idea of collaborative testing, researchers can design more sophisticated algorithms and systems to support other collaboration processes, achieve better efficiency in testing configurations, and discover more inter-component compatibility faults within a minimal time window after they are introduced.

In this dissertation research, certain assumptions are made in our implementation of collaborative testing processes, so that our work can focus on major challenges of enabling collaboration. For example, we are not considering malicious tester or untrusted testers who may share incorrect data. When testers share their functional testing results, we only focus on the tests whose results are deterministic.

Even though these assumptions may not be always true in industrial practice, they do not diminish the potential benefits that testers can obtain through collaborative testing.

1.1 Thesis Statement and Contributions

My thesis statement is: **Testers of shared software components can save test effort by avoiding redundant work, and improve the test effectiveness for each component as well as for each component-based software system by collaborating over their testing processes and by using information obtained when testing across multiple components.**

The contributions of my dissertation research include:

1. Our empirical study shows that overlaps and synergies exist in the testing processes of software components who share provider components. The overlaps can be eliminated to save test effort, and the synergies between testing of user components can be used to improve test quality of their shared provider components.

2. Using the ad-hoc collaborative testing process we implemented on our data sharing infrastructure, testers of components with shared provider components can significantly reduce their cost by avoiding both redundant testing tasks and unnecessary regression testing.

3. In regression testing, by leveraging our coordinated collaborative testing process, testers can not only save more effort than using the ad-hoc collaborative

testing process, but also minimize the window of finding compatibility faults introduced by provider component updates.

4. With collaborative testing processes, testers can discover compatibility faults that are not discoverable by testing components in isolation.

Chapter 2: Related Work

2.1 Distributed Software Development

Distributed software development has become common for software development, and many researchers have started investigating and evaluating such development processes. Ebert et al. studied the advantages and challenges of globally distribute development activities in [9]. Ramesh et al. [10] also discovered in their study that agile software development methods like extreme programming and distributed development can be blended to reap the benefits of both. In these studies, proper group collaboration and development coordination are considered a key factor to success. Although the two studies are both focusing on collaboration within a single organization, they do face some similar challenges of supporting continuous integration and efficient collaboration between participating groups.

To support distributed software development, researchers have emphasized the importance of tools for collaboration between distributed teams [11, 12]. Bird et al. [12] reported that globally distributed software development within a single company may not perform worse (in terms of failures) than centralized development. In [11], Begel et al. developed tools based on news-feeds to support developer teams collaborating with each other, because the teams should be aware of what other

teams are doing for managing risk in their development. However, these tools are designed to support human developers for better collaboration. They are not ready for automatic testing systems to adapt for supporting collaborative testing.

2.2 Regression Testing

Regression testing is designed to ensure that updates to software, such as adding new functionality or modifying existing features, do not falsely affect the functionality that should have been continuously supported. Usually a set of test cases is developed along with the software evolution to test modifications in previous versions, and regression testing is performed by running some or all of these test cases. There have been many research of techniques on designing regression test cases [13,14], regression tests prioritization [15,16], and regression tests selection [17, 18].

However, in the paradigm of component-based software systems, components are usually developed and maintained by different groups, each of which develops regression tests for their own component only. As a result, even though individual components may be well tested, the system consisting them may suffer compatibility faults across components. Many research target to address this problem by creating better cross-component regression tests [19, 20], or performing better continuous integration testing [21].

The research in this dissertation differs from the previous efforts, because we are relying on the regression tests created for individual components to improve the

overall compatibility of component-based systems. This research is based on two observations. First, creating cross-component regression tests may not be feasible in some scenarios. For example, in many open source communities, developers of a user component (usually the user-facing application, like SVN, etc.) may not have control over its provider components' design or development, nor do they have enough knowledge to these provider components. It will be challenging for the user component developers to create and maintain very comprehensive cross-component regression tests. Second, we observed that regression tests of user components are also testing the provider components by accessing their functionality. It is potentially of great merit if we utilize the information generated by such cross-component activities, use them to characterize the behavior of the whole system, and rely on the data to find potential compatibility faults.

The purpose of our research is not to replace integration testing or developing of cross-component regression tests, but to serve as a complementary means to achieve better compatibility testing.

2.3 Continuous Integration and Testing

There is some work on methods and tools to support continuous integration when multiple teams collaborate on large software projects [22, 23]. Elbaum et al. [22] designed algorithms to pre-select and prioritize test cases from test suites to make continuous integration processes more cost-efficient. Nilsson et al. [23] developed a visualization technique for visualizing end-to-end testing activities involved

in the continuous integration processes within projects or companies, so that such activities can be better arranged to support more efficient integration testing. However, Elbaum et al.'s method does not apply to the scenario of removing redundant effort between distributed component developers who collaborate with each other; and the tool from Nilsson et al. provides more complementary support for decision makings. The tool itself does not support automatic testing scheduling.

An important part of our work is the tools and infrastructure we provided to support coordinated collaborative testing, as part of the continuous integration. There are some distributed continuous quality assurance (QA) environments, such as Dart [24] and CruiseControl [25], to conduct continuous integration testing, which involves executing build and testing processes whenever check-ins to a repository occur. Users install agents that automatically check out software from a repository, build the software, execute functional tests, and submit the results to the server. However, the underlying QA process is hard-wired in Dart and CruiseControl and therefore other QA processes or implementations of the build and test process are not easily supported. Recently, continuous integration tools like Jenkins [26] and Autopkgtest [27] are becoming prevalent. Jenkins is a platform that supports continuous integration and delivery of software products. If properly configured, it can monitor the code repository changes of components and trigger testing activity. It is a good candidate platform that can be combined with the coordination scheduling. Autopkgtest is a tool supported by the Ubuntu community to facilitate compatibility testing in distributed environments. It enables developers to provide a set of functional test cases together with the released package. Other developers can easily

install the packages, and execute the provided test cases for compatibility testing. However, this process is neither automatic nor coordinated.

2.4 Software Product Lines Testing

A software product line(SPL) is a family of programs that are differentiated by their increments in functionality [28]. Since each product is derived from the core assets [29] based on the features to be exhibited by this product, compatibility testing must be applied to these products in order to validate the correctness of features implemented. To some extent, this process is similar to testing component-based systems. Researchers have proposed many approaches to test SPLs. Souto et al. [28] used a profile of passing and failing test runs to quickly identify failures that are indicative of real compatibility problems in test or code rather than specious failures due to illegal feature combinations. Lamancha et al. [30] worked on model-driven testing, which were used for one-off development, to an SPL setting. However, testing SPLs is fundamentally different from testing software components developed in isolation. SPLs are commonly derived from a single system for the purpose of reusability and productivity, thus they are commonly designed and maintained within a single group or organization, and a uniform model for the whole system is usually well-defined. Tests of products in an SPL can usually be derived from tests of core assets. The software components addressed in this dissertation, on the other side, are developed, maintained and tested in isolation, and there is no well-established compatibility tests generating methods for component-based systems.

Chapter 3: Background

This chapter describes the background of my dissertation research topic as well as other related research. I start by introducing the basics of component-based software systems, discussing different types of such systems and comparing their differences, then I describe the features of component-based systems that are targeted in this research. After that, I list a set of existing testing techniques for such systems, and also introduce some collaboration methods applied to such testing. Next I talk about a formal model for our target component-based software systems. Last, I introduce the example automatic testing process that we refer to as the baseline in which individual component testers test their products locally.

3.1 Component-based Software Systems

Component-based software engineering has been widely discussed for more than a decade, yet there is not yet a formal definition of *component* that is agreed on by everyone [31]. Different researchers have their own description of the features that a component should have. For example, He et al. developed a model called *rCOS* to define important concepts of component-based software development including interfaces, contracts, interaction protocols, components, etc., and used this

model to provide an integrated approach to facilitate component-based software development and verification [31]. Brereton et al. described a component as “an independently deliverable set of reusable services” [32] to explain the role of a component in a component-based system. Based on the goals of different research, the term “component” can be interpreted differently to emphasize different attributes of such component-based systems.

Within the scope of this research, we are studying component-based software systems from the perspective of collaborative software testing, especially functional testing. Thus we are interested in how testers of different components interact with each other during their development processes and perform functional testing of their components. From this perspective, components in this study possess the following features.

- **Independence**: Each component is developed and maintained independently by groups who have their own self-managed development and testing processes.
- **Dependency**: A component either depends on or is depended by other component(s). Here component A *depends on* component B means that B must be functioning correctly for component A to be successfully built and correctly functioning in the same environment.
- **Consistency**: A component provides a set of consistent interfaces to its users, and such interfaces do not change dramatically when the component is updated to newer versions.
- **Activeness**: A component is actively being updated and tested by its devel-

opers and testers, so that collaboration on development and testing between this component and other components is possible.

In this study, a software component can be an application that depends on a set of other components which further depend on other components. All components integrated together form a component assembly and provide the functionality offered by the top-level component. We define such a component assembly as a *component-based software system*. The formal model of component-based software systems that we adopt is going to be introduced in Chapter 3.3.

3.2 Existing Testing and Collaboration methods

As described in Chapter 3.1, each component has its own independent development and testing processes. It is very common that developers of a component manage their own test suites and exercise unit tests and regression tests regularly. When a component depends on multiple other components, it is usually tested against limited numbers of configurations that are most popular among users [33]. All testing tasks for a component are exercised locally by the testers of the same component. This is the most common practice of testing components, which is considered the baseline case of testing a software component.

In the baseline scenario, opportunities for collaborations are limited, and forms of collaborations are restricted. Since components are tested separately by their developers, the most common way of collaboration is that component testers report bugs to developers of components that they depend on, and the developers fix those

bugs in later versions. Many project hosting services such as Launchpad [34] and SourceForge [35] provide sophisticated bug reporting and tracking services. However, such a collaboration mechanism heavily depends on the willingness of testers to manually share their bug reports.

Since component-based software engineering has already attracted the attention of many researchers, many of them are also proposing methods to test component-based systems in different ways other than separated individual testing. Leeuwen et al. proposed a framework that can be used to evaluate properties of component-based software systems include liveness, progress of subsystems, robustness and fairness [36]. Wu et al. proposed Component Interaction Graph(CIG) to model component-based systems, then investigated different types of faults in component-based systems as well as some elements in testing such systems [37]. They also proposed a family of criteria to evaluate such testing. However, neither work proposed any practical technique that supports collaboration between testers of related components.

For this research, we model component-based software systems using *Annotated Component Dependency Model(ACDM)* [33], and create collaborative testing techniques based on the *Rachet* testing procedure for component-based software systems. Both *ACDM* and *Rachet* will be introduced in more details in later chapters. From the software testing perspective, *Rachet* provides a framework that systematically tests compatibility of systems composed from components with different versions on different platforms, which we believe is more comprehensive than other known testing methods of component-based systems. However, a *Rachet* testing

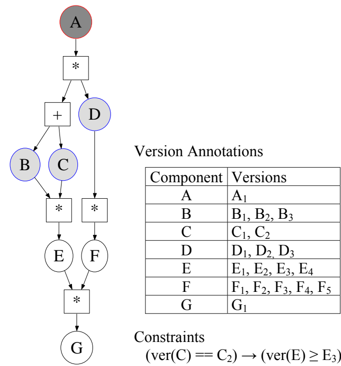
procedure is still hosted by a single tester to test the compatibility of components related to a specific top-level component. When multiple testers are running their own *Rachet* procedures, there still exists no collaboration between them. In this research, we are going to extend similar scenarios and provide a collaborative mechanism between testers of components running similar testing procedures.

3.3 Annotated Component Dependency Model

In this dissertation, we model component-based software systems using the *Annotated Component Dependency Model(ACDM)* [33]. In this model, a component-based system can be depicted with two parts: a directed acyclic graph called the *Component Dependency Graph(CDG)* and a set of *Annotations*. As shown in Figure 3.1, each node in the CDG represents a unique component, and inter-component dependencies are specified by connecting nodes with **AND**(*) or **XOR**(+) relationships. For example, in Figure 3.1 component A "depends on" component D and either one of B or C. Here dependency means that one component requires another component at build-time, runtime, or both. *Annotations* in this example include version identifiers for components, and constraints between different components and component versions, written in first-order logic.

When different systems share components, the relationships between these systems can be represented by an integrated CDG with overlapping regions. In the example CDG in Figure 1.1, the top-level components (*Serf, Flood, Subversion* and *Managelogs*) depend on different provider components. There are overlaps between

Figure 3.1: An Example System Model



Component	Versions
A	A_1
B	B_1, B_2, B_3
C	C_1, C_2
D	D_1, D_2, D_3
E	E_1, E_2, E_3, E_4
F	F_1, F_2, F_3, F_4, F_5
G	G_1

the set of required provider components, and the APR component is required by all top-level user components. This suggests that each top-level component developer will use his/her test resources to build the components contained in the shared subgraph, starting from the APR node to the bottom node, and then test the behavior of those components to ensure a functioning build of the top-level component. In this scenario, those developers are likely to perform redundant test efforts that could be eliminated or advantageously redirected if all of these components were able to share their test data and artifacts.

3.4 Rachet Automatic Testing Framework

One concern that component developers have is to make sure that their components build correctly. This activity has typically been performed by manually checking component builds on a handful of popular user configurations. However, this is time-consuming, error-prone and limited in scope given the large number of combinations of platforms, components, and versions in which components might be

built. In prior work in my research group, we designed a process and infrastructure called *Rachet* [33] to address this challenge.

Rachet tackles this problem in several ways. First, it reduces the number of configurations that must be tested, by applying a sampling strategy called *DD-coverage*. With this coverage criterion, all *direct dependencies* between components are covered at least once by sampled configurations. Second, *Rachet* generates a schedule to test sampled configurations, and then performs build testing in parallel using multiple nodes in a cluster or cloud computing environment. Each configuration is tested in a virtual machine (VM) environment hosted on a physical node. *Rachet* further reduces test effort by reusing virtual machine environments that instantiate partially-constructed configurations. Because building components is time-consuming and because multiple configurations often share common partial configurations, *Rachet* builds systems inside virtual machines and then reuses the virtual machines across different physical cluster nodes.

Even though *Rachet* utilizes distributed resources to conduct build testing, its test plans and associated test tasks are still managed and assigned in a centralized way locally by the tester. In other words, this infrastructure was designed to be used to test a single software system. In addition, the virtual machine instances that *Rachet* currently employs are quite large, which will be problematic in a collaborative test situation. In order to share build test results and cached virtual machine artifacts among multiple testers, an external collaborative framework is needed, a set of APIs must be provided to *Rachet* to interact with that framework, and *Rachet*'s virtual machine artifacts must be compact for efficient sharing. In the

research of this dissertation, we further improve the efficiency of testing component-based systems by reusing test results shared by collaborators, and we developed a technique called *environment differencing* that can significantly reduce the size of virtual machine artifacts.

3.5 Testing Regression Configurations

As components in a software system evolve, newer versions of the components become available. If a new version of a provider component in a CDG is released, it introduces new configurations that have to be tested by affected user component developers. For example, in Figure 3.1, assume that only one version of each component is available initially; A_1 for the component A, B_1 for the component B, and so on. If the developer of the component D releases a new version D_2, $\{B_1, D_2, E_1, F_1, G_1\}$ and $\{C_1, D_2, E_1, F_1, G_1\}$ are new configurations on which the developers of the component A need to test. We call these configurations **regression configurations**.

Given a **CDG** and an ordered list of all component updates **U**, the total number of regression configurations for each component developer can be computed. Figure 3.2 shows an example update history of the components in Figure 3.1 and the regression configurations introduced for A's developers after each update. For the given update history, a total of 21 regression configurations are introduced.

After a component update, developers of every user component must ensure that the user component can be built without any errors in the new regression configurations, and also the test results obtained by running the user component

Figure 3.2: Example Regression Configurations of `A`

Original Versions: A_1, B_1, C_1, D_1, E_1, F_1, G_1

Update Order: B_2, C_2, D_2, E_2, F_2, F_3, B_3, D_3, E_3, F_4, E_4, F_5

Regression Configurations:

A_1, B_2, D_1, E_1, F_1, G_1 → B_2
A_1, C_2, D_1, E_1, F_1, G_1 → C_2
A_1, B_2, D_2, E_1, F_1, G_1 → D_2
A_1, C_2, D_2, E_1, F_1, G_1 → D_2
A_1, B_2, D_2, E_2, F_1, G_1 → E_2
A_1, C_2, D_2, E_2, F_1, G_1 → E_2
⋮

test cases relevant to the update should remain identical to the ones obtained before the update. If there are any build or test failures, then compatibility faults have been introduced between the user component and the updated component. This activity has to be repeatedly executed over all regression configurations, and in this dissertation it is referred to as *regression testing*.

Chapter 4: Exploring Overlaps and Synergies

In this chapter, I describe my initial study of searching for overlaps and synergies in the testing processes of functionally related components. First, I talk about the model used to study how components get exercised by their *user* components in a component assembly. Then I introduce the research questions at this stage of our study. Next, I propose several metrics to quantify the informal definition of *overlaps* and *synergies* in the context of this study. Forwarding that, target software components and the study procedure are presented. At last, I analyze the data obtained from this empirical study to investigate the overlaps and synergies of testing shared components.

4.1 Modeling How Components are Exercised

First, we model how components are exercised by other components in a component assembly.

Induced Coverage: Suppose component a directly or indirectly uses component b, and a has a test suite T_a. In a system where a is successfully built on b, when running a's test suite, T_a, the fraction of b's coverage elements (lines, branches, functions, parameter values, faults, etc.) that get covered is called ***b*'s induced**

coverage from *a*, represented as C_b^a.

To demonstrate the concept of induced coverage, we take the sub-CDG from Figure 3.1 that contains components A, B, C and E as an example, and focus on line coverage. Suppose each component has a test suite, correspondingly named T_A, T_B, T_C, and T_E, and that there are 10 lines in E's source code. When running the four test suites, different lines of **E** get covered. Suppose lines 1, 2, 4, 5 get covered by T_A, lines 3, 4, 5, 6, 8 get covered by T_B, lines 5, 6, 9, 10 get covered by T_C, and lines 3, 4, 5, 7, 10 get covered by T_E. The induced line coverage from these components to component E is shown as in Figure 4.1. Each column represents a line in E's source code, and each row shows the corresponding coverage. A filled block means the line is covered, and a blank one means that it is not.

Union of Induced Coverage: When both components a and b use c, the union of their induced coverage for c ($C_c^a \cup C_c^b$) is defined as the fraction of c's elements that is covered by either a or b.

Intersection of Induced Coverage: When both components a and b use c, the intersection of their induced coverage to c ($C_c^a \cap C_c^b$) is defined as the fraction of c's elements that is in both a and b's induced coverage to c.

Difference of Induced Coverage: When both components a and b use c, the difference of a and b's induced coverage to c ($C_c^a - C_c^b$) is defined as the fraction of c's elements that is in a but not b's induced coverage to c.

$C_E^B \cup C_E^C$, $C_E^B \cap C_E^C$, and $C_E^B - C_E^C$ are also demonstrated in Figure 4.1.

Figure 4.1: Induced Coverage Example

1 2 3 4 5 6 7 8 9 10

A
B
C
E
$C_E^B \cup C_E^C$
$C_E^B \cap C_E^C$
$C_E^B - C_E^C$

4.2 Research Questions

Our vision of collaborative testing is based on the conjecture that there is actionable structure in the efforts of testing functionally related components. We believe that there are significant overlaps in the way components shared by multiple users are tested. If we are correct, then such duplicate work could be avoided by sharing test results with no loss of testing effectiveness. We also believe that different component users test shared components in unique ways, so the aggregate testing of the entire component assembly is often broader than that done by individual component providers.

In this initial empirical study, we attempt to formalize and quantify some of these issues in the context of two real software component assemblies in which some user components share a number of provider components. We selected these specific components because each component has its own build and functional tests. Our analyses involve executing the test cases and studying how various execution metrics overlap across components, and also show that some individual testers' efforts are

not duplicated so they can provide added test value.

Research Questions: More specifically, we are interested in answering the following research questions:

RQ1: Overlap: To what extent do testers of shared components duplicate test effort?

RQ2: Synergies: To what extent does testing by component users go beyond that done by the providers?

RQ3: Usage Distance [1] Effects: Do overlap and synergy measures change as usage distance grows?

4.3 Metrics

To answer the research questions, we first develop concrete metrics to quantify the informal concepts of "overlap" and "synergy". We treat build and functional testing separately due to the disjoint nature of their test artifacts – the former uses build scripts whereas the latter uses functional tests.

Metrics: For *build testing*, for each component C_j, we define $\psi(C_i, C_j)$ as the set of configurations of C_j build tested by the developer/tester of component C_i. Note that $\psi(A, A)$ is valid – it represents the set of configurations on which component A is build tested by the developer of A. Having defined ψ to return a set, we use set intersection, $\cap$, to study overlaps in build testing. We use set union, $\cup$, to study

[1] In a CDG, distance between two components is defined as the number of components on the shortest path between these two components.

the synergies in build testing by multiple testers.

For *functional testing*, we use code (line and branch) coverage, parameter value coverage, and faults detected, to measure overlaps and synergies among functional test cases. We use a matrix representation for code coverage and faults detected. For parameter value coverage, we record all values observed for each numeric parameter. More formally, given a test suite $TS(C_i)$ for a component C_i that invokes a set of functions $\mathcal{F}$ of component C_j, we record the following artifacts:

- A *code coverage matrix* that, for each test case in $TS(C_i)$, records the number of times a coverage element (line and branch) in C_j was covered.

- *Parameter values*, a list of values, one element for each numeric parameter of a function $f \in \mathcal{F}$.

- A *fault matrix* that records whether each test case in $TS(C_i)$ passed or failed, and the fault detected.

Given these artifacts, we compute several metrics: (1) induced code coverage for line and branch from testing a provider component and its users, (2) ranges of parameter values passed to functions of a provider component when running the test cases of its user components, and (3) number of faults detected in a provider component by running the test cases of its user components.

4.4 Subject Components

We study two widely-used open source software components: APR and MPICH2. APR[2] (Apache Portable Runtime) is widely used in the web services community, for instance, by components such as the Apache HTTP server and the Subversion version control system. MPICH2[3], from the high performance computing (HPC) community, is an implementation of the Message Passing Interface (MPI) standard that is used to implement scientific applications on many high performance and parallel computing platforms.

We further identify several components that use APR and MPICH2. In this study, the user components of APR include *flood*, a profile-driven HTTP load tester that collects important performance metrics for websites, *managelogs*, a log processing program used with Apache's piped logfile feature, *serf*, a C-based HTTP client library that provides high performance network operation with minimum resource usage, and *subversion*, a widely used version control system. The user components of MPICH2 include *FreePooma*, a C++ library that supports element-wise data-parallel and stencil-based physics computations using single or multiple processors, *PETSc*, a suite of data structures and routines for the scalable (parallel) solution of partial differential equations, and *ParMETIS*, a parallel library that implements many algorithms for partitioning unstructured graphs and meshes. *SLEPc*, a library for solving large-scale sparse eigenvalue problems on parallel computers, and *TAO*,

[2] http://apr.apache.org

[3] http://www.mcs.anl.gov/research/projects/mpich2

a library for large-scale optimization problems, are user components of the PETSc component, and therefore they indirectly use MPICH2. Figure 4.2 and 4.3 show the CDGs for APR and MPICH2, respectively. We highlight the components that we focus on in this study in the CDGs.

Figure 4.2: APR CDG

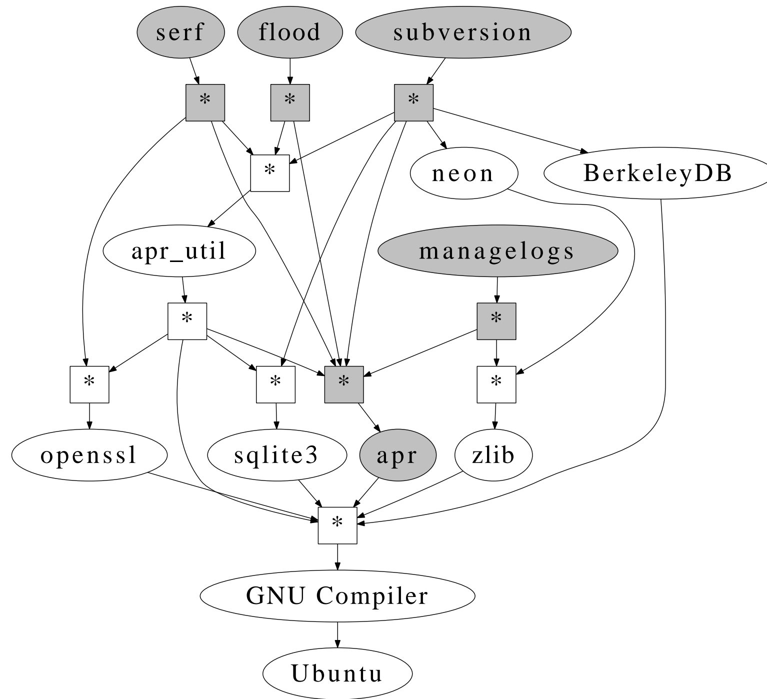
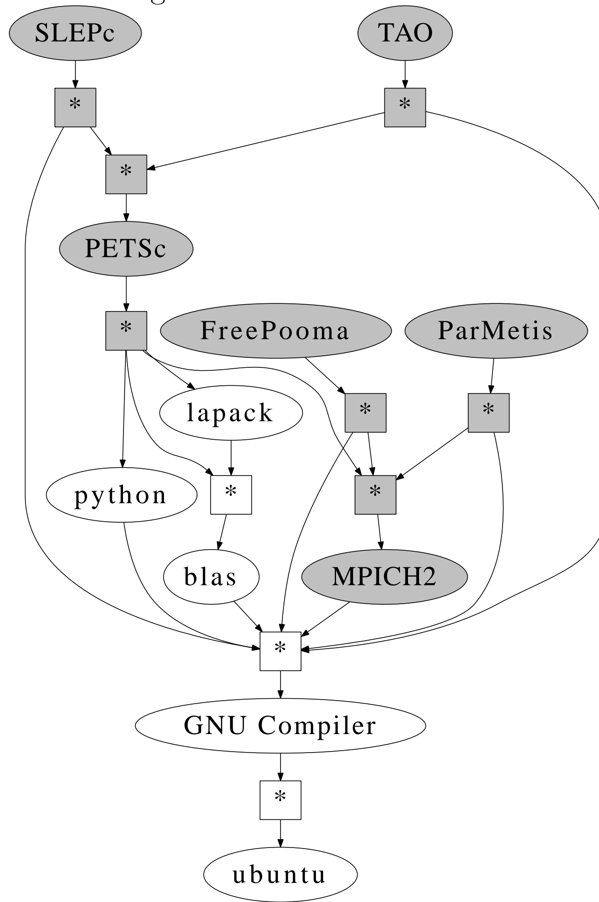


Figure 4.3: MPICH2 CDG



4.5 Study Procedure

APR, MPICH2 and their users provide their own test cases and for this study we execute the test cases for each component, measuring how the test cases cover APR and MPICH2 using each of the metrics described earlier.

For MPICH2 we further break down the test data by usage distance – i.e., higher-level components that directly use MPICH2 are differentiated from higher-level components that indirectly use MPICH2 because there is an intermediate component in the CDG between the top level component and MPICH2. We did this specifically for fault detection, to see if and how testing behaviors change as the distance between the user components and MPICH2 increases.

All measurements are conducted on virtual machines with 1GB RAM and a single core CPU simulated by Oracle VirtualBox 4.1.6. Components are built using the GNU compilers version 4.4.3 (which includes gcc, g++ and gfortran), and coverage information is collected by lcov 1.9. Code coverage information is collected for two operating systems: Ubuntu 10.04.3 32bit and FreeBSD 8.2 32bit. Since we have observed very similar code coverage on both systems, we conducted experiments to collect parameter value coverage and fault detection only on the Ubuntu platform.

4.6 Data and Analysis

We analyzed data obtained from component development documentation and test artifacts from our empirical study to understand the overlaps and synergies of shared test effort in loosely-coupled communities. In Section 4.6.1 we first identify configurations on which subject components were build-tested by component providers and users, then we discuss the possibility of broadening the set of tested configurations and saving test effort by sharing build test results. From Section 4.6.2 to Section 4.6.4 we analyze the code coverage, parameter value coverage and fault coverage information collected by running functional tests of subject components and also user components of the subject components.

4.6.1 Build Testing

For each component C_j in the CDGs in Figures 4.2 and 4.3, we first investigated $\psi(C_i, C_j)$ – i.e., we examined configurations *build-tested* by component providers and also configurations tested by component users. This was accomplished by inspecting documents provided by component providers (e.g., HTML documents, Wiki pages, user manuals, installation guides, and/or nightly-build test results). In some cases, component providers do not clearly specify configurations on which their components build successfully. For example, component providers can simply list prerequisite components and expected configurations on which their components may be built successfully. When we do not have sufficient information to determine working configurations, we examined files such as *Makefile* to find relevant information.

Table 4.1 lists configurations on which the components can be built successfully.

Table 4.1: Configurations tested by component developers

Component	OS (tested by providers)	Prerequisite components	Remarks
APR	UNIX variants, Windows, Netware, MAC OS X, OS/2	C compiler	
flood	Linux, Solaris	APR, C compiler	known to work on FreeBSD
managelogs	Linux	APR, C compiler	tested with apr 0.9, 1.3
serf	UNIX variants	APR, C compiler	
Subversion	Linux, FreeBSD, Windows, OpenBSD, MAC OS X, Solaris	APR, C compiler, SQLite, libz	use *buildbot* for build test
MPICH2	Linux, Cygwin, AIX (on IBM Blue Gene/P), MAC OS X, Solaris	C, C++, Fortran compilers	nightly build test for GNU, Intel, PGI, Absoft compilers
PETSc	AIX, Linux, Cygwin, FreeBSD, Solaris, MAC OS X	C, C++, Fortran compilers MPI library	nightly build test for platforms
FreePooma	AIX, Linux, Solaris	MPI library, C++ compiler	
ParMETIS	Linux	MPI library, C compiler	
TAO	Cygwin, MAC OS X, Linux, FreeBSD, AIX, Solaris, UNICOS(on Cray T3E)	PETSc, C++ compiler	
SLEPc	Linux	C, C++, Fortran, PETSc	

From Table 4.1, we observe that several components are regularly build-tested on sets of predetermined configurations, and the tests are performed using automatic build tools such as *buildbot* or custom scripts on dedicated machines (e.g., *subversion*, MPICH2 and PETSc). However, tested configurations mostly consisted of recent versions of operating systems and other components. One reason may be that developers focused their limited resources on testing their components with recent versions of required components, under the belief that users' configurations had been updated to recent versions of required components. However, this is not necessarily true.

We also observe that successful component builds are often tested on a wider set of configurations by component users than by component providers. For example, *subversion* is build-tested on top of virtual machines hosting different operating systems, as listed in Table 4.1. Since *subversion* requires APR, developers have to

first build APR successfully for the configurations, and some of those configurations are not explicitly tested by the APR developers. Build test results from the *subversion* developers can be used to increase the set of configurations on which building APR is known to be successful.

Even though the number of configurations tested by component users is small, that information can be valuable if the configurations are not commonly tested by other testers. For example, PETSc is tested on the AIX operating system, which is not tested by the nightly build system for MPICH2. Although the PETSc developers do not test PETSc for all configurations where MPICH2 can be built, their test results can be useful to inform other users of the configurations where MPICH2 is known to build successfully. In addition, the versions of components used in the configurations to test PETSc are not always the same as the ones used by the component developers. For example, the GNU C compiler version used by the MPICH2 developers can be different from the version used by the PETSc developers. Test results from the PETSc developers can therefore provide useful information to the MPICH2 developers, because successful component build can depend on the versions of the required components.

4.6.2 Line/Branch Coverage

This analysis of functional testing examined how line and branch coverage changed depending on which component's tests were being run. Other coverage metrics, such as method coverage, dataflow coverage, etc., could also be used to

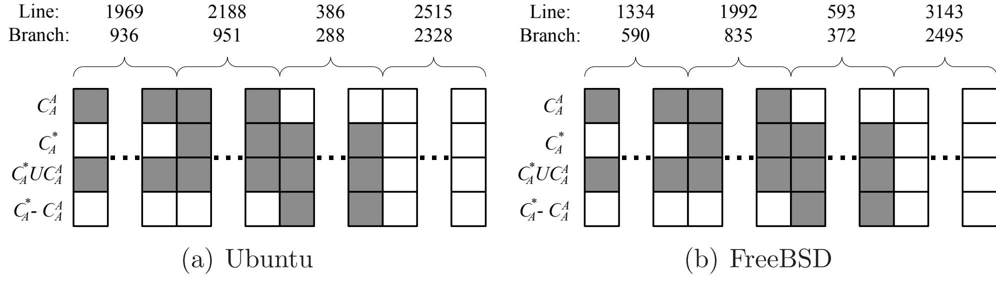
Table 4.2: Induced Coverage of APR and MPICH2(%)

APR	Indirect Test		Self Test		Union		Extra	
	b	l	b	l	b	l	b	l
Ubuntu	27.5	36.5	41.9	58.9	48.3	64.4	6.4	5.5
FreeBSD	28.1	36.6	33.2	47.1	41.9	55.5	8.7	8.4
MPICH2	**Indirect Test**		**Self Test**		**Union**		**Extra**	
	b	l	b	l	b	l	b	l
Ubuntu	10.6	15.3	39.1	47.8	39.3.	48.0	0.2	0.2
FreeBSD	10.4	15.2	39.2	47.2	39.5	47.5	0.3	0.3

Table 4.3: Induced Coverage Distribution of APR and MPICH2's users (%)

APR	One		Two		Three		Four	
	b	l	b	l	b	l	b	l
Ubuntu	16.05	18.23	6.75	9.05	1.51	3.50	3.20	5.68
FreeBSD	16.36	18.42	7.04	9.06	1.54	3.48	3.19	5.64
MPICH2	**One**		**Two**		**Three**			
	b	l	b	l	b	l		
Ubuntu	2.70	3.15	6.11	8.61	1.80	3.54		
FreeBSD	2.82	3.12	5.75	8.55	1.82	3.53		

Figure 4.4: Induced Coverage of APR



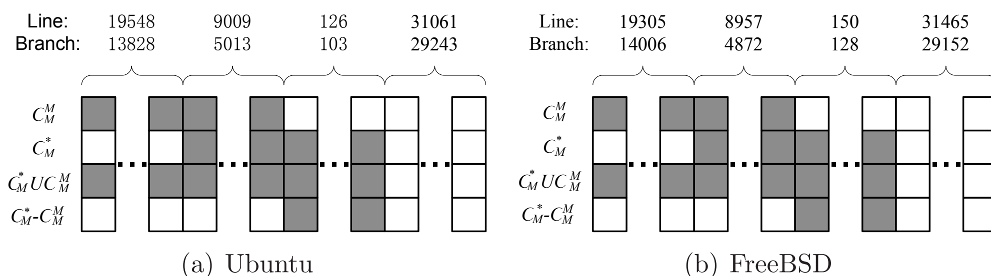
(a) Ubuntu (b) FreeBSD

measure the effectiveness of user components' test suites on testing the provider components.

Table 4.2 shows the induced line and branch coverage of APR and MPICH2 as measured by the *lcov* tool, on two OS platforms. The "Indirect Test" columns show the union of induced coverage from *all* the test suites of their direct users, while the "Self Test" columns show coverage from just APR's or MPICH2's own test suites. "Union" show the union of induced coverage from all components, while "Extra" shows the difference between induced coverage of APR and MPICH2 from their direct users and their own tests. Figures 4.4 and 4.5 show the actual number of lines/branches in the induced coverage of both components on two operating systems. C_A^A and C_M^M are the induced coverage from APR and MPICH2's own tests, while C_A^* and C_M^* denote the union of coverage induced from all direct users of the provider component in the subject systems.

The results of this analysis show that the user components together achieved substantial coverage, but not as much coverage as the test suites of the provider components did. However, the user components provided at least some additional coverage not achieved by the provider components' tests. The extra coverage for

Figure 4.5: Induced Coverage of MPICH2



(a) Ubuntu (b) FreeBSD

MPICH2 is small, because MPICH2 is an implementation of an industry standard with a well-documented and widely-used API, and as a result has a very thorough test suite.

Our second analysis examined how the cumulative coverage achieved by testing multiple user components broke down by the component that was doing the testing, i.e., by which component's tests were being run. Table 4.3 shows the fraction of APR's and MPICH2's code that are covered by only one, two, three or four of the user component(s), respectively, for branch and line coverage on two different platforms.

The results of this analysis suggest that for the APR example, while there was some overlap in coverage from different users of APR, the tests of different users tended to cover APR's functionality in different ways. More specifically, among all the lines or branches covered by the test suites of the users of APR, about half of them were covered by only one user component. For MPICH2, since it is an implementation of the MPI standard, there are a set of functions that are used by almost all MPI programs, such as MPI_init and $MPI_finalize$. Thus we observed more overlap of coverage among user components of MPICH2, compared to the

Table 4.4: Number of Numeric Parameter Values

APR	flood	svn	serf	managelogs		Union
a	13	38	11	11		62
b	62	85	59	35		123
MPICH	PETSc	FreePooma	ParMETIS	SLEPc	TAO	Union
a	13	1	10	3	18	26
b	247	55	269	140	246	302

Parameter values from user component tests are sometimes outside the range of values tested by the test suites for provider components. a: number of value range-extended parameters; b:Total number of numeric parameters

overlap between user components of APR. However, there was still around 20% of induced coverage from the users that were covered by only one user's tests, which again shows that different users have different ways of using the provider component. Thus the more users a component has, the better induced coverage it will get from its users' test cases.

Since we got very similar results on both the freeBSD and Ubuntu platforms, we only considered Ubuntu for the subsequent studies.

4.6.3 Parameter Value Coverage

To analyze values of individual parameters passed to functions in the provider components (i.e., APR and MPICH2), we instrumented all functions of APR and MPICH2 if they have at least one numeric parameter. We then ran the test suites of APR and MPICH2, and also the test suites of their users (See Figures 4.2 and 4.3). We collected the information about the parameter values passed into the instrumented functions, to see how the patterns of such values differed across the various test suites.

We observed that the test cases for user components often invoked functions in APR and MPICH2 with values outside the range of values covered by the provider components' test suites. In Table 4.4, for each provider component, we show the total number of numeric parameters in the functions invoked while running test cases for the user components, and also show the number of parameters for which values tested by the user components were outside the range of values covered by the test suites of the provider components. The rightmost column (Union) in the figure shows the total number of parameters tested by at least one user component.

For the APR component, 180 numeric parameters were covered by running both the test suites of APR and its user components. Among the parameters, 123 were covered by running only the test suites of user components, and parameter value ranges were extended for 62 parameters. That is, for the range-extended parameters, there was at least one value that is greater than the maximum value (or, smaller than the minimum value) covered by APR's own test cases. It is noteworthy that 14 parameters were not covered by any test case of APR but were tested by the test cases of one or more user components. For the MPICH2 component, 302 out of 762 numeric parameters were covered by the user components and the value ranges were extended for 26 parameters. For MPICH2 there was no parameter that is covered by testing user components but not covered by the tests of the provider component. Again, that is because MPICH2 contains many test cases to check compliance of the implementation to the MPI standard.

These results imply that boundary values for some parameters were not considered when provider component developers created their test cases, or that user

component developers used incorrect or unexpected parameter values. We do not have information on the relationship between the correctness of the functions and specific parameter values used in user components' test cases, and also we do not assume that extended value ranges from user components are better in quality than the value range of provider components. However, our results suggest that provider component developers can learn more about the actual uses of their components if they are provided with parameter value information from user components. Such information could be used to reduce developer efforts to create test cases, if developers and users share the information about parameter value coverage.

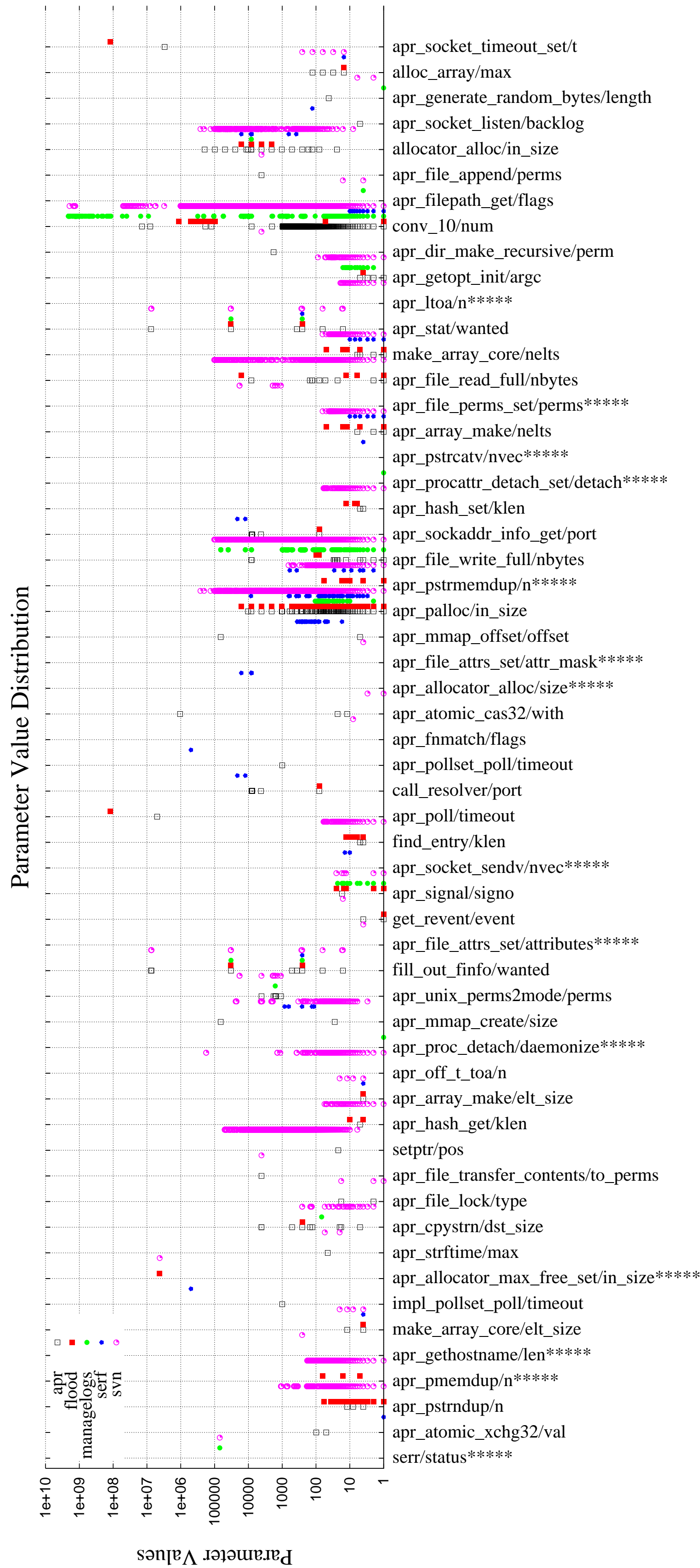


Figure 4.6: APR Parameter Value Distribution

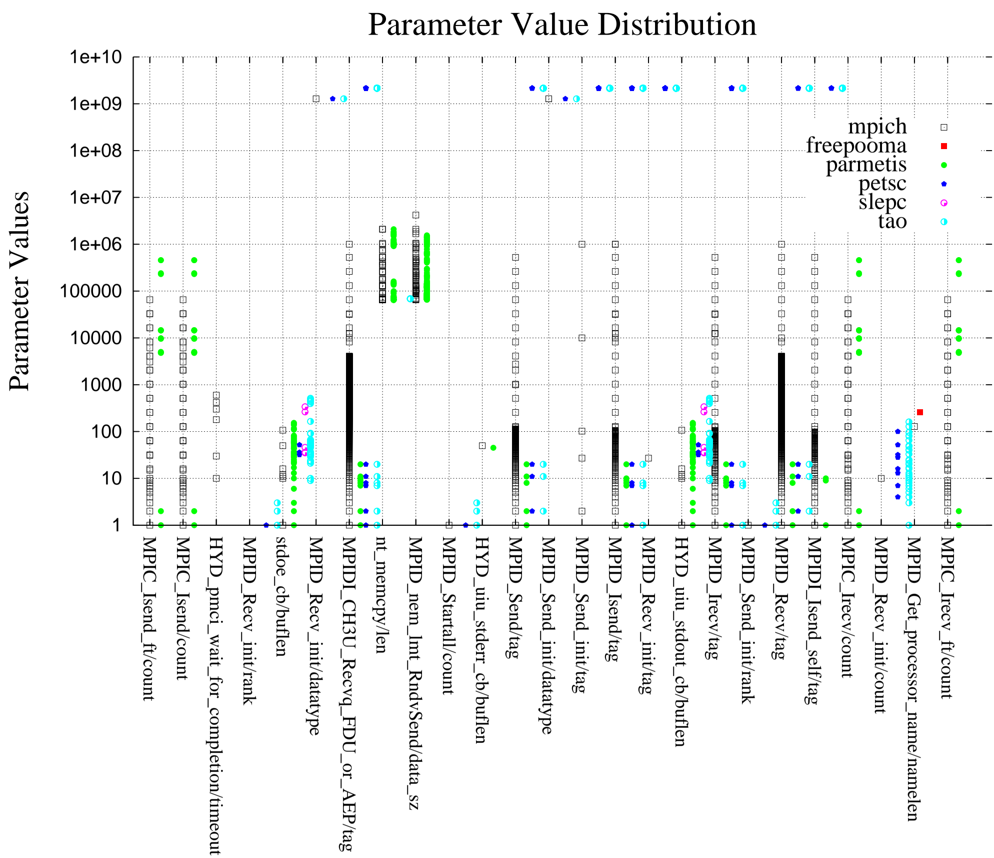
Table 4.4 shows the number of value range-extended parameters, but that information by itself does not say that a broader set of test values is used by user components than by the provider component test suite. For example, a user component may test a provider component function with only one value outside the value range covered by the test suite of the provider component. To look in more details at the actual parameter values covered by the provider component test suites and by the user components' test suites, we show the distribution of parameter values covered by individual subject components in Figures 4.6 and 4.7.[4]

In the figures, the x-axis is pairs of function-name/parameter-name in the provider components, while the y-axis is the values passed into the function from running the test suites of the providers and user components. For presentation purposes, we only showed parameters for which value coverage was extended by user components. The y-axis is log-scale because of the wide range of parameter values, so we also do not show parameters values less than or equal to 0. In each graph, the parameter values covered by the provider component are aligned to a vertical grid line, and values covered by user components are depicted on the right side of the line in the order shown in the graph legend.

In the figures, we observe that user components often test functions in the provider components with a larger set of parameter values. For the APR component, *svn* tested many APR functions with diverse values, compared to the values tested by APR's test suites. For example, the APR test suite used only -1 and 5 as

[4]We omitted 5 APR parameters that represents the time data type, and 1 MPICH2 parameter that is the 'argc' parameter of a program that uses MPICH2.

Figure 4.7: MPICH2 Parameter Value Distribution



the values of the *klen* parameter for the function *apr_hash_get*, but the *svn* test suite used 35 different values between -1 and 56, including 0. For the MPICH2 component, we see that the MPICH2 test suites test itself uses many parameter values. This is not surprising, because, as previously noted, MPICH2 developers must do rigorous testing to ensure compliance with the MPI standard. While it is true that the number of different parameter values tested does not always increase overall test quality, sharing test results from component users with component developers can help the developers identify faults, especially if specific parameter values are associated with the faults.

We also observe that there are APR functions tested only by user components. In Figure 4.6, such function-parameter pairs are indicated by appending "*****" to the *x*-axis labels. For example, the function *apr_pmemdup* in the APR component is not tested by the APR test suite, although the function is invoked with many values by the test suites of *flood*, *serf*, and *svn*.

Furthermore, we found different user components invoke different functions of a provider component. For example, *managelogs* heavily invoked the APR function *apr_file_write_full* but the function is not invoked from *serf*. That is, we expect that parameter value distribution can provide useful information about the actual usage patterns of the provider component by other components. If component users are willing to share parameter value distribution information with component providers, both sides will be rewarded, because providers can use the information to improve the quality of their components and users will end up using more thoroughly tested components.

4.6.4 Fault Detection

By now we have observed significant induced coverage to the provider components from testing their users, including both code coverage and parameter value range coverage. The real question we want to ask is: will such coverage help detect faults within the components? To answer this question, we seeded faults in one provider component, and observed whether such faults were detected when running the test suites of both the component that contained the seeded faults and the component(s) that directly or indirectly used it.

Given that a significant number of faults must be seeded and tested individually to ensure the validity of our study, and that each round of testing for all components may be very time consuming (about an hour for each fault in our study), our subject system for this analysis was limited to a sub-CDG from Figure 4.3. The sub-CDG included MPICH2 as the provider component, PETSc as a component that directly uses MPICH2, and TAO and SLEPc as components that indirectly use MPICH2 through PETSc. Two categories of faults were seeded to simulate real-world faults, which were:

1. **operator faults**: a change of an operator in the source code, including both arithmetic operators ('+', '-', '*' and '/') and comparison operators ('>', '<', '!=', '>=', '<=' and '==').
2. **constant faults**: a change of a constant value defined in macros in the source code. Non-zero constants are changed to zero, and vice versa.

In order to choose the locations to seed faults in an unbiased way, first we found all lines in the source code of MPICH2 that were covered by at least one of the four subject components above. Then for each such opportunity, we randomly generated a probability value between 0 and 1. When the probability exceeded a threshold, we chose that line to seed a fault. Since there were far more opportunities for operator faults than constant faults, we used different thresholds for the two fault categories. In our study, there were 6516 opportunities to seed operator faults, and 16 opportunities to seed constant faults. To generate a reasonable number of faults that covers both categories effectively, the probability threshold was set as 0.985 for operator faults, and 0.0 for constant faults. We included all opportunities for constant faults given that their total number was very small. As a result, 96 opportunities for operator faults and all 16 opportunities for constant faults were chosen.

The testing results are shown in Figure 4.8. Each four boxes in a column represents the test results for the four components built on/from the source code of MPICH2 which has a single fault seeded. A filled box means the fault is detected by testing the corresponding component, while a blank box means the fault is not detected. Table 4.6.4 presents a summary of the results. Several observations can be made from the results:

1. All faults detected by users were also detected by the test suite of the provider component, i.e., no extra faults were detected by users.

2. Among all 112 faults seeded, 87 of them were detected by MPICH2's own test

Figure 4.8: Test results from all components on 112 faults seeded into MPICH2.

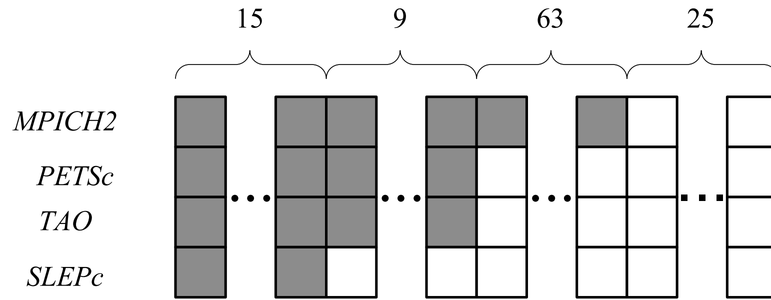


Table 4.5: Summary of Fault Detection Results

Fault Types	Faults Seeded	Provider	PETSc	TAO & SLEPc	Union Union	Extra Extra
Operator	96	75	21	21	75	0
Constant	16	12	3	3	3	0
All	112	87	24	24	87	0

suite, 24 of them were detected by testing its direct user (PETSc), and 24 of them were detected by testing its indirect users (TAO & SLEPc).

3. All faults detected by direct users were also detected by indirect users.

The first observation is not encouraging, but is quite reasonable for this case study. From Table 4.2 we can see that only 0.2% of MPICH2's code is covered exclusively by its users, while its own tests cover 47.8% lines. Thus the probability that a fault is seeded in the code which is exclusively covered by the users is less than 0.004. When choosing 112 faults uniformly among the code covered by either component, it is quite likely that no faults are seeded in such portion of MPICH2. We manually examined the code after the study and found that there were only 48 lines that were covered exclusively by MPICH2's users, and none of the automatically-seeded faults resided in those lines. This fact is consistent with our result.

The second observation shows testing MPICH2's users alone can detect about 20% of all seeded faults. Recall that all faults were seeded in the code covered by at least one component. Considering the fact that testing users alone covers about 15.3% lines of MPICH2's source code, while the union of all components' induced line coverage is 48.0% (refer to Table 4.2), The probability a fault is seeded in the code covered by users is about 30%. Thus the result of fault detection is roughly consistent with code coverage.

The third observation implies that faults are unlikely to be hidden by distance in the CDG. In other words, faults in a provider component are still discoverable by testing users that are far in the CDG from the component.

The second and third observation reveal the coherence between induced coverage and propagated faults from the provider component to its users. Since we already see considerable induced coverage in both subject communities, it is reasonable to claim that component users will help to find faults in the provider component by running their own test suites.

Finally, we are only testing one of MPICH2's direct users and two indirect users for fault detection. There are many more users of this component. We can expect more code to be covered exclusively by its users.

4.7 Summary

This empirical study is driven by our original research conjecture – that in the context of compatibility testing across shared software components, it may be profitable to optimize testing processes across multiple components, rather than within individual components as is generally done today. To explore this idea, we have conducted an initial study of two groups of user components that functionally depend on two common provider components. We reviewed their current testing processes, investigated how much their test efforts overlap and whether the total costs or quality could be improved if their testing activities are coordinated. The results suggest that the test cases designed and run by component users can be individually less comprehensive than those by component providers, but in some cases can exhibit new behaviors not covered by the original provider's test cases. In addition, we have found that testing done at different levels in a CDG by different components that use the same provider component appear to be complementary. Finally, these results suggest that test results from the higher level components might provide useful feedback for understanding usage patterns or operational profiles from a component user's perspective. Component developers could use this feedback to improve their own test suites. In conclusion, both overlaps and synergies do exist in the testing process of the subject systems. Thus it is worthwhile to build a collaborative testing infrastructure, which can help individual testers of avoid redundant test effort, and utilize the synergies to improve the test quality of their own components.

Chapter 5: Collaborative Testing Infrastructure

In the previous chapter, I described an empirical study aiming at exploring overlaps and synergies in the functional testing processes of components that share provider components. The result of our study over two subject component assemblies confirmed the existence of such overlaps and synergies. It is therefore worthwhile to build an infrastructure to support collaborative testing, which could eliminate overlaps and use the synergies to achieve better testing quality for all participants.

This chapter presents our work on building an example collaborative testing infrastructure. The core component is a web-service based data sharing repository called *Conch*, which allows testing tools for different component developers and testers to share their testing artifacts and results. To support scalable caching and sharing of testing artifacts in the format of virtual machine images , we also developed *Ede*, an environment differencing engine. Last, we built an *ad-hoc* collaborative testing process upon this infrastructure, and evaluated its effectiveness as well as performance over a set of example components.

5.1 Environment Model

Collaborative activities work when individual efforts can be leveraged in a common group activity or used as artifacts. For instance, configuration management systems allow individual developers to modify source code independently and then merge their changes into a common version. In order to leverage independent testing efforts of component-based software systems, it is necessary to control the test *environment* in which a component is built and tested so that test results will be comparable across different testers. Thus, we provide a notional definition of a test *environment* as follows:

Definition 1: An *environment* where a component built and tested in includes all **pre-built component** instances in a system, the **tools** to be used to build the new component, all **source code** needed by the build, and **all other controllable factors** known to determine the result of the component's build process and the correct functioning of the component.

Controlling the environment in this way maximizes the likelihood that two testers building and testing the same component can share and combine their test results. That is, any differences in results should be attributable only to differences in how the components were tested, not in where or by whom they were tested. To gain this control, we attempt to standardize the test *environment* used by each tester. We have identified several factors that may affect the build and functional testing of components, and therefore must be captured by the test *environment*. These factors include:

- Hardware parameters (processor type, memory system, etc.)
- Operating system (architecture, kernel version, system core libraries, etc.)
- Build environment (compiler, compiler options, extra instrumentation inserted, etc.)
- Provider components (versions, their build settings and installation options, etc.)

Of course, this approach is not bullet-proof. We cannot, for example, account for **unknowable or random** factors, such as transient hardware faults in one tester's computing device, which surely affect how a component behaves.

A *Virtual Machine*(VM) with an installed operating system and pre-built core components is an intuitive way to encapsulate an environment, and sharing of pre-built environments then becomes sharing of VM images. In order to describe the *environment* encapsulated in a VM image, we associate an XML description file with each shared VM image. The description contains information about the hardware parameters of the VM, operating system information, pre-built components and their build options, and other information that may affect the test results. When accessing the repository, test tools search for VM images instantiating specific *environments* based on the description files.

The information contained in a typical description file is shown in Figure 5.1. In this *environment* there are six components, including the operating system and a compiler. Two of them (*SQLite* and *APR*) are built from source code, and their

Figure 5.1: Simplified example environment description for a VM

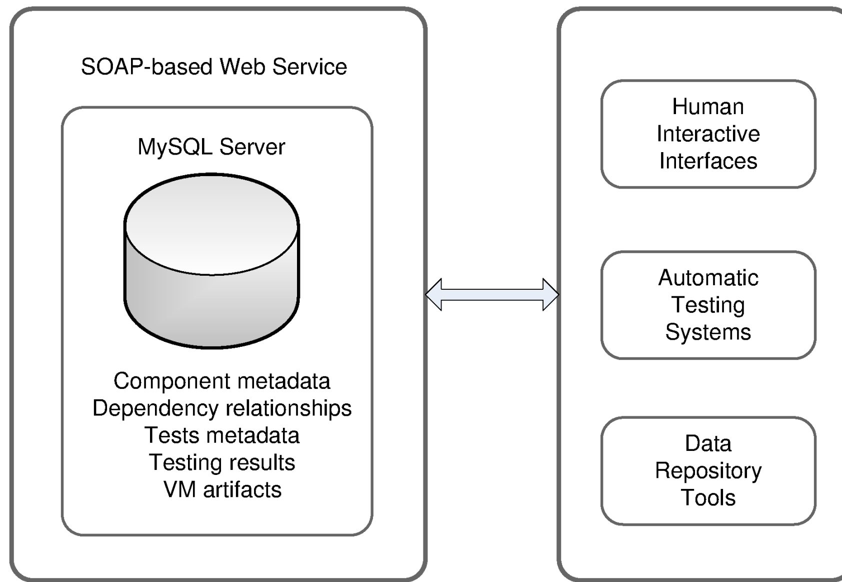
Component	
name	sqlite3-1.3.5
compiler-used	gcc-4.4.6
build-flag	with-apr="/usr/local/apr"
build-flag	CFLAGS="-O0"
Component	
name	apr-1.4.5
compiler-used	gcc-4.4.6
build-flag	CFLAGS="-O0"
Component	
name	bdb-6.0.20
status	system-prebuilt
Component	
name	neon-1.6
status	system-prebuilt
Compiler	
name	gcc-4.4.6
status	system-prebuilt
OS	
name	Ubunt-12.04
archtecture	X86_64
Hardware	
cpu_cores	2
cpu_frequency	2.8GHz
RAM	1024M

build flags are shown. The other components, except the operating system, are pre-built binary packages provided in the Ubuntu 12.04 software distribution. Three hardware parameters are also included in this *environment*.

5.2 Conch Data Sharing Repository

To facilitate data sharing among testers and their tools, we have designed and implemented a web-service based data repository called *Conch*. The structure of *Conch* is shown in Figure 5.2. The repository uses a MySQL database as the back-end, and provides a set of data query and management methods wrapped as web services. The web services are described using WSDL [38] and can be accessed via standard SOAP [39] protocols. Using the protocols, testers or other third-parties

Figure 5.2: The Conch Data Sharing Repository



can easily write tools and plug-ins that allow their automated test systems to access the repository, to analyze repository data, and to coordinate their testing processes with those of other testers.

Depending on the type of collaborations between automated test tools, the data types shared in the repository can be different, thus the data schema for the repository can be customized too. For the sharing scenarios we consider in this dissertation, the data stored in *Conch* has five major types: (1) component metadata, (2) component dependency relationships, (3) test case metadata, (4) test results, and (5) virtual machine artifacts (environments).

When an automated test tool submits test results to the repository, a unique test data record is created for each result. Each test data record is associated with the *environment* in which the test activity was performed, and with an outcome or test result, such as test success or failure. Other information regarding the tests (e.g., the raw output of running such tests) can also be stored in the repository

Figure 5.3: Request for *SQLite* dependency data

Conch Request	
command_name	getCDG
command_session	001
component	SQLite

Figure 5.4: *Conch* response with *SQLite* dependency data

Conch Response	
command_name	getCDG
command_session	001
component	SQLite
CDG	[+ gcc pgcc intelc] ncurses tclsh

for other test tools to interpret. Testers and their tools can retrieve existing test results by searching through the *environment* descriptions of existing test results. Users can submit or query test data by sending and receiving messages to/from the repository via Web service interfaces.

A response from the *Conch* server may contain links to access data, instead of actual data. For example, a response may contain a URL that points to a virtual machine image file. The information shown in Figure 5.3 and Figure 5.4 illustrates the content of example message exchanges for a user's request for dependency information for the *SQLite* component. The dependency data is returned back to the requester as a string in the server's response. The data request is initiated by a user-side automatic testing system that provides *Conch* with the information in Figure 5.3, and the response is in the form of an XML file that contains the information in Figure 5.4.

5.3 Sharing Virtual Machines with Environment Differencing

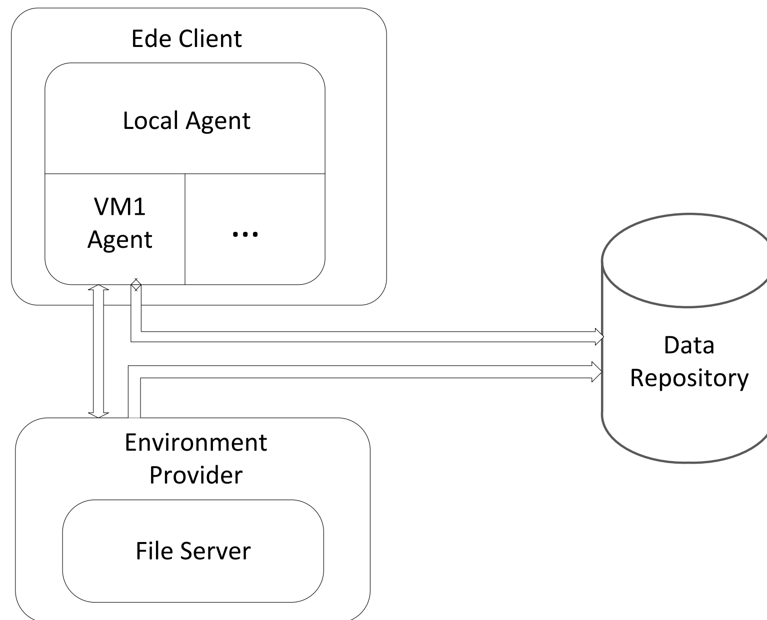
Before building and testing a component, an *environment* that contains all its provider components must be prepared. Such an *environment* can be encapsulated as a virtual machine (VM) image. However, unlike test results or component metadata, the size of a VM image can be very large[1]. Moreover, the sheer number of potential pre-built *environments* that could be shared among testers and their tools makes it difficult to store the VM images in the repository, and limited network bandwidth makes it challenging to transfer the environments over a wide-area network, if they are cached locally at individual testers' sites.

To overcome these challenges, we have developed a tool called *Ede* (**E**nvironment **D**ifferencing **E**ngine) [40] that supports automated *Environment Differencing*. Whenever a new *environment* containing a pristine operating system is prepared, *Ede* creates a signature file for the whole operating system, which includes the state of all existing files. After building and installing additional components in this environment, *Ede* inspects all files and records all changes as a *delta* file. A *delta* file records file deletions and creations, permission changes, etc., and can be automatically applied to another VM that has the **same** pristine operating system installed.

The high-level design of *Ede* is shown in Figure 5.5. *Ede* can have multiple clients and environment providers, and the system in Figure 5.5 has one of each.

[1] The size of a virtual machine image that encapsulates just a Linux operating system can easily be greater than 1 GB, even with only a minimal installation.

Figure 5.5: High-level Design of Ede Infrastructure



Every *Ede* client has two parts: a *local agent* and *virtual machine(VM) instance(s)*. The local agent acquires pristine virtual machine instances, and controls the state of each VM instance managed by the agent by applying the *update* operations. The local agent accomplishes this by communicating with a process called *VM agent* running inside each VM instance. The process is invoked when the VM boots up and executes commands for various tasks required for managing the VM state.

When the VM agent receives from the local agent the information on the system environment that needs to be provisioned, it first queries the data repository to search for a prebuilt system environment. If found, the agent updates the VM state to the specified state and informs the local agent that the environment is ready. Otherwise, the local agent is responsible for provisioning the desired system environment locally, which means that required components should be built in the VM from a pristine state or from another locally stored system environment.

With help from *Ede*, test tools that target at systems with common components can share their pre-built *environments* by storing only *delta* files, along with *environment* descriptions, in the repository. Sharing pre-built *environments* will save time for provisioning a new environment, compared to building an environment from scratch. Consider the components illustrated in the CDG of Figure 3.1. Testing components D and F requires an environment where G is installed. However, a tool that tests component D can save test effort and focus on testing only component D if the tool can reuse a pre-built *environment* in which F is already built. In this case, the tool testing component F will first retrieve from the *Conch* repository a *delta* file for a VM that has G installed. The tool can then restore the full VM locally by invoking *Ede*, build and test F in the VM, then create yet another *delta* file that contains both G and F in the corresponding VM. This *delta* file and its description file are then stored into the repository for later sharing.

Environment Differencing requires individual test tools to locally store root virtual machine images, which encapsulate environments with a pristine operating system installed on a specific hardware platform. Whenever a tester needs a pre-built environment that is available in the *Conch* repository, the test tool can download the desired *delta* file and automatically apply it using *Ede*. Storing delta files and transferring them over a wide area network is not too expensive. The size of a typical *delta* file is small (often between 10MB and 100MB), and the patch process does not take long (usually less than one minute). This enables the repository to store many environments created during test sessions. This approach is more cost effective than our previous approach of transferring whole virtual machine images [33].

5.4 Ad-hoc Collaborative Testing Process

In this section we describe an ad-hoc collaborative test process for component-based software systems implemented upon the *Conch* data sharing repository and the *Ede* environment differencing engine. In this process, pre-built *environments* and functional test results are shared by different testers, as well as coverage information for provider components induced from testing their user components. Testers of different components collaborate by sharing test data stored in the *Conch* repository and do not need to directly communicate with each other in order to benefit from the collaboration.

5.4.1 Testing Procedures for Component-based Systems

Before introducing the collaborative testing process, we need to revisit the process for testing a single component-based software system. A component-based system can be considered as a top-level user component plus all the provider components that it depends on. Thus whenever a provider component is updated, part or the whole of this component-based system needs to be rebuilt and tested to validate whether the newer version of the modified component still works in the system correctly. Three steps should be followed for the system validation activity at such changes:

1. Build and run functional tests of the new version of the provider component in desired environments.

2. Build and run functional tests of all other provider components dependent upon the modified component directly or indirectly.

3. Build and run functional tests of the user component.

Consider, for instance, the *Subversion* system in Figure 5.6. If a new *APR* version is available for the *Subversion* system, *Subversion* testers will first need to build the *APR* version on system configurations they support, and then run the test suite of *APR* to make sure it functions correctly on the configurations. Afterward, all other components that directly and indirectly depend on the *APR* component need to be rebuilt and functionally tested with the new *APR* version. If everything works correctly, testers will build and test *Subversion* last to make sure it behaves correctly.

Since components are developed and maintained by separate groups, when *APR* is updated, testers of not only *Subversion* but also all other components in Figure 5.6 that use *APR* may be interested in the effects of the update. Thus part of the building and testing work conducted by testers of *Subversion* may also be repeated by testers of other components. In addition, as seen in Figure 1.1, *APR* is used by multiple other systems as well. It is very likely that testers of those components repeat the identical build and test activity that may have already been conducted by other testers. Hence the opportunity to reuse existing pre-built *environments* and functional test results generated by other testers does exist if component-based systems are tested collaboratively. In Chapter 5.4.2, we will discuss how to use *Conch* to share pre-built environments and functional test results

and save test time by avoiding redundant work.

A component typically accesses only a subset of code regions in its provider components when its test cases are executed. In the example of testing *Subversion* upon a newer version of *APR*, testers would run the whole test suite of *Subversion*. However by sharing code coverage data, a regression test tool for *Subversion* can keep the mapping between individual test cases and the code regions in *APR* covered by executing the test cases. Thus the regression test tools for *Subversion* and for all other user components of *APR* should be notified when the *APR* code is changed. Then, the tools can execute only the selected test cases relevant to the change by analyzing the coverage data, and this will contribute to reducing the test workload further.

If a regression test fails for a revision of a provider component when it used to pass with a previous revision of the provider component, it means either the newer version introduces a new fault that makes the test fail, or there are problems in the failed test itself. In the former case, testers may provide feedback to the developer of the provider component, so that the fault can get fixed in later revisions. In the latter case, the testers can fix the erroneous test. In either case, the developers and testers benefit from receiving regression test results promptly.

5.4.2 Collaborative Build and Functional Testing

In a component-based software system, build testing of a specific component can be considered as a part of its functional testing, because the component can

be functionally tested only if it can be successfully built in an *environment* (or *configuration*). In addition, all the components on which it depends (i.e., its provider components) must also be built and function correctly.

Assuming that an operating system deployed on a hardware platform provides hardware independence, one of the primary interests of component testers will be to test the correct build and behavior of their components on a large set of heterogeneous *environments*. Note that an *environment* on which a component is to be built and tested is an instantiated subgraph of a CDG – i.e., all its provider components are assigned a specific version already.

Given a component and an *environment*, a test tool can use Algorithm 1 to provision the *environment*. The algorithm is designed to reuse existing pre-built *environments* in *Conch* as much as possible to rapidly provision the environment before building and testing the component. There is no guarantee that a desired environment is already shared by others, nor is it mandatory for all testers to share their pre-built environments. We call this testing process *ad-hoc collaborative testing*.

In this algorithm, ***C*** is the subject component to be tested, ***Env*** is the desired *environment* in which ***C*** will be tested, and ***Repo*** is the data sharing repository that stores pre-built *environments* as VM artifacts. If the desired *environment* ***Env*** is already instantiated (by this tester or a different tester) and available in the repository, the test tool can simply retrieve the VM that encapsulates the environment, and build and test ***C*** (line 1–3). Otherwise, the tool retrieves all provider components and their versions contained in ***Env*** (line 5), finds a pre-built *environment*

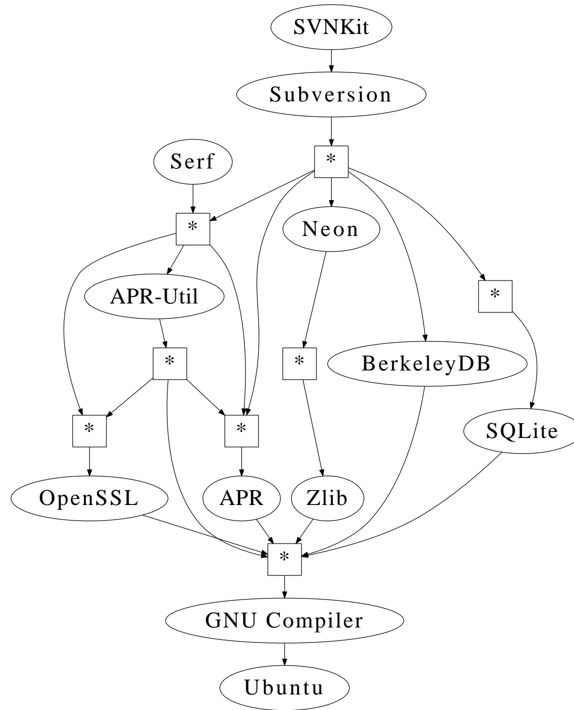
Algorithm 1: RapidTest(*C*, *Env*, *Repo*)

```
   Data:
   C: subject component
   Env: target environment
   Repo: repository that includes pre-built environments
1  if Env exists in Repo then
2  |  Retrieve Env from Repo;
3  |  Build and test component C in Env;
4  else
5  |  P ← getProviders(Env);
6  |  subEnv ← findBestMatch(Env, Repo);
7  |  P' ← getProviders(subEnv);
8  |  Build and test P − P' on subEnv;
9  |  Build and test component C on subEnv;
10 end
```

from ***Repo*** that requires the minimum extra build effort to create the desired *environment* (line 6). The tool can then build the extra components required by ***C*** (line 7–8), and finally build and test ***C*** (line 9).

The procedure *findBestMatch()* can be implemented using either historical records or heuristics to find a partial *environment* that a test tool can modify to meet its requirements. In the special case that no pre-built *environment* is found and ***subEnv*** is empty, the test tool will have to start from scratch – i.e., all components contained in the *environment* ***Env*** (except the operating system) must be built and tested.

Figure 5.6: Subject Systems on Ubuntu for collaborative Testing



5.5 Evaluation

In this section we evaluate the benefits of applying the ad-hoc collaborative testing process described in the previous sections to test components with overlapping regions in their CDGs, compared to testing the components in isolation.

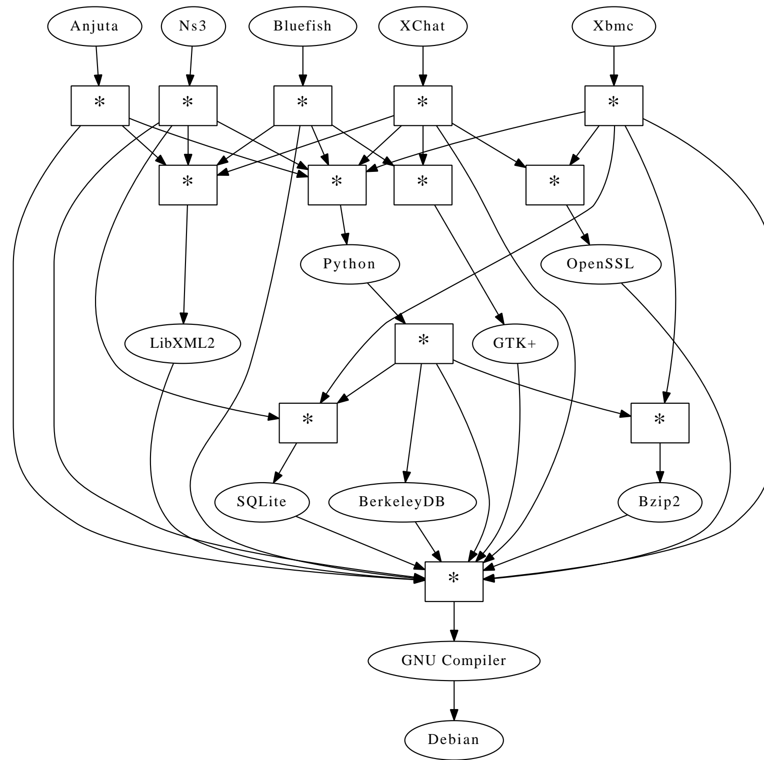
In Section 5.5.1, we evaluate the benefits of the collaborative testing process with two sets of top-level components that share provider components, as shown in Figures 5.6 and 5.7. While replaying the version release history of the components contained in the CDGs over a period of time, we conducted compatibility testing using *Rachet* [33] at each component version release, and measured the building and testing time that could be saved when different sharing strategies supported by *Conch* are applied.

In Chapter 5.5.2 we demonstrate the value of collaborative regression testing in the development process. We ran the regression tests of user components at new provider component version releases, and found bugs in both provider components and user components' test cases. That is, developers can discover problems caused by the changes in their provider components quickly after the problems are introduced, as well as can find previously undiscovered problems in users' tests. We also have developed a tool that uses regression test data stored in *Conch*, selects test cases that have to be rerun when a provider component changes, and then triggers the regression tests with the selected test cases. The tool uses *Jenkins* [26] as the automatic regression test client. We evaluate the collaborative testing process with the version release history of the components in Figure 5.6 over one year.

5.5.1 Collaborative Build and Functional Testing

In order to evaluate the benefits of collaborative testing, we first recorded the wall-clock time required for building and testing the components in the CDGs shown in Figures 5.6 and 5.7 on an environment (i.e., a VM image) sandboxed with VirtualBox. For each component, the recorded time includes only the time required for building and testing the component itself, assuming that all its provider components are already built in the environment. Only default test suites supplied with the component source code are executed and the running times are measured. In Figure 5.6, the top-level components are *SVNKit* and *Serf*. *SVNKit* is an Open Source Pure Java Subversion Library, and *Serf* is a high performance C-based HTTP

Figure 5.7: Subject Systems on Debian for Collaborative Testing



client library. In Figure 5.7, the top-level components are *Anjuta*, *Ns3*, *Bluefish*, *Xchat* and *XBMC*, all of which are user applications in the Debian Linux system. The CDGs also show the components on which the top-level components depend (i.e. their provider components). Brief descriptions of the components are given in Table 5.1.

For each component, we replayed all its version releases over one year (between 8/3/2012 and 8/3/2013). At each version release, we test the compatibility of the version with existing versions of its provider components, and also trigger compatibility testing of all its user components. For the existing provider component versions, we used the versions released between 8/3/2011 and 8/3/2012. The direct-dependency coverage (DD coverage [33]) criterion is used to compute configurations

Table 5.1: Subject Components

Component	Description
SVNKit	Open Source pure Java Subversion library
Subversion	version control system
Neon	HTTP and WebDAV client library
Zlib	compression library
BerkeleyDB	library for embedded database
APR	supporting library for Apache projects
APR-util	support library for *APR*
SQLite	SQL database engine
Openssl	open source toolkit for SSL/TLS
Gcc	GNU C compiler
Ubuntu	Ubuntu Operating System
Anjuta	GNOME Integrated Development Environment
Ns3	discrete-event network simulator
Bluefish	editor targeted towards programmers
XChat	multi-platform IRC chat program
XBMC	open Source Home Theater Software
Python	object-oriented programming language
LibXML2	XML C parser and toolkit of Gnome
GTK+	toolkit for creating GUI on multiple platforms
Bzip2	high-quality, open-source data compressor
Debian	operating system

newly introduced because of the version releases. The recorded times required for building and testing components are then used to simulate the total test time using the following three sharing strategies. We used the time cost of successfully building each component and executing all its tests for the simulation, so that the simulated time cost reflects the worst scenario.

Strategy 1: No sharing. This is the baseline strategy, which is the most time-consuming, because testing any component in a CDG requires both building and functional testing of *all* its provider components (i.e., all the components in the CDG sub-graph rooted at the component being tested), before building and testing the target component. In this strategy, there is no test data sharing between testers

at all. **Strategy 2: Sharing test results only**. Test tools share functional test results for each component tested. Provider components still must be built, but their functional tests will not be run if the results are available in the *Conch* repository. That is, the tools execute functional tests of the provider components only when there has been no previous test session that contains the test results. **Strategy 3: Sharing test results and pre-built environments**. Test tools share not only functional test results, but also pre-built *environments*. In this strategy, a test tool can select a pre-built *environment* in the format of a Virtual Machine delta file from the repository, and only build and test the components missing from the retrieved environment.

For Strategies 2 and 3, when a new component version is released, we expect that different developer groups will start testing their components with the new version at different times. Then the group that starts its testing later will have more opportunities to reuse test results and artifacts produced during the test sessions performed by other groups. For a fair evaluation, we have the repository notify the different developer groups in random orders for Strategies 2 and 3, and we repeated each simulation 100 times and computed the average times. We assume a bandwidth of 4MB/s for transferring VM delta files over the Internet.

To better understand the amount of work that can be saved by sharing test information via the *Conch* repository, we added up the times required for building and testing newly introduced configurations at each version release of the provider components of the top-level components. We call the sum the **total configuration preparation cost**. Table 5.2 shows the total configuration preparation cost for

each top-level component shown in Figures 5.6 and 5.7.

In Table 5.2, the first column shows the names of the top-level components in both CDGs, the next three columns present the average configuration preparation cost (in hours) for each component in our simulation for the different strategies, and the last two columns show the configuration preparation cost saving in percent for Strategy 2 and 3, respectively, compared to Strategy 1. The table shows that sharing functional test results alone reduces the preparation cost by 10% to 15% for most components. We see huge time savings when testers start sharing test results and pre-built test environments. The total cost was reduced by 52.2% for testing *SVNKit* and *Serf*, and by 77.2% for testing the top-level Debian components. These results clearly show that testers can significantly reduce their testing workload by sharing their test results and pre-built environments with other testers though *Conch*.

5.5.2 Continuous Collaborative Regression Testing

In this section we replay the continuous development of three provider components, *APR*, *Openssl* and *SQLite*, contained in the combined CDG in Figure 5.6 using their version release history between 8/3/2012 and 8/3/2013. Our tool monitors the code repositories of the three components. Whenever there are source code changes in any of the components, the tool (1) identifies all user components whose regression tests could be affected, (2) automatically builds the affected user component(s) as well as all other required components relying on the new provider components, and (3) reruns the selected regression tests whose result could be affected

Table 5.2: Configuration Preparation Cost(hours) and Benefits(%)

Comp.	Strategy 1	Strategy 2	Strategy 3	Save-2	Save-3
SVNKit	2194.4	1863.9	1050.0	15.1	52.2
Serf	12.1	9.6	5.0	20.7	58.7
total	2206.5	1873.5	1055.0	15.1	52.2
Anjuta	2311.1	2036.1	327.8	11.9	85.8
Ns3	2330.6	2055.6	438.9	11.8	81.2
Bluefish	2500.0	2219.4	591.7	11.2	76.3
XChat	2972.2	2700.0	1072.2	9.2	63.9
XBMC	2344.4	2080.6	411.1	11.3	82.5
total	12458.3	11091.7	2841.7	11.0	77.2

by the code changes. Pre-built environments are reused to reduce the component build times.

We considered two user components, *Subversion* and *Serf*, from Figure 5.6. The components rely on the three provider components described above and also have default regression test suites. The regression tests were performed for fixed versions of the user components (*Subversion 1.8.1* and *Serf 1.3.0*) on the days when there were code changes for at least one of the provider components.

During the one year time period, there were 80 *APR* revisions, 148 *Openssl* revisions and 221 *SQLite* revisions. From all those revisions, we had to build and test *Subversion* 241 times and *Serf* 148 times. We now demonstrate four observed benefits from running regression tests of user components when a provider component changes.

Detecting faults in provider components: Regression tests for user components can reveal faults in provider components, and the fault-revealing test cases of the user components can be carved as new test cases of provider components. Techniques have been developed that potentially enable automatic carving of such test cases [41].

One example we found was that the test case `wc-queries-test` of the *Subversion* failed when it was built with *SQLite* revision `d7a25cc797`. The error occurred because a series of valid queries to *SQLite* returned errors.

We manually carved out the queries and created a unit test for *SQLite* and confirmed that the test case exposes the identical fault. Even though the fault was fixed in later releases, this example suggests that our automatic regression testing process can be used to detect faults relevant to provider components quickly, and also to produce new test cases that can detect the faults, thereby contributing to enriching the test suites of the provider components. Moreover, developers of other user components can also benefit from finding such faults because they are informed of the faults and can avoid spending time to find out the causes.

Discovering problems in accessing provider components: When changes in a provider component cause problems in building and testing user components, the collaborative testing process can be used to notify the provider component developers of the problems, so that they can use the information to pinpoint the origins of the problems.

In our experiment we found that multiple test cases of *Serf* and *Subversion* failed with the error message: `Couldn't perform atomic initialization`, when they were built and tested with some revisions of *SQLite* – for example, revision `62225b4a4c`). A simple Web search result revealed that many *SQLite* users experienced the same problem. The problem occurred when the *SQLite* library was linked in an obsolete way that was no longer supported. If *SQLite* developers had

been informed of the problem quickly, they could have fixed the problem, or at least could have updated user documentation so that users could be made aware of the problem.

Discovering faults in user components or in their test suites: When user components are built with a new provider component version, running regression tests for the user components can often reveal faults in their own test cases.

For example, *Subversion*'s test cases written in Python encountered unhandled exception errors, when *Subversion* was built and tested with specific *SQLite* revisions (e.g. revision `6f21d9cbf5`). This example suggests that the quality of user components and their test suites can be improved if and when our collaborative testing process is adopted by provider and user component developers.

Reducing the number of regression tests to run: We also observed that maintaining a mapping between the individual test cases of user components and code coverage information for provider components can greatly reduce the number of test cases that must be rerun when a provider component changes. When the changed part of the provider component is not previously covered by a regression test, we don't necessarily need to rerun that test. User component testers can share their unit-test coverage data for provider components in Conch, and such a mapping can be easily obtained by analyzing the coverage data.

Our experimental result is presented in Table 5.3. The "Rerun-Required Updates" row in the table shows the percentage of provider component revisions that caused rerunning the regression tests of its user components, compared to the num-

Table 5.3: Regression Test Selection Results

	Subversion		Serf	
	APR	SQLite	APR	Openssl
Rerun-Required Updates	29%	72%	9%	55%
Reduced Test Suite Size	80%	59%	98%	30%

ber of revisions that contain source code changes. As we can see, when the source code of *APR* changes, the regression tests are triggered in only 29% of such changes. The "Reduced Test Suite Size" row shows the average percentage of selected regression tests of user components that must be rerun, compared to the total number of regression tests. In the 29% cases when changes in *APR* triggered regression tests of *Subversion*, we don't need to return the whole regression test suite of *Subversion* either. On average, only 80% of the regression tests need to be rerun. From Table 5.3, this trend also exists for other evaluated components. It is evident that testers can save considerable effort on regression testing if they share the coverage information across components and properly use them for regression test selection.

5.6 Summary

As a step toward making collaboration between testers of component-based software systems easier, we have developed infrastructure and support tools, which include a model to specify test *environments*, a sharing repository for exchanging test data, an initial implementation of a collaborative testing process, and an empirical evaluation of the process. The model for test *environments* can accurately capture the hardware, system and inter-component relationships for build and test-

ing processes, so that test data shared between testers are comparable. The data sharing repository enables test tools to easily store or retrieve test data by querying the repository. By applying the developed approach to an example testing process, we have shown that developers not only saves significant time for build and functional testing, but also can improve regression test effectiveness.

Chapter 6: Coordinated Collaborative Testing Process

In the last chapter, I introduced the collaborative testing infrastructure that consists of the *Conch* data sharing repository and the environment differencing engine *Ede*. I also built an ad-hoc collaborative testing process utilizing Conch and Ede for data sharing. This process was designed to work in a greedy fashion without careful coordination between developer groups. However, in today's continuous regression testing processes, multiple groups may start to conduct regression testing at every component update, if their components are affected by the update. This is a rational choice for the groups because the configurations that contain the new component version are newly introduced and the groups intend to minimize the time in which potential compatibility faults are exposed to their user community. However, the strategy increases redundancy in test effort spent by the groups, especially if there are inter-component dependencies or if the component is shared by multiple groups.

In this chapter, I present a *coordinated collaborative regression testing process* for multiple developer groups, with the objectives of reducing the overall test redundancy across the groups as well as minimizing the time in which compatibility faults are exposed to the user community. The process involves a test scheduling

and notification mechanism across developer groups, so that each group is made aware of the configurations under test by other groups, enabling the groups to avoid performing redundant tests. I apply this process to a set of software systems with shared components in an Ubuntu distribution, emulate the application of the process over the 2-year history of the component development, and evaluate the cost and effectiveness of the process. Our experiments show that the coordinated collaborative testing process can greatly reduce test redundancy and can discover cross-component compatibility faults quickly.

6.1 Coordinated Collaborative Testing Process

We first outline notification-based test coordination, and then describe the detailed decision algorithm to distribute testing tasks to different developer groups, based on the availability, credibility and the performance of developer groups.

6.1.1 Notification Scheme for Coordinated Collaborative Testing

The *Conch* test data sharing repository not only maintains the dependency relationships between components, but also monitors the source code repository of the components to track their update releases [7]. For the purpose of discovering compatibility faults as soon as possible, whenever *Conch* sees a new version of a component, the developer groups of all user components of the updated component are notified, and they immediately start testing the new regression configurations. But because no group has yet tested or shared regression configurations containing

the new component version, the developer groups will not find reusable test results or configurations in *Conch*. Therefore, multiple groups will start testing identical configurations locally because they still need to minimize compatibility fault exposure time. As a result, the groups will end up performing redundant tests. That is, ad-hoc collaborative testing without proper coordination will waste testing time and testing resources of the developer groups.

To avoid the redundancy yet still achieve efficient regression testing, we enhance the notification scheme used in *Conch* to support coordination across multiple developer groups. This is different from ad-hoc collaborative testing in two aspects. First, for a new component version release, *Conch* notifies the affected user component developer groups to start testing the shared portion of regression configurations, in an order determined based on the availability, past test performance, and the failure rate of the groups. Second, if a set of new regression configurations that contain the new version is assigned to a developer group and is currently being tested, *Conch* monitors the test status, and notifies other groups of the status, if they request it. The groups can wait for the result to become available, or start testing the configurations locally. If they choose to wait, *Conch* will notify them when the result is ready. This scheme allows developer groups to conduct testing independently, and make their own decisions about whether or not to perform redundant tests.

6.1.2 Strategy for Coordinated Collaboration

When a component is shared by multiple developer groups and a new version of the component is released, sets of regression configurations defined for its user components have to be tested by the groups. Because the component is shared, there must be overlaps in the regression configuration sets, and the overlaps – a set of partial configurations – must be tested first. *Conch* selects one of the developer groups to test those partial configurations without causing test redundancy, based on the following factors:

- **Availability**: a binary value that indicates whether a developer group can immediately start testing a set of new regression configurations
- **Performance**: how fast a developer group can complete testing on their testing resources
- **Reliability**: how likely a developer group can complete assigned testing tasks

The **performance factor** of a developer group G is defined as the ratio of the execution time required to run a sample test suite using the testing resources of the group and the resources at the *Conch* repository, as shown in Equation 6.1.

$$PF(G) = \frac{T_G}{T_{Conch}} \tag{6.1}$$

We next define the **test failure rate** of a developer group G to quantitatively measure the reliability of the group. It is defined as the ratio of the number of failed test suite executions and the total number of test suite executions by the group.

$$TFR(G) = \frac{FC_G}{TC_G} \tag{6.2}$$

In Equation 6.2, TC_G is the total number of test suite execution requests that have been assigned to the group G, and FC_G is the number of test suite execution requests that failed to complete successfully. Reasons for failure to run a test suite may include abnormal termination of the test suite execution and failure to report test results back to *Conch* (e.g., because the test developer resource crashes, or loses its network connection), but does not refer to the success or failure of individual test case executions.

Based on the performance factor and the failure rate of a developer group G, we define the **Expected Performance Factor (EPF)** of the group as:

$$EPF(G) = \frac{PF(G)}{(1 - TFR(G))} \tag{6.3}$$

The *EPF* value will be small when both the performance factor value and the failure rate are small, and *Conch* prefers to distribute testing workload to a group with the smallest *EPF* value.

When a provider component is updated, we first determine the user components for which functionality might be affected by the updated provider component. Then we compute the regression configurations for the user components and also compute the overlaps between the configurations. The overlaps are a set of partial regression configurations on which the updated component has to be built and run without any faults. A developer group selected by applying Algorithm 2, will then

be requested to build and test the updated component over the partial configuration set.

```
Algorithm 2: CoordinateTester(C, CDG, A, PFs, FRs)
   Data:
   C: updated provider component
   CDG: component dependency graph
   A: availability of groups
   PFs: performance factor values of groups
   FRs: failure rate values of groups
 1 groups ← available direct user comp. developer groups ;
 2 sort groups by EPF ;
 3 while groups ≠ ∅ do
 4     group ← groups.getNext() ;
 5     result ← assigntask(group, C) ;
 6     update FR of the group ;
 7     if result == Success then
 8         update result in Conch ;
 9         Conch notifies subscribers of C's results ;
10         break ;
11     end
12 end
```

Algorithm 2 first identifies the developer groups of direct user components of the updated component C and eliminates the groups that cannot start regression testing immediately.[1] The candidate groups are sorted by the *EPF* values and then the group with the smallest *EPF* value will be requested to test C over the given regression configuration (Line 5). If the group completes (or fails to complete) the test, the **FR** value of the group will be updated accordingly.

If other groups request the result from testing C over the regression configuration while the test is under execution, *Conch* simply notifies the groups that the

[1] Developer groups of indirect user components are not considered because they can reuse the results produced by a direct user component developer group.

result is not yet available.

The groups can choose to run the same test using their resources, or instead wait for the result by subscribing to the test in *Conch*, and then run other tasks. If the test execution is completed, all groups interested in the test result will be notified (line 7-11). Otherwise, the next group in the sorted list of groups will be assigned to execute the test.

By applying the algorithm, our experiments will show that *Conch* can coordinate multiple collaborating developer groups, while minimizing both redundant test effort across the groups and the exposure time of compatibility faults introduced by component updates.

6.1.3 Regression Testing based on Cross-Component Coverage

We have presented a strategy to coordinate multiple developer groups, while avoiding redundant test effort. However, in the end we are still running full test suites of all user components that might be affected by the updated provider component – i.e., if there are user-provider relationships between the components in a CDG. We showed in Chapter 5 that developers can save test effort up to 70% by selectively running regression test cases based on the mapping between the individual test cases of user components and the code coverage of provider components.

In this part of the dissertation, coverage-based regression testing is conducted at two different granularity levels. If *Conch* maintains the code coverage mappings between each user component test case and each provider component, only a subset

of the test cases that cover the updated regions of the provider component must be run. If *Conch* maintains the mappings between the test suite of a user component and each provider component and if a provider component update is relevant to one or more test cases of the user component, we rerun the whole test suite at a provider component update. We describe in the next section why *Conch* maintains only coarse-grained mappings.

6.2 Experiments

In this section we evaluate the effectiveness and performance of the coordinated collaborative testing process presented in Chapter 6.1. We selected a set of subject components, obtained and sampled their evolution history, emulated the regression testing processes that would be performed by the developer groups of the components, and evaluated the performance and the effectiveness of our coordinated collaborative testing approach versus other approaches.

6.2.1 Subject Components

We picked twelve subject components (i.e., 12 developer groups) from the Ubuntu platform for our experiment. The components and their dependency relationships are shown as the CDG in Figure 6.1. We also obtained the update history of these components over roughly a one and half year period between October 2013 and March 2015. The subject components fall into various categories, including an interpreter (Python), an encryption library (OpenSSL), database systems (SQLite,

Table 6.1: Subject Components

Component	Description	Versions
Bzip2	high-quality, open-source data compressor	6
Zlib	compression library	3
Tcl	a dynamic programming language	6
Openssl	open source toolkit for SSL/TLS	18
SQLite	in-memory SQL database engine	15
Python	object-oriented programming language	5
BerkeleyDB	library for embedded database	4
LibXML2	XML C parser and toolkit of Gnome	10
Ns3	discrete-event network simulator	2
XBMC	open source home theater software	2
Subversion	version control system	26
Ubuntu	operating system	3

BerkeleyDB), system utilities (Bzip2, zlib), and a GUI application (XBMC). Three Ubuntu releases are considered. Table 6.1 contains brief descriptions of the components and the number of versions of each component.

The source code released by component developers is included as-is in the Ubuntu distribution, but in many cases the components are customized by Ubuntu developers to address compatibility issues. The developers maintain and update the code using version control systems like Bazaar [42] or Subversion [43]. Figure 6.2 shows the 87 total component versions released during the test period, ordered by day from the start of the test period.

6.2.2 Testing Strategies

Developers are pressured to complete testing their components with a limited amount of testing time and resources, and such pressure drives developers to conduct testing over only a sampled subset of configurations.

Among many different sampling strategies, one **naive** but commonly used

Figure 6.1: Subject Components for Continuous Collaborative Testing

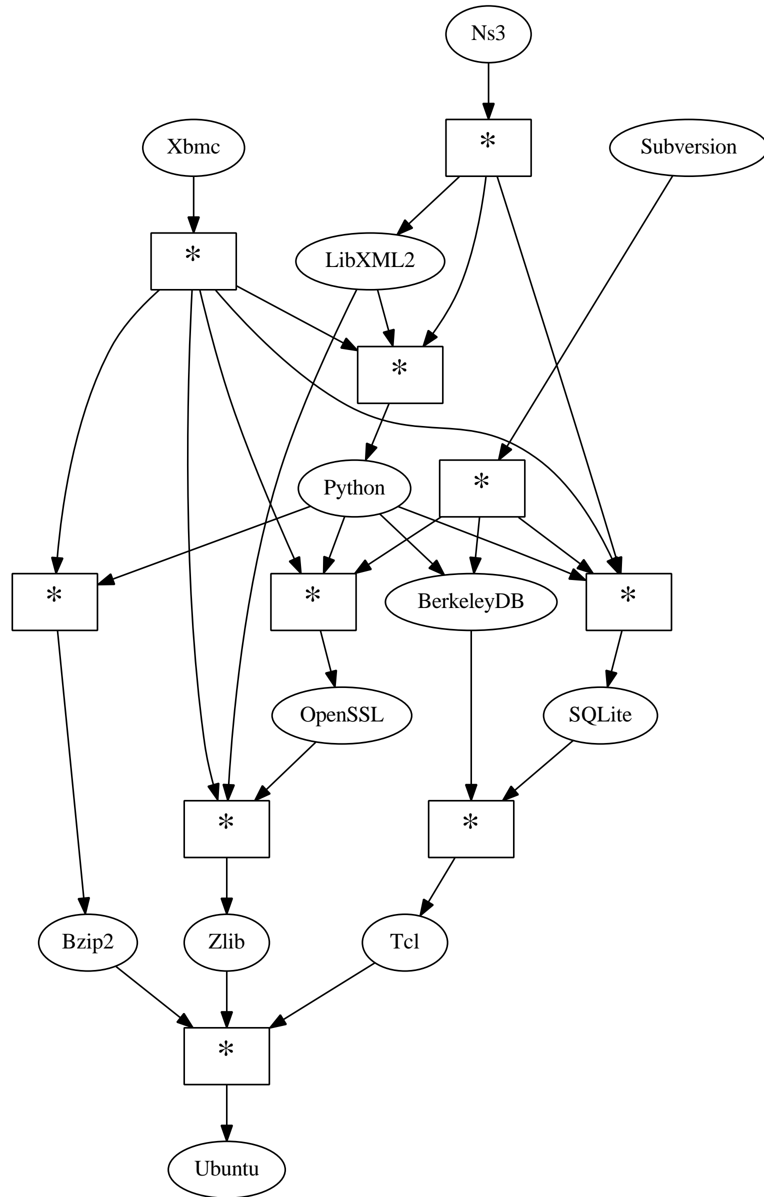
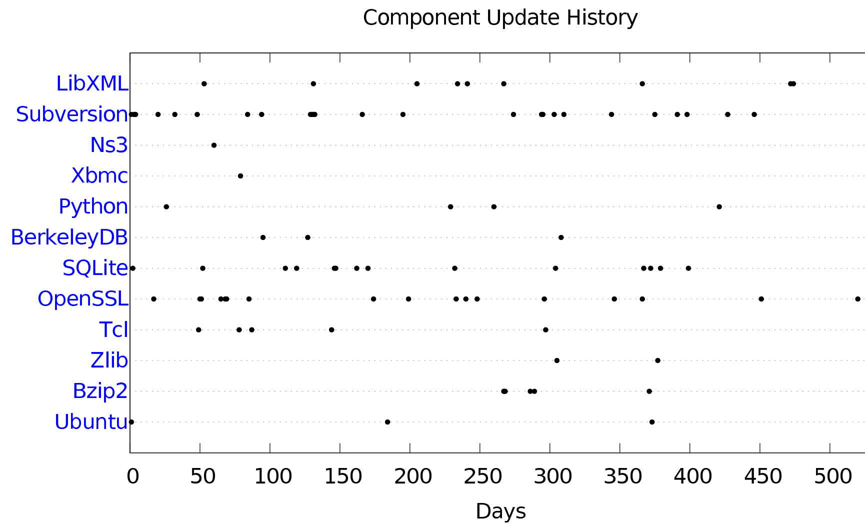


Figure 6.2: Subject Components Update History



strategy is to run the test suite of a component under development over a set of regression configurations of the component, where each configuration contains the latest version of all provider components. Developers of a component then compute regression configurations and run its test suite when they release a new version of their component. If we apply this strategy to the component update history in Figure 3.2, for example, the developers of the component B would test B_2 and B_3 over the regression configurations $\{B_2, E_1, G_1\}$ and $\{B_3, E_2, G_1\}$ respectively, and the developers of D would test D_2 over $\{D_2, F_1, G_1\}$, when D_2 is released. However, A_1 would never be tested over the configurations that contain new provider component versions (e.g., $\{A_1, B_2, D_1, E_1, F_1, G_1\}$), because there is no update record of A – i.e., in this strategy, the developers of A never monitor the changes in A's provider components and simply assume that A will function correctly over the configurations.

The second strategy we consider is that developers of a component constantly monitor the updates of all provider components, and run the test suite of their

components whenever a new provider version is available. For this strategy, we assume that developers test their components in an isolated way without sharing any test results. We call this strategy **eager testing** and this would be the practice when developers want to perform very thorough and timely compatibility testing over new provider components. If all developers adopt this strategy, each component will be tested over all its regression configurations, but the downside is that developers will end up performing redundant tests, since different groups do not coordinate their testing of overlapping components. In the running example in Figure 3.2, the component `A` will be tested over all 21 regression configurations with the eager testing strategy, and therefore the developers of `A` can quickly notice if a compatibility fault is introduced by including a specific provider component version in a regression configuration. However, the developers of `B` will also test `B` over all its regression configurations. In total, 55 regression configurations will be considered for testing by the developer groups, and clearly there will be a significant amount of overlap in the test effort expended by the groups.

The third strategy is **ad-hoc collaborative testing**. As described in Chapter 6.1.1, developers can aid each other by sharing test data through the *Conch* repository. In this strategy, developers always query *Conch* first to search for reusable test data. We consider three variants of ad-hoc collaborative testing. The first variant is to maximize the reuse of test data, by serializing the work required for testing each regression configuration between developer groups. The second variant is to minimize fault exposure time by allowing all developer groups to start testing their regression configurations immediately after each provider component update. In the

last variant, developers also apply the *coverage-based test case selection* technique described in Chapter 6.1.3, in addition to the second variant.

In our experiments, we collected the cumulative testing time and the maximum fault exposure time to compare the strategies above and the coordinated collaborative testing strategy described in Chapter 6.1.2.

6.2.3 Experimental Setup

Virtual machines (VMs) are used to install components contained in regression configurations, and then execute their test suites. Each VM is configured to have two virtual CPUs, 4GB of virtual memory, and 80GB of virtual disk space. Ubuntu is used as the operating system and all VMs are hosted on a private cloud cluster running OpenStack [44]. Default test suites provided by the original component developers are used to test the functionality of the installed components, but we excluded a subset of the full *BerkeleyDB* test cases because these test cases took too long (more than a week) to finish. They are designed for stress testing instead of functional testing, and including them will bias our experimental result to a specific component. [2]

To replay the component update history shown in Figure 6.2, we first performed **eager** testing for the top-level components in Figure 6.1. That is, we did all the test activities that must be performed by the 12 developer groups, and measured

[2]The test suite execution times vary widely between components. For example, the default test suite of *bzip2* only contains 6 test cases, each taking less than a second. On the other hand, *subversion* and *BerkeleyDB* have comprehensive test suites that take hours to days.

the time required to install components and run their test suites. The results from test case execution are also recorded. The test data acquired from eager testing is then reused to simulate the tests for the other testing strategies.

For *coverage-based test case selection*, we also maintain the coverage for each user/provider component pair. For example, we collect the *OpenSSL* (the provider) code regions covered by running the test suite of *XBMC* (the user). If no code region is covered, we do not need to retest *XBMC* when a new *OpenSSL* version is later released. The coverage mappings are updated when a new version of *XBMC* or Ubuntu is released. *Gcov*, the coverage collection tool of the GNU compiler collection [3],was used to collect the coverage information.

The performance of computing resources at multiple developer sites are assumed to be heterogeneous. We used a Gaussian distribution with mean value 1 to model the performance factor distribution, and performed experiments using 5 distributions each with different standard deviation values between 0.1 and 0.5 (See Table 6.2). We also need to model test failure rates for different developer groups. We assume that a developer group that successfully completed executing a test suite within a pre-defined time to completion would have a higher probability to succeed again at the next test request, and also assume that the inverse holds. This characteristic is modeled by using the test failure rate of a group to estimate the time to the next failure. Each time a developer group starts executing the test suite of a component, we generate a random value from an exponential distribution based on the current test failure rate of the group as an input. The value represents the

[3]http://gcc.gnu.org

expected time to the next failure. If the value is greater than the pre-defined time required for executing the test suite, we report the test execution is a success. The test failure rate is adjusted after each test execution. The initial failure rate is set to 0.1 for all developer groups.[4]

6.2.4 Experimental Results

Given the CDG in Figure 6.1 and the update history in Figure 6.2, there are 87 regression configurations. However, there was a compatibility fault between *OpenSSL* and its user components when testing the regression configurations generated by 9 component update events. The failures made all other user components untestable. So in the following results that compare test execution times across testing strategies, we used the results obtained by testing components over the 78 remaining configurations. In this section, we are interested in answering the following research questions:

1. **RQ1:** How efficient is the coordinated collaborative testing strategy compared to other strategies?
2. **RQ2:** Is the coordinated collaborative testing strategy effective in revealing cross-component compatibility faults?

[4]We also tried other initial failure rate values, and did not observe a significant impact on our results, unless the initial failure rate was unreasonably high for everyone.

6.2.4.1 Comparing Cumulative Test Execution Time

In order to answer RQ1, we compared the cumulative test execution times required to test components over the regression configurations by all developers for the different sampling strategies. We added up the times all the individual groups spent to install components and run their test suites. Table 6.2 shows the cumulative time (in hours) when different testing strategies are used. For each strategy, we show multiple results obtained by using different performance factor distributions with the 5 different *standard deviation* values.

In Table 6.2, we find that **naive** testing has a very short cumulative time. This is because when using naive testing, only 75 unique regression configurations are covered at all tester sites, while all the other strategies cover the same set of 377 unique regression configurations. At the other extreme, **eager** testing took the longest total time, because developer groups test their components in isolation, not removing any redundancy. With the **ad-hoc** collaborative testing strategy, the cumulative time is reduced to about 30% of **eager** testing, if developers prefer to maximize the test data reuse (*Ad-hoc max reuse*). However, the time savings compared to eager testing is negligible, if developers prefer to minimize the exposure time of latent faults (*Ad-hoc min exp. time*). We see better results when the coverage-based test case selection is also applied to(*Ad-hoc min exp. time, cov-sel*), because developers can skip executing many test cases based on the cross-component code coverage information. The coordinated collaborative testing strategy (*Coordinated, cov-sel.*) performed the best, and reduces the cumulative time to roughly 9% of the

Table 6.2: Cumulative Time in Testing Strategies (in hours)

	Standard Deviation for PF				
	0.1	0.2	0.3	0.4	0.5
Naive testing	73.1	73.6	73.5	73.7	73.5
Eager testing	593.9	596.6	592.4	592.5	593.3
Ad-hoc max reuse	177.4	178.0	177.3	177.8	177.6
Ad-hoc min exp. time	574.7	577.4	575.3	574.2	575.3
Ad-hoc min exp. time, cov-sel.	127.7	128.1	127.1	126.4	127.5
Coordinated, cov-sel.	54.4	55.1	54.5	55.3	54.6

time required for **eager** testing. The strategy even outperformed the **naive** testing strategy, because it coordinates developers to not spend test effort unnecessarily. Furthermore, *Coordinated, cov-sel.* can help developers find compatibility faults earlier, as we now describe.

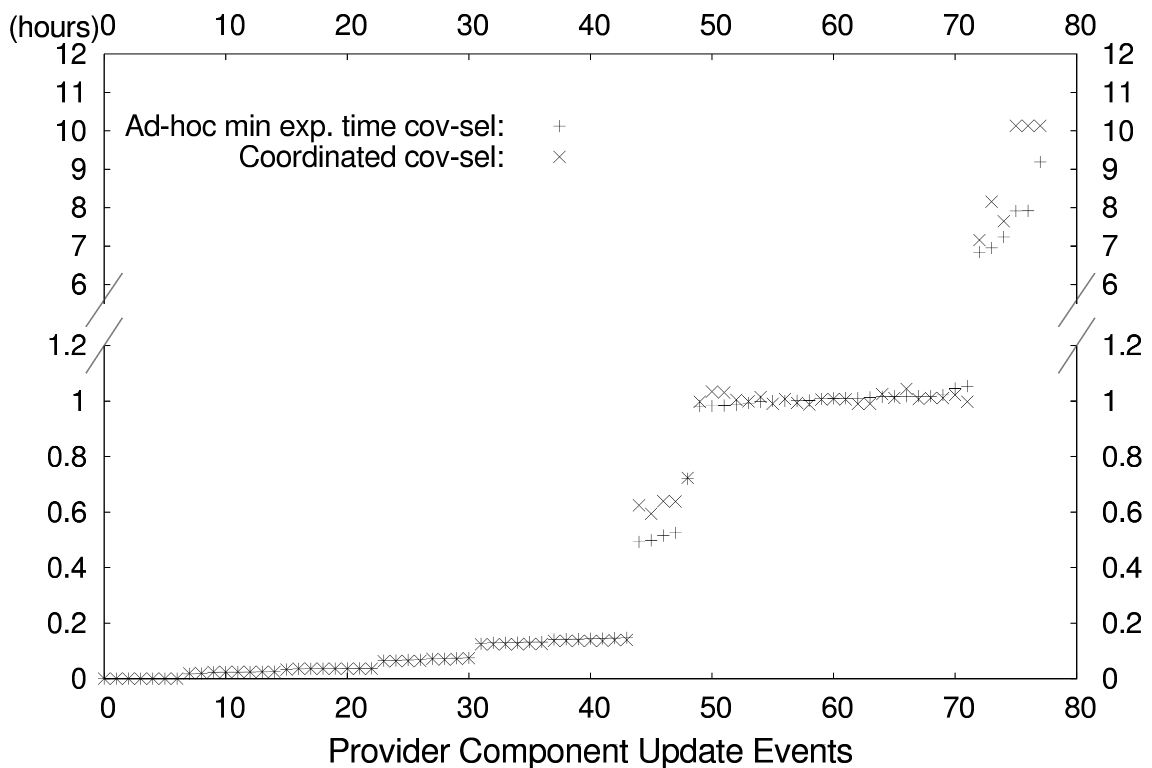
6.2.4.2 Comparing Maximum Fault Exposure Time

The *cumulative test execution time* represents the overall test effort across multiple developer groups, but it is also important to reduce the time until a compatibility bug can be discovered. We measured the **maximum fault exposure time**, which is the maximum time until every compatibility fault introduced by a provider component update is discovered, assuming that the fault can be discovered by testing components over regression configurations computed at the update. A smaller value means that faults are discovered earlier.

In Figure 6.3, we compared the maximum fault exposure times obtained by running the regression testing process for the 78 provider component updates, for two testing strategies that performed very well in the previous experiment: (1) the third variant of Ad-hoc collaboration (*Ad-hoc min exp. time, cov-sel*), and (2) the

coordinated collaborative testing strategy (*Coordinated, cov-sel.*). The x-axis represents the 78 provider component update events ordered by the fault exposure time. As described previously, we considered a regression configuration in this experiment only if we could install all components contained in the configuration, and also had to be able to complete running the test suites of the components. The y-axis shows the estimated maximum fault exposure time (in hours).

Figure 6.3: Maximum Fault Exposure Time



Maximum fault exposure time obtained by running the regression testing process with the strategies *Ad-hoc min exp. time, cov-sel* and *Coordinated, cov-sel.*

We observe that the fault exposure time is very short for roughly half the component updates. In fact, for 7 updates (*bzip2* and *zlib* updates), we did not need to test any user components, because both strategies use the coverage-based test case selection and the updated code regions of the components were not covered

by running the test suites of the user components. We observe that the maximum fault exposure times are similar between the strategies, and also that the ad-hoc strategy shows a little bit better results for a few updates. This is because multiple developer groups simultaneously test shared components contained in a regression configuration with the ad-hoc strategy. In contrast, for the coordinated testing each component in the configuration is always tested by only one developer group, as discussed in Chapter 6.1.2. Test failures also contribute to making the difference larger, because *Conch* has to choose another group to retest the component if a group fails to complete an assigned testing task.

Overall, the results in Figure 6.3 and Table 6.2 show that developer groups can reduce redundancy in their test efforts within a software community by adopting coordinated collaborative testing, and the coordination does not delay the testing processes of individual developers.

6.2.4.3 Analyzing Cross-Component Compatibility Faults

In order to determine whether coordinated collaborative testing can be effective in revealing cross-component compatibility faults (RQ2), we analyzed faults that could have been discovered if the coordinated collaborated testing had been performed as part of a continuous integration practice, which is a core practice in agile software development [45]. We classify the faults captured in the subject components in our experiments into three categories, and discuss further in the following paragraphs.

First, coordinated collaborative testing can be used to discover *cross-component compatibility faults introduced by a provider component update*. One example fault discovered in our experiments is that *XBMC* and Python fail to work with a newer version of *OpenSSL* (1.0.1e-5). When the*OpenSSL* developers released a new version on 12/22/2013, users who installed the version experienced a fault with the error message: *"OpenSSL version mismatch. Built against 1000105f, you have 10001060"*. This fault was classified as a *critical* bug in the Debian bug tracking system. If coordinated collaborative testing had been performed before the release, the fault could have been fixed before being released to a user community.

Second, *user components can fail due to behavioral changes in provider components.* Provider component developers may change the behavior of externally visible APIs in a new version, without noticing that the changes could create compatibility faults with user components. For example, the *SQLite* developers changed the *progress_handler*() API code in version *3.8.4*. Although the new version passed all regression test cases, a test case in a user component test suite (in this case, the test case *test_sqlite* in the *Python* test suite.) captured the fault[5].

Third, *faults in a component can be discovered by user component developers.* Component developers often use the latest, but maybe unstable, provider component versions (or builds). If user component developers conduct coordinated collaborative testing continuously, they can aid provider component developers by running the test suite of the provider components. For example, two test cases, "test_urllib2net" and "test_urllibnet", access the *Python* document webpage during execution, but a

[5] http://bugs.python.org/issue18873

change in the page made the test cases fail[6]. In another example, a *Python* security update was applied to the Python core but not applied to all modules of all Python versions[7]. With coordinated collaborative testing, a few test cases in the "test_ssl" user component could find the faults.

In addition to being able to detect faults, coordinated collaborative testing via *Conch* can also help developers by providing the capability to reproduce the configurations that contain compatibility faults, as virtual machine images.

6.3 Summary

In this chapter, we have presented a coordinated collaborative regression testing strategy that makes use of a scheduling algorithm to distribute testing workload across multiple developer groups based on both the capability and the reliability of the different developer groups. Through a comparative study against **naive** testing, **eager** testing, and **ad-hoc collaborative** testing, we have demonstrated that coordinated collaborative regression testing can help component developers quickly discover compatibility faults while also reducing redundancy in the total test effort expended by the developer groups. We also showed examples of the kinds of compatibility faults that can be exposed by adopting coordinated collaborative testing as part of a continuous integration process.

[6]http://bugs.python.org/issue21115, and https://bugs.python.org/issue20939

[7]https://www.python.org/dev/peps/pep-0476/

Chapter 7: Conclusions and Future Work

This chapter concludes my dissertation by reviewing the thesis research and its contributions. Future research directions are also discussed.

7.1 Thesis and Contributions

The thesis I support in this dissertation is: *By avoiding redundant work, collaborating across testing processes, and using information obtained through testing multiple related software components, testers of shared components can not only save test effort, but also improve the test effectiveness of each component as well as each component-based software system.* The goal of my thesis research is to explore the types and amount of overlaps and synergies that may exist in the testing processes of shared software components, and to develop tools and techniques that rely on that information to improve testing efficiency as well as quality of components. The contributions made by this dissertation include:

A collaborative testing infrastructure

Sharing test data is the core functionality required to support collaborative testing. From the result of our empirical study, component developers can save significant

effort by reusing testing results and artifacts that are shared between developers, and improve the test quality by utilizing the shared information. For this purpose, we developed the *Conch* web service based data sharing repository to enable automated testing tools used by isolated component developers and/or testers to share their testing data, and the *Ede* environment differencing engine to support scalable caching and sharing of portable testing environment in the form of virtual machine images. Component developer groups can easily modify their existing automated testing tools to use our infrastructure to enable collaboration.

When performing regression testing of systems that share components, we observed that a large amount of test effort was spent on building the same partial testing environments and running redundant functional test suites in the same testing environments. To evaluate the effectiveness of our infrastructure, we selected two sets of software systems that share provider components, simulated their regression testing processes relying on our infrastructure for collaboration, using one year of historical component revision data. Simulation results show that a large amount of testing time spent by testers of these systems can be saved through collaboration, without missing any faults that were discovered when testing these systems in isolation.

Two collaborative testing processes

Based on the testing data sharing tools, we further developed two collaborative testing processes that characterize two user scenarios. The first is an **ad-hoc** collaborative testing process. In this process, component developers follow their own

schedule to perform compatibility testing locally. Their automated testing tools query *Conch* before building any testing environment, or running any functional test suite. If such data have already been shared by other component developers, they will simply reuse them. Otherwise, they still build their environments and run the tests locally, and optionally share them to *Conch*. The second is a **coordinated** collaborative testing process. In this process, testers run their regression test suites as soon as their provider component is updated, for the purpose of minimizing the time window of finding any possible compatibility faults that may have been introduced by the new provider component version. *Conch* actively assigns regression testing tasks among affected component testers to avoid redundant effort. For better performance, *Conch* utilizes performance histories of individual testers to decide to whom a task will be assigned.

To evaluate the effectiveness and efficiency of these two processes, we ran simulations over historical data of real-world sets of components, and compared using our collaborative testing processes versus testing everything in isolation. A large amount of test effort was saved in both cases, and we also found that 1) compatibility faults which were not identified by isolated testing were discovered by our collaborative testing processes; and 2) using the coordinated collaborative testing process, compatibility faults were found much faster than they were in the real-world setting. Thus, the two collaborative testing processes have proved to ben both efficient and effective.

7.2 Future Work

My dissertation research is an initial study to search for benefits of collaborative testing. Several possible extensions and improvements can be made based on the current work.

Improve Scheduling Algorithm In the coordinated collaborative testing process, *Conch* can schedule a common testing task to one of the affected component developers to avoid redundant testing. We considered *availability, reliability* and *performance* in a simple model, and used these factors to determine who the task will be assigned to. This coordinated collaborative testing process can be expanded upon in the future. First, a more refined tester model instead of the variable of testers' *performance factor* should be used by the scheduling algorithm. The new model needs to capture more real behaviors of testers, and should consider many other factors, such as network bandwidth and task assigning overhead. Second, in addition to minimizing redundancy, target optimization should also be taken into consideration. For example, testers may want to utilize all idle machines to finish testing all configurations as soon as possible. Third, individual testers should be able to specify their preference over configurations to test. Last, when multiple testers are available, we could assign different parts of the same test suite to different testers, so that the overall progress of testing all new configurations can be finished sooner, and the maximum exposure time of compatibility faults can be further reduced.

Modify Popular Tools to Support Conch

Our work used *Rachet* [33] as the example automated testing tool of isolated component developers for experimental simulation. However, this technique does not require testers to have a specific tool in order to enable collaborative testing. *Conch* provides a web service that can be easily supported by different tools with minimum modification. Nowadays, there are some widely used continuous testing platforms including *Jenkins* [26]. Jenkins provides various ways of connecting different testing tools via a wide selection of extensions and plugins. From the system perspective, we would like to explore how testing projects which are already using existing systems like Jenkins for their continuous integration testing can utilize our infrastructure for collaboration.

Our data sharing infrastructure uses virtual machine images to encapsulate prebuilt configurations, and relies on Ede to make sharing of prebuilt configurations efficient. Recently, light-weight container techniques are also used to pack applications and their dependencies, transport them across sites and platforms, and deploy them in various environments. Popular tools like Docker [46] can wrap up software components in a complete file system that contains everything they need to run, including code, runtime, system tools, system libraries, etc. In future work, we will consider using Docker to capture prebuilt environments, and share these environments in addition to virtual machine incremental files through Conch. Using Docker-wrapped environments will allow testers to easily rebuild their testing environments on various resources, including cloud resources.

Improve Tests of Individual Components

My dissertation research shows that testing of user components can test extra

parts of provider components that are not covered by the test suites of the provider components themselves. In the future, we would like to work on automatic test case generation for the provider components, so that if a user component test is accessing uncovered parts of a provider component, we can create a local test case automatically for the provider component. In this way, the coverage of the provider component is permanently improved. There are existing techniques that may be related to this topic. For example, Elbaum et al. developed techniques that generate unit test cases from system test cases [41]. However, their techniques still require specific language support, and there are still a lot of challenges, such as recording the access patterns from the user components to the provider components, and recreating the state of the provider component using a local unit test.

Improve Security and Consistency In the dissertation research we did not consider security issues like malicious collaborators or unreliable shared data, neither does Conch have a policy for scenarios when data shared from different testers conflicts. In the future, we can start addressing these issues in several ways. To limit the effects of incorrect testing results being shared from unreliable sources or malicious users, the *Conch* repository should have a mechanism to check inconsistency in the shared data. If any data shared by a user conflicts with existing data, the conflicting data should be marked and further verified, and users of these inconsistent data should also be notified. To prevent collaborators from gaming the collaboration policy and relying on others to finish their own testing tasks, the scheduling algorithm should be carefully designed and reviewed to make sure tasks are balanced between participating developers.

Bibliography

[1] Floris Erich, Chintan Amrit, and Maya Daneva. A mapping study on cooperation between information system development and operations. In Andreas Jedlitschka, Pasi Kuvaja, Marco Kuhrmann, Tomi Mnnist, Jrgen Mnch, and Mikko Raatikainen, editors, *Product-Focused Software Process Improvement*, volume 8892 of *Lecture Notes in Computer Science*, pages 277–280. Springer International Publishing, 2014.

[2] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[3] Ilchul Yoon, Alan Sussman, Atif Memon, and Adam Porter. Towards incremental component compatibility testing. In *Proceedings of CBSE '11*, pages 119–128, 2011.

[4] William E. Lewis. *Software Testing and Continuous Quality Improvement, Third Edition*. Auerbach Publications, Boston, MA, USA, 2nd edition, 2008.

[5] Leonardo Mariani, Sofia Papagiannakis, and Mauro Pezze. Compatibility and regression testing of COTS-component-based software. In *Proc. of ICSE '07*, pages 85–95, Washington, DC, USA, 2007. IEEE Computer Society.

[6] Teng Long, Ilchul Yoon, Adam Porter, Alan Sussman, and Atif Memon. Overlap and synergy in testing software components across loosely-coupled communities. In *Proc. of ISSRE'12*, Dallas, TX, USA, 2012. IEEE Computer Society.

[7] Teng Long, Ilchul Yoon, Atif Memon, Adam Porter, and Alan Sussman. Enabling collaborative testing across shared software components. In *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '14, pages 55–64, New York, NY, USA, 2014. ACM.

[8] Teng Long, Ilchul Yoon, Alan Sussman, Adam Porter, and Atif Memon. Scalable system environment caching and sharing for distributed virtual machines. In *Proceedings of the 2014 IEEE 28th International Symposium on Parallel and*

Distributed Processing Workshops and PhD Forum, IPDPSW '14, Phoenix, Arizona, USA, 2014. IEEE Computer Society.

[9] Christof Ebert and Philip De Neve. Surviving global software development. *IEEE Softw.*, 18(2):62–69, March 2001.

[10] Balasubramaniam Ramesh, Lan Cao, Kannan Mohan, and Peng Xu. Can distributed software development be agile? *Commun. ACM*, 49(10):41–46, October 2006.

[11] Andrew Begel and Thomas Zimmermann. Keeping up with your friends: Function foo, library bar.dll, and work item 24. In *Proc. of the First Workshop on Web2.0 for Software Engineering*, May 2010.

[12] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. Does distributed development affect software quality? an empirical case study of windows vista. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 518–528, 2009.

[13] Bogdan Korel and Ali M. Al-Yami. Automated regression test generation. *SIGSOFT Softw. Eng. Notes*, 23(2):143–152, March 1998.

[14] K. Taneja and Tao Xie. Diffgen: Automated regression unit-test generation. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 407–410, Sept 2008.

[15] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929–948, Oct 2001.

[16] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, pages 10 pp.–, Nov 2005.

[17] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, April 1997.

[18] Yanping Chen, Robert L. Probert, and D. Paul Sims. Specification-based regression test selection with risk analysis. In *Proceedings of the 2002 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '02, pages 1–. IBM Press, 2002.

[19] Alessandro Orso, Hyunsook Do, Gregg Rothermel, Mary Jean Harrold, and David S. Rosenblum. Using component metadata to regression test component-based software. *Software Testing, Verification and Reliability*, 17(2):61–94, 2007.

[20] Chengying Mao, Yansheng Lu, and Jinlong Zhang. Regression testing for component-based software via built-in test design. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, pages 1416–1421, New York, NY, USA, 2007. ACM.

[21] Seojin Kim, Sungjin Park, Jeonghyun Yun, and Younghoo Lee. Automated continuous integration of component-based software: An industrial experience. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 423–426, Washington, DC, USA, 2008. IEEE Computer Society.

[22] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 235–245, New York, NY, USA, 2014. ACM.

[23] Agneta Nilsson, Jan Bosch, and Christian Berger. Visualizing testing activities to support continuous integration: A multiple case study. In Giovanni Cantone and Michele Marchesi, editors, *Agile Processes in Software Engineering and Extreme Programming*, volume 179 of *Lecture Notes in Business Information Processing*, pages 171–186. Springer International Publishing, 2014.

[24] Atif M. Memon, Ishan Banerjee, Nada Hashmi, and Adithya Nagarajan. DART: A framework for regression testing nightly/daily builds of GUI applications. In *Proceedings of the 19th International Conference on Software Maintenance*, pages 410–419, Sep. 2003.

[25] Cruisecontrol. *cruisecontrol.sourceforge.net/*, 2010.

[26] Jenkins: an extendable open source continuous integration server. *http://jenkins-ci.org/*, 2013.

[27] Automatic testing of Debian-format packages. *launchpad.net/autopkgtest*, 2015.

[28] Sabrina Souto, Divya Gopinath, Marcelo d'Amorim, Darko Marinov, Sarfraz Khurshid, and Don Batory. Faster bug detection for software product lines with incomplete feature models. In *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, pages 151–160, New York, NY, USA, 2015. ACM.

[29] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

[30] B.P. Lamancha, O. Diaz, M. Azanza, and M. Polo. Software product line testing: A feature oriented approach. In *Industrial Technology (ICIT), 2012 IEEE International Conference on*, pages 298–305, March 2012.

[31] He Jifeng, Xiaoshan Li, and Zhiming Liu. Component-based software engineering the need to link methods and their theories. In *Proc. of ICTAC05, International Colloquium on Theoretical Aspects of Computing, Lecture Notes in Computer Science 3722*, pages 72–97. Springer, 2005.

[32] P. Brereton and D. Budgen. Component-based systems: a classification of issues. *Computer*, 33(11):54–62, Nov 2000.

[33] Ilchul Yoon, Alan Sussman, Atif Memon, and Adam Porter. Effective and scalable software compatibility testing. In *Proc. of ISSTA '08*, pages 63–74, New York, NY, USA, 2008.

[34] Launchpad: a software collaboration platform. *launchpad.net*, 2014.

[35] SourceForge: an Open Source community resource. *sourceforge.net*, 2014.

[36] Gregor Gssler, Sussane Graf, Mila Majster-Cederbaum, M. Martens, and Joseph Sifakis. An approach to modelling and verification of component based systems. In Jan Leeuwen, GiuseppeF. Italiano, Wiebe Hoek, Christoph Meinel, Harald Sack, and Frantiek Plil, editors, *SOFSEM 2007: Theory and Practice of Computer Science*, volume 4362 of *Lecture Notes in Computer Science*, pages 295–308. Springer Berlin Heidelberg, 2007.

[37] Ye Wu, Dai Pan, and Mei-Hwa Chen. Techniques for testing component-based software. In *Engineering of Complex Computer Systems, 2001. Proceedings. Seventh IEEE International Conference on*, pages 222–232, 2001.

[38] Web Services Description Language (WSDL) 1.1. *www.w3.org/TR/wsdl*, 2001.

[39] SOAP Version 1.2. *www.w3.org/TR/soap12-part1/*, 2007.

[40] Teng Long, Ilchul Yoon, Alan Sussman, Adam Porter, and Atif Memon. Scalable system environment caching and sharing for distributed virtual machines. In *Proceedings of the IPDPS Workshop on High-Performance Grid and Cloud Computing*, 2014.

[41] S. Elbaum, Hui Nee Chin, M.B. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *IEEE Transactions on Software Engineering*, 35(1):29–45, Jan 2009.

[42] Bazaar Version Control System. *bazaar.canonical.com/en/*, 2015.

[43] Apache Subversion: Enterprise-class centralized version control for the masses. *subversion.apache.org/*, 2015.

[44] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3):38–42, 2012.

[45] S. Stolberg. Enabling agile testing through continuous integration. In *Proceedings of the 2009 Agile Conference*, pages 369–374, Aug 2009.

[46] Docker: An Open Platform for Distributed Applications. *http://www.docker.com/*, 2015.