Enabling Collaborative Testing Across Shared Software Components

Teng Long[1], Ilchul Yoon[2], Atif Memon[1], Adam Porter[1] and Alan Sussman[1]
[1]UMIACS and Department of Computer Science, University of Maryland
College Park, MD, USA
[2]Department of Computer Science, State University of New York
Incheon, South Korea
[1]{tlong,atif,aporter,als}@cs.umd.edu, [2]icyoon@sunykorea.ac.kr

ABSTRACT

Components of numerous software systems are developed and maintained by multiple stakeholders, and there is significant overlap and synergy in the process of testing systems with shared components. We have designed and implemented infrastructure that enables testers of different components to share their test results and artifacts so that they can collaborate in testing shared components. We also develop an example collaborative testing process that leverages our infrastructure to save effort for regression testing of systems with shared components. Our empirical study of this process shows that collaborative testing of component-based software systems can not only save significant effort by sharing test results and artifacts, but also improve test quality of individual components by utilizing synergistic data shared among component testers.

Keywords

component-based software, software integration

1. INTRODUCTION

The "old school" of testing advocated "test everything on your own." This go-it-alone approach worked well for developer groups and organizations that implemented much of their software from scratch in isolation. Over the years, the practice of software development has changed; so too must the practices of testing. Rarely does a developer group or an organization develop software from scratch anymore. Instead, they rely on third-party software components, knitting them together to implement their system. However, when it comes to testing these systems, they continue to follow the old school approach.

In this paper, we posit that the paradigm shift to component-based software development has created numerous opportunities for sharing test effort. We exploit these opportunities in the context of an important software maintenance activity, *regression testing*. Our supposition is based on two characteristics of component-based systems that we discuss via the example shown in Figure 1 (we will discuss the nomenclature of the figure later in Section 2).

The first characteristic is that components in component-based software systems have relationships between them, i.e., some components *use* or depend on other components. Consider the top-level right-most shaded node in Figure 1 labeled *Subversion*, which relies on other "lower-level" 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, the higher-level components can inform the lower-level components about the context in which they are being used. Similarly, the lower-level components can inform the higher-level components relying on them 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 by multiple software systems. Consider, for example, that the *Apache Portable Runtime* library (*APR*) in Figure 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 [15], tests run on one low-level component could be extracted and applied to the other low-level components.

Our preliminary results [13] showed that testing top-level components such as *Subversion* induced line coverage of the lower-level components beyond that achieved by running the lower-level components' own unit tests. This implies that top-level components exercise lower-level components

CBSE'14, June 30–July 4, 2014, Marcq-en-Baroeul, France.

http://dx.doi.org/10.1145/2602458.2602468.

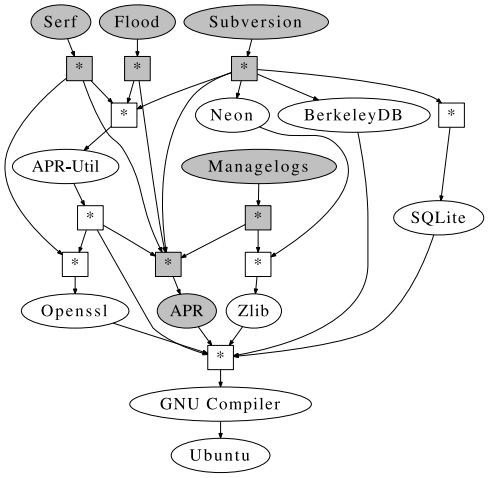

Figure 1: Systems with Common Components

in ways not anticipated by the testers of the lower-level components. We also saw substantial overlap in the line coverage induced by each individual top-level component. Moreover, despite the overlaps, each top-level component induced line coverage that was not induced by the other top-level component. Such observations and prior results suggest that **collaborative testing** of multiple component-based systems may provide benefits beyond that offered by individual component unit testing and beyond that provided by testing any single top-level component. Eliminating redundancies in testing across different components may also reduce costs without affecting test quality.

While the collaborative test processes we envision appear to be promising, current test approaches cannot support them effectively. First, there lacks a formal model of test environment of component-based systems that testers can use to describe their test environment, nor can testers exchange their test environment efficiently, which prevents testers from sharing and reusing their test results at all. Second, current tools lack methods and algorithms to guide the sharing of test efforts between provider-user pairs, or to improve local tests of components from test data generated by others. Third, collaborative testing will require efficient and easy-to-use mechanisms to automatically coordinate the test processes of different systems, for sharing and reusing test artifacts, and for comparing and merging independent test results.

As a step towards realizing collaborative test processes, this paper proposed a formal model for a test environment, presents a test process and support infrastructure for collaborative testing, and empirically explores the two opportunities we described within this process. The results of our study demonstrate that i) significant time spent on build and functional testing of component-based systems can be saved by reusing test results and leveraging pre-built environments shared from testing of other components; ii) hidden faults in both ends of the user-provider pair are discovered by analyzing shared test results, and local unit tests of provider components can be created from tests of user components; and iii) the size of the regression test suite to run can be greatly reduced when changes to a subcomponent only affect parts of the test suite.

The contributions of our work include:

1. Formal modeling of the environment that a component is to be built and tested in.
2. A web-service based data sharing repository that enables effective and efficient test data and environment sharing between testers.
3. Initial implementation of a collaborative test process over the support infrastructure we developed.
4. Empirical evaluation of the test processes over multiple component collections to explore the benefits of collaboration.

The rest of the paper is organized as follows. Section 2 presents background that forms the foundation for our current work. Section 3 describes our test data sharing infrastructure, and Section 4 explores the processes of collaborative testing using our infrastructure. We present experimental results of evaluating the benefits of such collaboration in Section 5, address related work in Section 6 and conclude in Section 7.

2. BACKGROUND

We now discuss our prior work on modeling component-based systems and efficiently build testing them across a large configuration space. We also summarize an initial empirical study that demonstrated the potential utility of collaborative testing.

2.1 Modeling Component-based Systems

We model component-based systems using a representation [21] that contains two parts: a directed acyclic graph called the *Component Dependency Graph(CDG)* and a set of *Annotations*. As shown in Figure 2, 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 2 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.

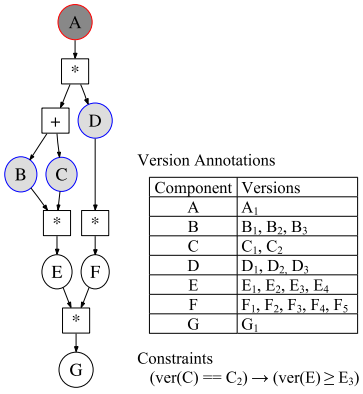

Figure 2: An Example System Model

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, the top-level components (*Serf, Flood, Subversion* and *Managelogs*) depend on different provider components. There are overlaps between 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 sub-graph, 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 spending redundant test effort that could be eliminated or advantageously redirected if all of these components were able to share their test data and artifacts.

2.2 Automatic Build 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 our prior work we designed a process and infrastructure called *Rachet* [21] 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 criteria, 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 is currently designed to be used to test a single software system at a time. 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.

2.3 Overlap and Synergy in Collaborative Functional Testing

In our previous work [13], we studied overlap and synergy achievable from sharing test data across multiple component-based systems. More specifically, we measured the line and branch coverage of a shared component and the spectrum of parameter values used to invoke methods of this component, both for executing the test suite of the shared component and for executing multiple test suites of components that depend on the shared component. Our analysis of the resulting test data showed that test cases designed and run by the higher-level components were individually less comprehensive than those of the shared component, but in some cases exhibited new behaviors or used unexpected test inputs not covered by the shared component's test cases. The results suggest that test data collected by component developers can be complementary. However, sharing such test data is only possible when the data to be shared is well-defined, and when a systematic way to share that data is available to component developers. In the remainder of this paper, we describe ways to define the desired test data and also discuss techniques and tools for sharing the data across multiple component-based systems.

3. DATA SHARING INFRASTRUCTURE

To support collaborative testing we have developed an automated data sharing infrastructure that provides support for creating test environments, for storing and sharing test data, and for efficiently managing and sharing test environments.

3.1 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 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 test efforts. Thus, we provide a notional definition of a test *environment* as follows:

Definition 1: An *environment* for a component to be 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.)

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

Figure 3: Simplified example environment description for a VM

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 3. 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 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*.

3.2 Data Sharing Repository via Web Services

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 4. 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 [19] and can be accessed via standard SOAP [17] protocols. Using the protocols, testers or other third-parties 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 test processes with those of other testers.

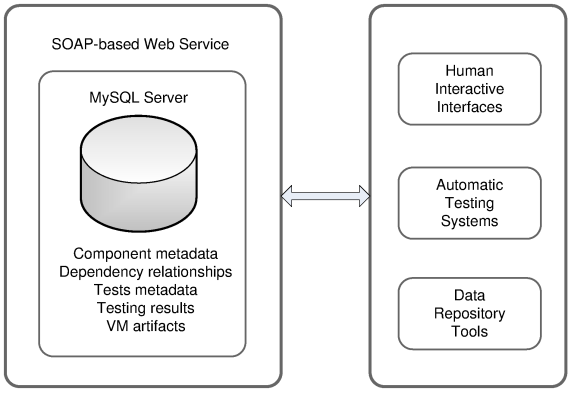

Figure 4: The Conch Data Sharing Repository

Conch Request	
command_name	getCDG
command_session	001
component	SQLite

Figure 5: Request for *SQLite* dependency data

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 paper, 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 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 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 and Figure 6 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, and the response is in the form of an XML file that contains the information in Figure 6.

3.3 Sharing Virtual Machines with Environment Differencing

Before building and testing a component, an *environment* that contains all its provider components must be prepared.

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

Figure 6: *Conch* response with *SQLite* dependency data

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) 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. More details about our work on *Ede* are described in [14].

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 2. 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 [21]. In Section 5, we describe the performance benefits from sharing pre-built environments.

[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.

4. COLLABORATIVE TEST PROCESS

In this section we describe a collaborative test process for component-based software systems implemented upon the data sharing infrastructure we introduced in Section 3. 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 accessing test data stored in the *Conch* repository and do not need to directly communicate with each other in order to benefit from the collaboration.

4.1 Testing Procedures for Component-based Systems

A component-based system can be considered as a top-level user component plus all the provider components where it depends. 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 7. 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 7 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, *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 Section 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.

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.

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* 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.

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
```

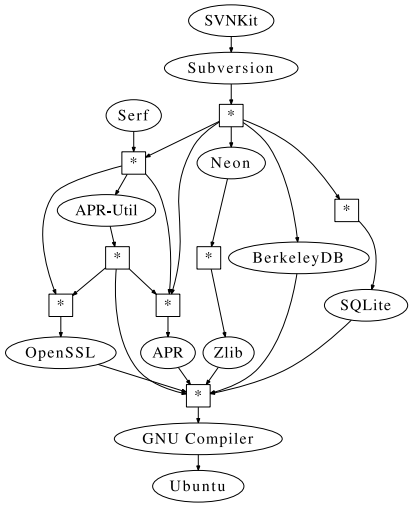

Figure 7: Subject Systems on Ubuntu for Collaborative Testing

5. EXPERIMENTAL RESULTS

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

In Section 5.1, we evaluate the benefits of the collaborative test process with two sets of top-level components that share provider components, as shown in Figure 7 and 8. While replaying the version release history of the components contained in the CDGs over a period of time, we conducted compatibility testing using *Rachet* [21] 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 Section 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 com-

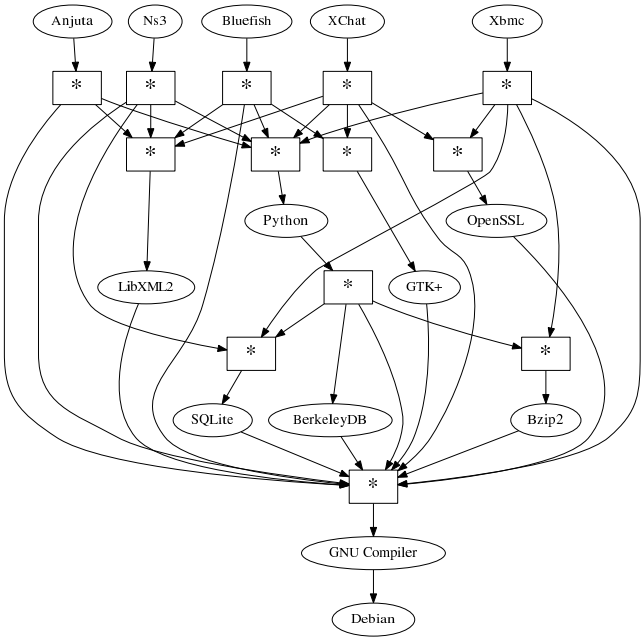

Figure 8: Subject Systems on Debian for Collaborative Testing

ponent 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* [11] as the automatic regression test client. We evaluate the collaborative test process with the version release history of the components in Figure 7 over one year.

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 Figure 7 and 8 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 7, 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 client library. In Figure 8, 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 1.

Table 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

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 [21]) criterion is used to compute configurations 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 functionally 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 result. **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 pro-

Table 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

duced 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 2 shows the total configuration preparation cost for each top-level component shown in Figures 7 and 8.

In Table 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*.

5.2 Continuous Collaborative Regression Test

In this section we replay the continuous development of three provider components, *APR*, *Openssl* and *SQLite*, contained in the combined CDG in Figure 7 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 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 7. The components rely on the three provider components described above and also have regression test suites with reasonable sizes. 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 [8].

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 test 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 test 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 test 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

Table 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%

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. With help from *Conch*, it is feasible for user component testers to share their unit-test coverage data for provider components, and such a mapping can be easily obtained by analyzing the coverage data.

Our experimental result is presented in Table 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 number 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 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 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.

6. RELATED WORK

Our work focuses on component-based software systems that share components. The components are developed by different groups; i.e., there is no central control over the component development. We maintain the information in a shared repository but in a distributed manner.

Researchers have emphasized the importance of tool support for collaboration between distributed teams [3, 4]. Bird et al. [4] reported that globally distributed software development within a single company may not perform worse (in terms of failures) than centralized development. In [3], 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.

Software researchers have also begun to examine the notion of self-organizing software development teams. For example, as social media gain increasing popularity, researchers have started to discuss the impact that social media has on software development, especially on enabling new ways for software teams to form and work together [2].

Distributed continuous quality assurance (QA) environments such as Dart [16] and CruiseControl [6] are systems for conducting continuous integration testing, which involves executing build and test 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.

Web services are another example of component-based systems where each service may be developed by different developer groups. Bai [1] developed a tool to test Web services by coordinating distributed test agents for conducting decomposed testing tasks, and Dallmeier [7] developed a tool to test the compatibility of Web applications on multiple browsers by identifying all functionally different states of the services at runtime. However, these approaches are designed for a single developer group. Our approach is focused on reducing the overall test cost and on deriving synergy from the collaborative testing across developer groups.

An important aspect of our approach is the shared repository of software and test artifacts. As software gets larger and more complex, many researchers have focused on analyzing and leveraging software repositories. For example, Zeller [18] has worked on analyzing repositories to better understand the reason systems fail. Xie has also studied the wide variety of information largely unused even though the information is stored in software repositories [10]. Researchers have also developed new data mining techniques that make use of software repositories to extract error patterns [12] or to understand correct API usage from code examples [20]. Such techniques could be used to analyze test results that are accumulated in the *Conch* repository.

Our work is broadly related to prioritization and selection of regression test cases. Elbaum et al. introduced different techniques for test case prioritization [9], with some of them based on code coverage information, which is similar to our regression test case selection technique. But those techniques are not for component-based software systems developed independently by multiple developer groups. Berries et al. have developed a tool called ProxiScientia [5] that helps to visualize the dependencies among collaborating software development teams, but that system does not provide functionality to facilitate collaboration.

7. CONCLUSIONS

Our work is based on the hypothesis that when two or more component-based systems use one or more common components, testers of such systems can lower test cost and improve test effectiveness by sharing test artifacts.

As a step toward making collaboration between testers of such 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 test 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 processes, so that test data shared between testers are compatible. The data sharing repository enables test tools to easily store or retrieve test data by querying the repository. We have shown that the example test process not only saves significant time for build and functional testing, but also improves regression test effectiveness.

The ultimate goal of our research is to enable collaborative testing across different testers and their test tools to avoid redundant work, as well as to improve test quality for all

component-based software developers. In order to accomplish the goal, we will continue to develop collaborative test processes that utilize the repository for testing component-based systems, and extend the repository to accommodate those processes. We will also design algorithms and methods to conduct deeper analyses of test data stored in the repository. Modifying existing frameworks such as *Rachet* to use the infrastructure is also a short-term goal for better coordination of test tasks across multiple components.

Acknowledgments

This work was partially supported by the US National Science Foundation (ATM-0120950, CCF-0811284, CNS-1205 501, CNS-0855055), the National Research Foundation of Korea (NRF-2013010695), and the MSIP of Korea (NIPA-2013-H0203-13-1001).

8. REFERENCES

[1] X. Bai, G. Dai, D. Xu, and W.-T. Tsai. A multi-agent based framework for collaborative testing on web services. In *Proceedings of the The 4th IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, and the 2nd International Workshop on Collaborative Computing, Integration, and Assurance*, pages 205–210, 2006.

[2] A. Begel, R. DeLine, and T. Zimmermann. Social media for software engineering. In *Proceedings of the FSE/SDP Workshop on the Future of Software Engineering Research*, November 2010.

[3] A. Begel and T. 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.

[4] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. 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.

[5] A. Borici, K. Blincoe, A. Schroter, G. Valetto, and D. Damian. Proxiscientia: Toward real-time visualization of task and developer dependencies in collaborating software development teams. In *Proceedings of the 5th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 5–11, June 2012.

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

[7] V. Dallmeier, M. Burger, T. Orth, and A. Zeller. Webmate: A tool for testing web 2.0 applications. In *Proceedings of the Workshop on JavaScript Tools*, pages 11–15, 2012.

[8] S. Elbaum, H. N. Chin, M. 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.

[9] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, Feb. 2002.

[10] A. E. Hassan and T. Xie. Software intelligence: the future of mining software engineering data. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 161–166, 2010.

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

[12] B. Livshits and T. Zimmermann. *Mining Software Specifications: Methodologies and Applications*, chapter DynaMine: Finding Usage Patterns and Their Violations by Mining Software Repositories. CRC Press, May 2011.

[13] T. Long, I. Yoon, A. Porter, A. Sussman, and A. Memon. Overlap and synergy in testing software components across loosely-coupled communities. In *Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 171–180, November 2012.

[14] T. Long, I. Yoon, A. Sussman, A. Porter, and A. 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.

[15] L. Mariani, S. Papagiannakis, and M. Pezze. Compatibility and regression testing of COTS-component-based software. In *Proceedings of the 29th International Conference on Software Engineering*, pages 85–95, 2007.

[16] A. M. Memon, I. Banerjee, N. Hashmi, and A. 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.

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

[18] W. Tichy. An interview with Prof. Andreas Zeller: Mining your way to software reliability. *Ubiquity*, 2010.

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

[20] T. Xie and J. Pei. MAPO: Mining API usages from open source repositories. In *Proceedings of the 3rd International Workshop on Mining Software Repositories*, pages 54–57, May 2006.

[21] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Effective and scalable software compatibility testing. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 63–74, 2008.