Coordinated Collaborative Testing of Shared Software Components

Teng Long*, Ilchul Yoon†, Adam Porter*, Atif Memon* and Alan Sussman*

*Department of Computer Science, University of Maryland, College Park, USA
†Department of Computer Science, State University of New York, Incheon, South Korea
{tlong,atif,aporter,als}@cs.umd.edu, icyoon@sunykorea.ac.kr

***Abstract*—Software developers commonly build their software systems by reusing other components developed and maintained by third-party developer groups. As the components evolve over time, new end-user machine configurations that contain new component versions will be added continuously for the potential user base. Therefore developers must test whether their components function correctly in the new configurations to ensure the quality of the overall systems. This would be achievable if developers could provision the configurations in house and conduct regression testing over the configurations. However, this is often very time-consuming and also there can be redundancy in test effort between developers when a common set of components is reused for providing the functionality of the systems.**

In this paper, we present a coordinated collaborative regression testing process for multiple developer groups. It involves a scheduling method for distributing test effort across the groups at component updates, with the objectives of reducing test redundancy between the groups and also shortening the time window in which compatibility faults are exposed to user community. The process is implemented on *Conch*, a collaborative test data repository and services we developed in our previous work. *Conch* has been modified to function as the test process coordinator, as well as the shared repository of test data. Our experiments over the 1.5-year evolution history of eleven components in the Ubuntu developer community show that developers can quickly discover compatibility faults by applying the coordinated process. Moreover, total testing time is comparable to the scenario where the developers conduct regression testing only at updates of their own components.

I. Introduction

Software developers commonly build their software systems by reusing components developed and maintained by third-party developer groups. 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.

At each new component version release, developers of other components (including top-level systems) must ensure that their components build and function correctly over the new configurations, potentially discovering cross-component compatibility faults, because any unidentified compatibility faults that make their way out into user community could make end-users waste time deploying the component onto their desired machine configurations. The importance of quickly discovering cross-component compatibility faults has also been emphasized by other researchers. For example, Artho et al. [3], [4] claimed that determining cross-component conflicts is important to assure the quality of package-based distributions, and categorized various cross-component conflicts found in the bug tracking systems of the Debian and Red Hat Linux distributions.

The sheer number of feasible configurations and the lack of systematic and automated cross-component compatibility testing with adequate coverage are the major reasons that cross-component compatibility bugs are not captured during the in-house testing process, but instead by frustrated end-users after version releases. In our previous work [5], [6], we showed that developers can discover cross-component compatibility faults and shorten the time during which the faults are exposed to the user community by performing compatibility testing over a set of carefully selected configurations, continuously as components evolve. However, it is still very costly for an individual developer group to conduct testing over all feasible configurations, because as previously noted the number of configurations will continue to increase rapidly as components contained in the configurations evolve. In addition, performing compatibility testing individually and in isolation by each developer group can reduce the value of testing by wasting resources unnecessarily. We have observed in earlier work [7], [8] that significant amounts of testing effort are redundant across developer groups, when there are dependency relationships between components developed by the groups and also when a set of identical components are reused for developing the components – i.e., when the components are *shared* by the groups. A large amount of test effort could be saved if the groups collaborate by sharing both test data via a shared repository, and cross-component compatibility faults can then be more quickly discovered.

However, the collaborative test process in our previous work was designed to work in an ad-hoc, greedy fashion without careful coordination between developer groups. At every component update, multiple groups may start to conduct

regression testing 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 want to minimize the time in which potential compatibility faults are exposed to user community. But, on the other hand, 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 paper, we 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. We apply this process to a set of software systems with shared components in an Ubuntu distribution, emulate the application of the process over the 1.5-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. For example, our process was able to discover a compatibility fault related to the OpenSSL library earlier than it was discovered by the developers, and the fault had been classified as critical by the community. We also found configuration-specific issues heavily discussed in online forums, but were not well documented by the developers.

The testing process is realized in an automated tool by extending *Conch*, a set of data sharing services we previously developed [7], [9]. Conch originally provided a repository that stored various test artifacts, including functional test results of components under different configurations, virtual machine images that realize configurations, and line/branch code coverage information obtained by running component test suites. Testers pulled data from *Conch* and reused the data to improve local testing. In this work, *Conch* has been modified to perform the role of the test coordinator for the new test process. When an updated component version is released, Conch analyzes the code coverage information for all components that rely on the updated component and pushes notifications to the developers of the components affected by the update. Conch also informs the developers when to locally start regression testing, so that they can better leverage test data created by others developer groups. With *Conch*, testers performing regression testing continuously but isolated from other testers can test new configurations more promptly without introducing redundant effort across testers.

This paper describes several contributions, including:

1) a coordinated collaborative testing process for minimizing both fault exposure time and test redundancy across developer groups;
2) an automated tool to execute the coordinated collaborative testing process; and
3) empirical evaluation of the testing process over a set of real-world components in a large software community.

The rest of this paper is organized as follows. Section II provides background and context from our earlier work. In Section III, we describe the testing process designed to coordinate the collaborative testing process across multiple developer groups, and present experimental results in Section IV. Related work is described in Section V and we conclude in Section VI.

II. Background

In this section, we first describe our prior work on modeling component-based software systems and their configurations, and then we describe overlaps and synergies that can be obtained through collaborative testing across multiple developer groups. After that, we outline the shared test data repository we have previously developed for enabling collaborative testing. At last, we introduce *regression cofigurations*, which are the new configurations that components need to be tested on when their provider components are updated.

A. Modeling Component-based Systems

We represent component-based software systems with an *Annotated Component Dependency Model(ACDM)* [5]. An ACDM contains two parts: (1) a directed acyclic graph called the *Component Dependency Graph (CDG)* and a set of *Annotations*. In the example model in Figure 1, each node in the CDG represents a unique component, and inter-component dependencies are specified by connecting nodes with **AND(*)** or **XOR(+)** relationships. Component `A` *depends* on component `D` and either one of `B` or `C`. In this model, a dependency means that one component requires another component at build-time, runtime, or both. We call a component that depends on others a **user component**, and a component on which other components depend a **provider component**. *Annotations* include version identifiers for components, and constraints between different components and component versions, written in first-order logic.

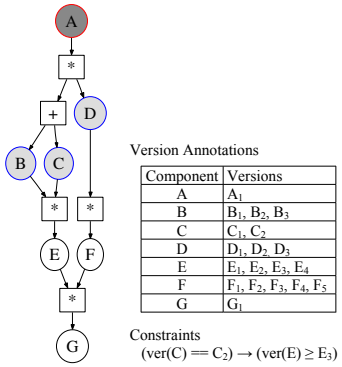

Fig. 1. An Example System Model

The example above is for a single software system (the component `A`). When multiple systems share a set of components, we can create an integrated CDG with overlapping subgraphs. For example, in Figure 2, `Serf, Flood, Subversion`

and Managelogs depend on the APR component. Each developer group of a top-level component tests configurations where each contains a specific set of component versions, including the ones in the shared sub-graph rooted at the APR node. That is, the components contained in each configuration are built and the behavior of the components are tested before building and testing the top-level component, which is the software system developed by the group.

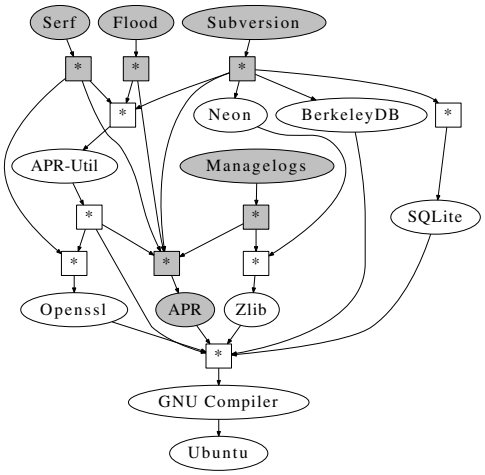

Fig. 2. Systems with Common Components

B. Synergies from Collaborative Testing

When multiple systems share a set of components, efforts to test partial configurations can be performed by multiple developer groups, introducing redundancy. For example, in Example 2, all top-level component developer groups have to build and test configurations that contain the components in the shared sub-graph rooted at the APR node. In [8], we studied the overlaps and synergies achievable by sharing test artifacts (e.g., test results, logs, and configurations realized in virtual machine images) across developer groups.

More specifically, we measured the line/branch coverage of a shared component and the spectrum of parameter values used to invoke methods of the component, when the component is executed by running its test suite or by executing the test suites of its user components [8]. The results showed that the test cases designed and run by component users can be individually less comprehensive than those by shared component developers, but in some cases can exhibit new behaviors not covered by the original provider's test cases.

In a followup study [7], we collected the 1-year update history of 5 user components in a Debian Linux distribution and also the components shared by the user components, and then replayed the regression testing of the user components. At every component update, we generated configurations that had to be newly tested or retested, and estimated the time required for testing the configurations by the user component developer groups. The experiment results showed that a large amount of build and functional test efforts are redundant across the groups, because all user component developers had to prepare

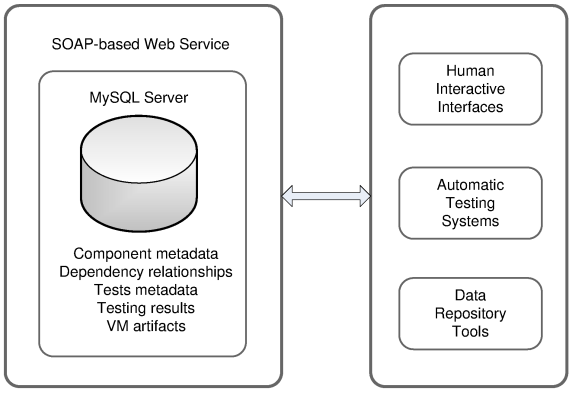

Fig. 3. The Conch Data Sharing Repository

(in virtual machine formats) identical working configurations on which their components are deployed. Roughly, 11% percent of the preparation time could be saved by sharing test results, and up to 77% of the time could be saved by sharing partially prepared configurations realized in virtual machine images.

C. Repository for Sharing Test Artifacts

To facilitate sharing test artifacts among developers, we have designed and implemented a Web-service based data repository called *Conch* [7]. The structure of *Conch* is illustrated in Figure 3. The repository provides Web services that can handle requests for accessing previously executed test results via WSDL [10] and the SOAP [11] protocol. Developers can write tools or plug-ins that enable their automated test systems to access the data stored in the repository.

Conch shares (1) component dependency information, (2) component build and functional test results, (3) pre-built configurations realized in virtual machine images, and (4) code coverage of all provider components when they are executed by running the test suite of each user component. Developers can use the information to analyze the access patterns of user components, or to preclude configurations that cannot be used, due to one or more build or functional test failures of components contained in the configurations. They can also reuse pre-built configurations to save configuration preparation time. Since the size of full virtual machine images is very large, we developed a tool called *Environment Differencing Engine(Ede)* [9]. Ede supports efficient system environment differencing for creating incremental representations of system state, and also supports restoring a system state from an incremental representation.

D. 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 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 4 shows an example update history of the components in Figure 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.

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:

$A_1, B_2, D_1, E_1, F_1, G_1 \rightarrow B_2$
$A_1, C_2, D_1, E_1, F_1, G_1 \rightarrow C_2$
$A_1, B_2, D_2, E_1, F_1, G_1 \rightarrow D_2$
$A_1, C_2, D_2, E_1, F_1, G_1 \rightarrow D_2$
$A_1, B_2, D_2, E_2, F_1, G_1 \rightarrow E_2$
$A_1, C_2, D_2, E_2, F_1, G_1 \rightarrow E_2$
⋮

Fig. 4. Example Regression Configurations of A

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 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 paper it is referred to as *regression testing*. In the next section, we present a method to better coordinate regression testing activities performed by multiple developer groups when a common set of components is shared by the groups.

III. Coordinated Collaborative Testing Process

This section introduces a coordinated collaborative regression testing process whose objectives are to minimize the time during which compatibility faults introduced by component updates are exposed to the user community, as well as to minimize redundant test effort between developer groups. 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, historical credibility and performance of the developer groups.

A. 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 our previous work 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 notify 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.

B. 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 redbenchmark test suite using the testing resources of the group and the time required to run the benchmarks on the testing resources at the *Conch* repository, as shown in Equation 1. The benchmark

test suite is a representative test suite selected by the *Conch* administrator, used solely to measure relative execution times to run the test suite on different resources.

$$PF(G) = \frac{T_G}{T_{Conch}} \quad (1)$$

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.

$$TFR(G) = \frac{FC_G}{TC_G} \quad (2)$$

In Equation 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:

$$EPF(G) = \frac{PF(G)}{(1 - TFR(G))} \quad (3)$$

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 1, will then be requested to build and test the updated component over the partial configuration set.

Algorithm 1 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. Developer groups of other direct and indirect user components will defer their local testing until the test result for C is shared via *Conch*.

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

Algorithm 1: 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
```

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 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.

C. 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 our previous work [7] 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 the current work, 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. The prior mapping enables more elimination of unnecessary testing effort, however, it also generates much more overhead with more frequent coverage collection. For this reason, we adopt coverage-based selection on test suite level for our implementation.

IV. Experiments

In this section we evaluate the effectiveness and performance of the coordinated collaborative testing process presented in Section III. 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.

A. 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 5. 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, BerkeleyDB), system utilities (Bzip2, zlib), and a GUI application (XBMC). Dependency relationships between the subject components were manually determined and entered into *Conch*. In this work, inter-component dependencies are static. We considered three Ubuntu releases for the experiments. Table I 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 [12] or Subversion [13]. Figure 6 shows the 87 total component versions released during the test period, ordered by day from the start of the test period.

B. Testing Strategies

As previously discussed, 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 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

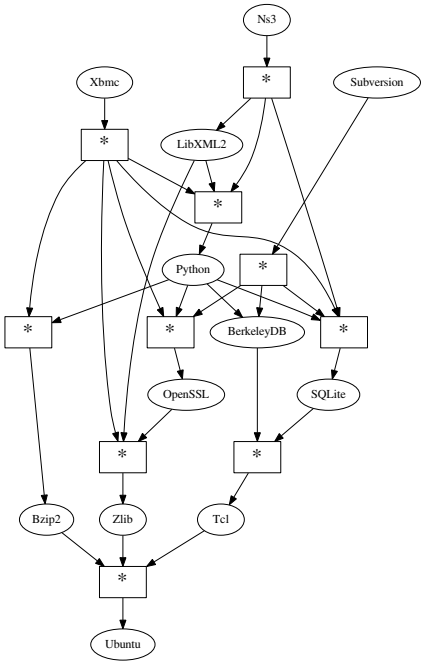

Fig. 5. Subject Components for Continuous Collaborative Testing

TABLE I
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

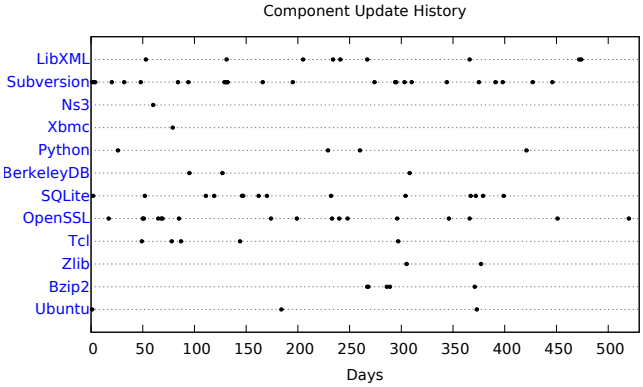

Fig. 6. Subject Components Update History

we apply this strategy to the component update history in Figure 4, 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 4, 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 Section III-A, 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 Section III-C, in addition to the second variant. During the testing process, whenever a user component test suite covers any of its provider components' code, we update the code coverage mapping in *Conch*.

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 Section III-B.

C. 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 [14]. 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, we first performed **eager** testing for the top-level components in Figure 5. That is, we did all the test activities that must be performed by the 12 developer groups, and measured 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 II). 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 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

[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.

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

adjusted after each test execution. The initial failure rate is set to 0.1 for all developer groups.[4]

D. Experimental Results

Given the CDG in Figure 5 and the update history in Figure 6, 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?

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 II 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 II, we find that **naive** testing has a very short cumulative time. This is because it covers the smallest number of 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% that 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 (*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 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.

[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.

TABLE II
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

2) Comparing Maximum Fault Exposure Time: The cumulative test execution time exposes the costs of redundancy in joint 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 time **maximum fault exposure time**, which is the maximum time until every compatibility fault introduced by a provider component update is discovered, assuming 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 7, 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).

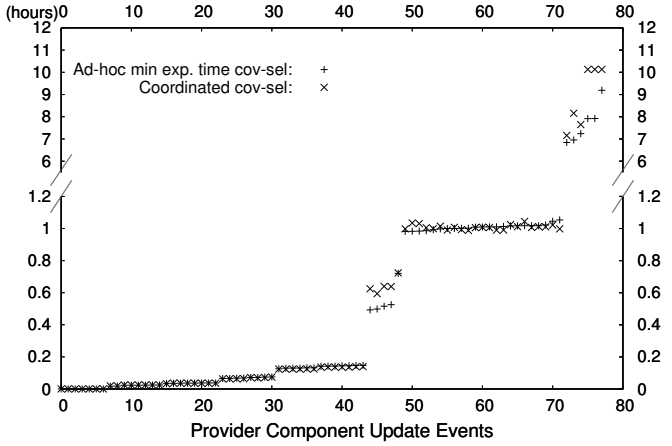

Fig. 7. Maximum fault exposure time obtained by running the regression testing process with the strategies *Ad-hoc min exp. time, cov-sel* and *Coordinated, cov-sel.*

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 Section III-B. 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 7 and Table II 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.

3) Analyzing Cross-Component Compatibility Faults: In order to determine whether the coordinated collaborative testing process can be effective in revealing cross-component compatibility faults (RQ2), we describe example faults that could have been discovered if coordinated collaborative testing had been performed as part of a continuous integration practice, which is a core practice in agile software development [15]. We classify the faults captured in the subject components in our experiments into three categories, and discuss further in the following paragraphs. Overall, the investigation of the captured faults shows that *Conch* can be useful for revealing various compatibility faults between software components with dependencies among them.

First, coordinated collaborated 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 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 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, several test cases in the "test_ssl" user component were able to 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.

E. Threats to Validity

As an empirical study, our study has potential threats to the validity that practitioners have to consider when interpreting the presented results and conclusions. The primary threats to validity in our work are external, including the selection of subject components, their test suites, and platforms run upon. We used the subject components from the Ubuntu distribution, and the components do not necessarilty represent features of components in other fields such as scientific or business domains. Also, components are tested using their test suites distributed together with their source code. A few subject components contained a small number of test cases in the test suites of several components. Another threat is the probablistic models we used to model the performance and the reliability of developer groups. It is possible that developers participating in the coordinated collaborative testing show different distributions.

A threat to internal validity is that components are built and their test suites are executed based on our understanding of the components through manuals or other available resources. A component can show different performance depending on the way it is configured. For example, different compilers or optimization levels can be used for building components.

V. Related Work

Distributed software development has become very common, and many researchers have started investigating and evaluating such development processes. Ebert et al. studied the advantages and challenges of globally distributed development activities in [16]. Ramesh et al. [17] discovered in their study that agile software development methods like extreme programming and distributed development can be blended to

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

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

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

reap the benefits of both. In these studies, group collaboration and development coordination are considered key factors to success. Although the two studies are both focusing on collaboration within a single organization, similar challenges are faced to support continuous integration and efficient coordination across multiple groups and/or organizations.

To support distributed software development, researchers have emphasized the importance of tools for collaboration among distributed teams [18], [19]. Bird et al. [19] reported that globally distributed software development within a single company may not perform worse (in terms of failures) than centralized development. In [18], 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 targeted at automatic testing systems for supporting continuous integration.

There has also been work on methods and tools to support continuous integration when multiple teams collaborate on large software projects [20], [21]. Elbaum et al. [20] designed algorithms to pre-select and prioritize test cases from test suites to make continuous integration processes more cost-efficient. Nilsson et al. [21] developed a 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 only complementary support for decision making. The tool itself does not support automatic test scheduling.

An important part of our work is the tools and infrastructure we provide to support coordinated collaborative testing, as part of the continuous integration process. There are many tools used to support continuous integration. Jenkins [22] 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 used with our coordinated scheduling process. Autopkgtest [23] 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 a 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. .

VI. Conclusions

As reuse of third-party components has become a common theme in today's software development processes and component developers release new versions frequently, it is important to discover cross-component compatibility faults early in the development process to ensure the quality of overall systems as well as the shared components.

In this paper, 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.

In the future, we plan to apply coordinated testing to a set of components in the high-performance computing domain, and also to improve the test scheduling algorithm to utilize end-user computing resources in addition to the resources at developer sites, to support more comprehensive testing and improve overall software quality.

Acknowledgments

This work was partially supported by the US National Science Foundation (CCF-0811284, CNS-1205 501, CNS-0855055), the National Research Foundation of Korea (NRF-2013010695), and the MSIP of Korea (IITP-2015-R0346-15-1007).

References

[1] F. Erich, C. Amrit, and M. Daneva, "A mapping study on cooperation between information system development and operations," in *Product-Focused Software Process Improvement*, ser. Lecture Notes in Computer Science, A. Jedlitschka, P. Kuvaja, M. Kuhrmann, T. Mnnist, J. Mnch, and M. Raatikainen, Eds. Springer International Publishing, 2014, vol. 8892, pp. 277–280. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-13835-0_21

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

[3] C. Artho, K. Suzaki, R. di Cosmo, R. Treinen, and S. Zacchiroli, "Why do software packages conflict?" in *Proceedings of the 9th Working Conference on Mining Software Repositories (MSR 2012)*, Zurich, Switzerland, 2012, pp. 141–150.

[4] C. Artho, K. Suzaki, R. di Cosmo, and S. Zacchiroli, "Sources of inter-package conflicts in debian," in *Proceedings of the Workshop on Logics for Component Configuration (LoCoCo 2011)*, Perugia, Italy, 2011, short presentation.

[5] I. Yoon, A. Sussman, A. Memon, and A. Porter, "Effective and scalable software compatibility testing," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA 2008)*, New York, NY, USA, 2008, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390640

[6] ——, "Towards incremental component compatibility testing," in *Proceedings of 14th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE-2011)*, 2011, pp. 119–128.

[7] T. Long, I. Yoon, A. Memon, A. Porter, and A. Sussman, "Enabling collaborative testing across shared software components," in *Proceedings of the 17th International ACM SIGSOFT Symposium on Component-based Software Engineering*, ser. CBSE '14. New York, NY, USA: ACM, 2014, pp. 55–64. [Online]. Available: http://doi.acm.org/10.1145/2602458.2602468

[8] 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 2012)*. Dallas, TX, USA: IEEE Computer Society, 2012.

[9] 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 2014 IEEE 28th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, ser. IPDPSW '14. Phoenix, Arizona, USA: IEEE Computer Society, 2014.

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

[11] "SOAP Version 1.2," *http://www.w3.org/TR/soap12-part1/*, 2007.

[12] "Bazaar Version Control System," *http://bazaar.canonical.com/en/*, 2015.

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

[14] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "Openstack: toward an open-source solution for cloud computing," *International Journal of Computer Applications*, vol. 55, no. 3, pp. 38–42, 2012.

[15] S. Stolberg, "Enabling agile testing through continuous integration," in *Proceedings of the 2009 Agile Conference*, Aug 2009, pp. 369–374.

[16] C. Ebert and P. De Neve, "Surviving global software development," *IEEE Software*, vol. 18, no. 2, pp. 62–69, Mar. 2001. [Online]. Available: http://dx.doi.org/10.1109/52.914748

[17] B. Ramesh, L. Cao, K. Mohan, and P. Xu, "Can distributed software development be agile?" *Communications of the ACM*, vol. 49, no. 10, pp. 41–46, Oct. 2006. [Online]. Available: http://doi.acm.org/10.1145/1164394.1164418

[18] A. Begel and T. Zimmermann, "Keeping up with your friends: Function foo, library bar.dll, and work item 24," in *Proceedings of the First Workshop on Web2.0 for Software Engineering*, May 2010.

[19] 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)*, 2009, pp. 518–528. [Online]. Available: dx.doi.org/10.1109/ICSE.2009.5070550

[20] S. Elbaum, G. Rothermel, and J. 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*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 235–245. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635910

[21] A. Nilsson, J. Bosch, and C. Berger, "Visualizing testing activities to support continuous integration: A multiple case study," in *Agile Processes in Software Engineering and Extreme Programming*, ser. Lecture Notes in Business Information Processing, G. Cantone and M. Marchesi, Eds. Springer International Publishing, 2014, vol. 179, pp. 171–186. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-06862-6_12

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

[23] "Automatic testing of Debian-format packages," *http://launchpad.net/autopkgtest*, 2015.