Overlap and Synergy in Testing Software Components Across Loosely-Coupled Communities

Teng Long, Ilchul Yoon, Adam Porter, Alan Sussman and Atif Memon
UMIACS and Department of Computer Science, University of Maryland
{tlong, iyoon, aporter, als, atif}@cs.umd.edu

ABSTRACT

Component integration rather than from-scratch programming increasingly defines software development. As a result software developers often playing diverse roles including *component provider* – packaging a component for others to use, *component user* – integrating other providers' components into their software, and *component tester* – ensuring that other providers' components work as part of an integrated system. In this paper, we explore the conjecture that we can better utilize testing resources by focusing not just on individual component-based systems, but on groups of systems that form what we refer to as *loosely-coupled software development communities*, meaning a set of independently-managed systems that use many of the same components. Using components from two different open source development communities that share a variety of common infrastructure components, such as compilers, math libraries, data management packages, communication frameworks and simulation models, we demonstrate that such communities do in fact exist, and that there are significant overlaps and synergies in their testing efforts.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*component-based software, test coverage, test automation*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*software portability, software configurations*

General Terms

component dependency graphs

Keywords

component-based software, software integration

1. INTRODUCTION

Component integration rather than start-from-scratch programming increasingly defines software development. As a result, developers are spending increasing amounts of time and effort assembling, coordinating and verifying the interactions of ever-larger numbers of independently provided components over which developers have limited control. To address this trend, numerous research groups are studying and developing methods, tools and techniques to support large-scale, component-based software engineering [4, 14, 15, 16]. The results of these efforts are appearing in specialized symposia and workshops(See [1] and [2]).

In this paper, we expand on this existing research by focusing not just on individual component-based systems, but on groups of systems that form what we refer to as *loosely-coupled software development communities* – i.e., independently managed software systems that use many of the same components. We conjecture that such communities are quite common, forming naturally where functional requirements, hardware and/or industry standards lead software developers to draw on common infrastructure components.

Today such loosely-coupled software development groups work largely in isolation from each other. We believe however that there are significant overlaps in the work they do to customize, integrate, and test the components they use. We also believe that these groups could act in synergistic ways to share effort and knowledge in ways that improve components common among the groups.

As a first step in exploring our conjecture that there exist communities that share common components and that significant work could be saved by sharing test effort across these groups, we conduct an initial study focused on two communities. The first community includes *APR* (the Apache Portable Runtime), a system-independent runtime library, and four components (*flood, managelogs, serf* and *subversion*) that utilize APR for network communication purposes. The second contains *MPICH2*, an implementation of the Message Passing Interface standard [9, 22] for high performance computing, and five components that depend on it (*FreePooma, PETSc, ParMETIS, SLEPc* and *TAO*), all of which can be built on MPICH2 to use its message passing capabilities. We use the project-provided test suites for these components to show that there are overlaps (1) in how the test cases for these components exercise the full component assemblies, (2) in the code these test cases cover, and (3) in the bugs they detect. We leverage our previous work on component dependency graphs (CDGs) [27, 28] that gives us a formal foundation for studying component-based systems and communities. In particular, it enables us to represent component provider-user relationships, and to produce *operators* for computing overlaps and synergies in how components get exercised and used.

Through a set of studies on the subject components and their test cases, we obtain the following preliminary findings:

1. Often the *users* of a component build and test it on a broader set of configurations than the component's provider does;
2. When a base component is used by another component, testing the new component induces significant code coverage in the base component;
3. When a base component is used by another component, testing the new component can invoke functions in the base component with a broader set of parameter values than were tested by the base component's own tests;
4. When a base component is used by another component, faults in the base component are often discoverable by testing the new system alone;
5. Even if a new system indirectly uses a component (i.e. through directly using yet another component), faults in the base component are still likely to be discovered via testing the other component alone;

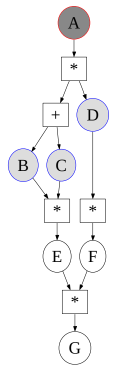
Our work makes several initial contributions including:

1. Modeling of loosely-coupled communities based on component dependency graphs (CDGs);
2. Formal definitions of criteria to quantify overlaps and synergies in testing software components assemblies;
3. Careful examination, testing and evaluation of multiple components from two different communities, as well as analyses of their testing overlap and synergies, using several criteria that we propose.

The rest of the paper is organized as follows. We introduce our previous work on modeling of component-based systems and component dependency graphs, and extend the dependency relationship to model component assemblies in Section 2. In Section 3 we formally define some terms to quantitively model how components are exercised by others in a component assembly. Section 4 describes objectives, metrics and experimental settings of our empirical study, Section 5 presents and analyzes results obtained during the empirical study, Section 6 surveys related work and in Section 7 we conclude with a brief discussion and future work.

2. MODELING COMPONENT ASSEMBLIES

We model the component provider/user/tester relationships using a formal representation of component assemblies that we developed in prior work. Our formal representation has two parts: a directed acyclic graph called the *Component Dependency Graph (CDG)* and a set of *Annotations*. As illustrated in the example in Figure 1, a CDG specifies inter-component dependencies by connecting components with AND and XOR relationships. For example, component `A` in the example CDG depends on component `D` and exactly one of either `B` or `C` (captured via an `XOR` node represented by `+`). *Annotations* include version identifiers for components, and constraints between components, written in first-order logic.



Version Annotations

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

Constraints
$(\mathrm{ver}(C) == C_2) \rightarrow (\mathrm{ver}(E) \geq E_3)$

Figure 1: An Example System Model

We have developed several testing techniques to *cover* this model [27, 28, 26]. One specific coverage criterion, relevant to our work here, is based on the observation that the ability to successfully test a component c is strongly influenced by the components on which c *directly depends*. In CDG terms, a component c directly depends on a set of components, DD, such that for every component, $DD_i \in DD$, there exists a path, not containing any other component node, from the node encoding c to the node encoding DD_i. From this definition, we can obtain relationships, called *DD-instances*, between versions of each component and versions of all other components on which it directly depends. We call this criterion *DD-coverage*.

We now extend the dependency relationships in the CDG and DD-coverage for our current work, which includes both build testing and functional testing. We say that component a *directly uses* component b if a directly depends (as defined above) on b. Component a *indirectly uses* component f if there exists at least one component b such that a directly uses b, and b either directly or indirectly uses f. The number of components on the shortest directed path between two components is defined as the *distance* between those two components. Finally, a component *provider* makes a component available for others to use. In Figure 1 each of the components B, C, D, E, F, and G are provided for use; component A directly uses B, C, and D; component D directly uses F, which directly uses G; both B and C directly use E, which directly uses G; components B, C, and D indirectly use G. And A indirectly use E, F, and G.

3. MODELING HOW COMPONENTS ARE EXERCISED

Starting with the definitions in Section 2, we now develop several formal terms to quantitatively study how components get exercised by other components in a component assembly.

Definition *Induced Coverage*: Assume component a directly or indirectly uses component b, and a has a test suite T_a. In a system where a is successfully built on b, when running a's test suite, T_a, the fraction of b's coverage elements (lines, branches, function, parameter value, faults, etc.) that get covered is called ***b*'s induced coverage from *a***, represented as C_b^a.

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

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

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

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

$C_E^B \cup C_E^C$, $C_E^B \cap C_E^C$ and $C_E^B - C_E^C$ are also demonstrated in Figure 2.

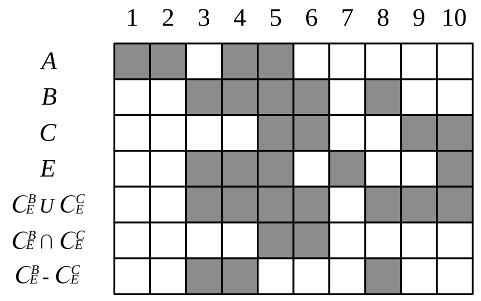


Figure 2: Induced Coverage Example

4. EMPIRICAL STUDY

Our vision of community-based testing is based in large part on our conjecture that there is actionable structure to the testing efforts of community members. For example, we believe that there are significant overlaps in the way components shared by multiple users are tested. If true, then in theory such duplicate work could be avoided by exchanging of testing results with no loss of testing effectiveness. We also believe that different component users test shared components in unique ways, so the aggregate testing of the entire community is often broader than that done by individual component providers.

We now conduct an initial empirical study, attempting to formalize and quantify some of these issues in the context of a few real communities that share a number of components. We selected these specific communities because each component in the community has its own build and functional tests. Our analyses involve executing the test cases and studying how various execution metrics overlaps across community members, and also show that some community members' efforts are not duplicated, so they can provide added synergies of test value.

4.1 Research Questions

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

RQ1: Overlap: To what extent do community members duplicate test effort?

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

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

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

4.2 Metrics

For *build testing*, we compare the set of configurations, an ensemble of components and their versions, on which a component provider tested a component, to the set of configurations on which component users tested the same component. More specifically, for a component C_j, we define $\psi(C_i, C_j)$ as the set of configurations of C_j build tested by the developer/tester of component C_i. Note that $\psi(A, A)$ is valid – it represents the set of configurations on which component A is build tested by the developer of A. Having conveniently defined ψ to return a set, we can now use the usual set intersection operation $\cap$ to study the overlaps in build testing; and set union $\cup$ to study the synergies in build testing by multiple testers.

For *functional testing*, we use several criteria to measure the overlaps and synergies among functional test cases. In particular, we use code (line and branch) coverage, parameter value coverage, and faults detected. Without loss of generality, we use a matrix representation for code coverage and faults detected. For parameter value coverage, we record all values observed for each numeric parameter.

More formally, given a test suite $TS(C_i)$ for a component C_i that invokes a set of functions $\mathcal{F}$ of component C_j, we record the following artifacts:

- A *code coverage matrix* that, for each test case in $TS(C_i)$, records the number of times a coverage element (line and branch) in C_j was covered.
- *Parameter values*, a list of values, one element for each numeric parameter of a function $f \in \mathcal{F}$.
- A *fault matrix* that records whether each test case in $TS(C_i)$ passed or failed, and the fault detected.

Given these artifacts, we compute several metrics: (1) induced code coverage for line and branch from testing a base component and its users, (2) ranges of parameter values passed to functions of a base component when running the

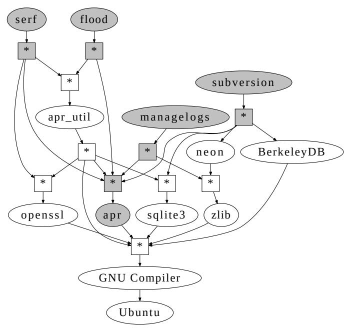


Figure 3: APR CDG

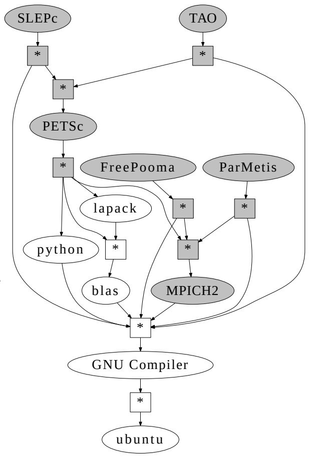


Figure 4: MPICH2 CDG

test cases of its user components, and (3) number of faults detected in a base component by running the test cases of its user components.

4.3 Subject Components

We chose to study two widely-used open source software components: APR and MPICH2. APR[1] (Apache Portable Runtime) is widely used in the web services community, for instance, by components such as the Apache HTTP server and the Subversion version control system. MPICH2[2], from the high performance computing (HPC) community, is an implementation of the Message Passing Interface(MPI) standard that is used to implement scientific applications on many high performance and parallel computing platforms.

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

SLEPc, a library for solving large-scale sparse eigenvalue problems on parallel computers, and *TAO*, a library for large-scale optimization problems, are user components of the PETSc component, and therefore they indirectly use MPICH2. Figure 3 and 4 show the CDGs for APR and MPICH2, respectively. In the CDGs, we highlighted the components we focus on in this study.

4.4 Experimental Operation

APR and MPICH2 provide their own test cases and for this study we execute the test cases for each component, measuring how the test cases happened to cover APR and MPICH2 using each of the metrics described in Section 4.2. Our goal was to better understand 1) how the testing of a given higher-level component in a CDG induces coverage in lower-level components it uses, and 2) how that induced coverage compares to that obtained when the lower-level component was tested with its own test cases.

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

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

5. DATA AND ANALYSIS

Based on our model of component assemblies defined in Section 2 and the metrics in Section 4.2, we analyzed data obtained from component development documentation and test artifacts from our empirical study to understand the overlap and synergies of shared test effort in loosely-coupled communities. In Section 5.1 we first identify configurations on which subject components were build-tested by component providers and users, then we discuss the possibility of broadening the set of tested configurations and saving test effort by sharing build test results in communities. In Section 5.2 we analyze the coverage information collected by running functional tests of subject components and also user components of the subject components.

5.1 Build Testing

For each component C_j in the CDGs in Figures 3 and 4, we first investigated $\psi(C_i, C_j)$ – i.e., we examined configurations *build-tested* by component providers and also configurations tested by component users. This was accomplished by carefully inspecting documents provided by component providers (e.g., HTML documents, Wiki pages, user manuals, installation guides). In some cases, component providers do not clearly specify configurations on which their components build successfully. (e.g., component providers can simply list prerequisite components.) When we do not have sufficient information to determine working configurations, we examined files such as *Makefile* to find relevant information. Table 1 lists rough information on configurations on which subject components can be built successfully.

From Table 1 we saw that none of the components are regularly build-tested on more than a handful of configurations. Although build tests for several components are performed systematically using automatic build tools such as *buildbot* or scripts for dedicated machines (e.g., *subver-*

[1]http://apr.apache.org
[2]http://www.mcs.anl.gov/research/projects/mpich2

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

Table 1: Configurations tested by developers

sion, MPICH2 and PETSc), the configurations are mostly focused on recent versions of operating systems and the required components.[3] One reason is that developers usually choose to focus their limited resources on testing their components with recent versions of required components, under the belief that users' configurations must be updated to use recent versions of require components. However, this is not necessarily true.

We also observed that successful component builds were often tested on a wider set of configurations by component users than by component providers. For example, *subversion* is build-tested on top of virtual machines hosting different operating systems, as listed in Table 1. Since *subversion* requires APR, developers have to first build APR successfully for the configurations, and some of those configurations are not explicitly tested by the APR developers. For example, build test results from the *subversion* developers can be used to increase the set of configurations for which building APR is known to be successful.

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

5.2 Line/Branch Coverage

This analysis of functional testing examined how line and branch coverage changed depending on which component's tests were being run. Table 2 shows induced coverage of APR and MPICH2 as measured by the *lcov* tool, on two OS platforms. The "Indirect Test" columns show the union of induced coverage from *all* the test suites of their direct users, while the "Self Test" columns show coverage from just APR's or MPICH2's own test suites. "Joint" show the union of induced coverage from all components, while "Extra" shows the difference between induced coverage of APR and MPICH2 from their direct users and their own tests. Figures 5 and 6 show the actual number of lines/branches in the induced coverage of both components on two operating systems. C_A^A and C_M^M are the induced coverage from APR and MPICH2's own tests, while C_A^* and C_M^* refer to the union of coverage induced from all direct users of the base component in the subject systems.

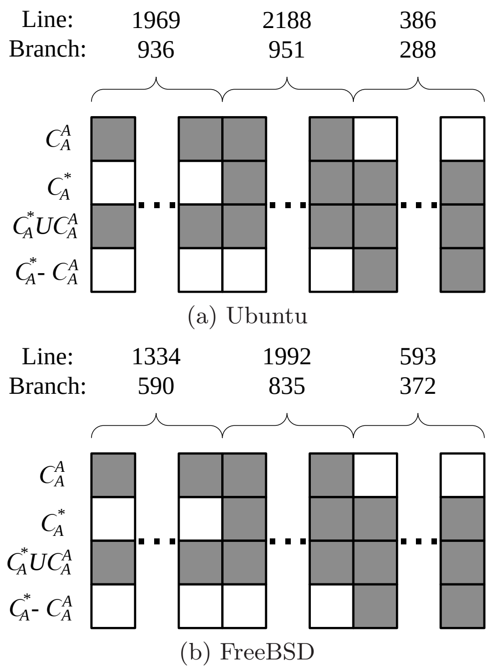


Figure 5: Induced Coverage of APR

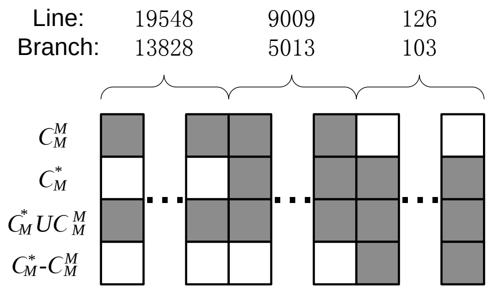
[3]Note that in the table we only show operating systems and compiler languages for the configurations and did not specify compiler vendors or versions.

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

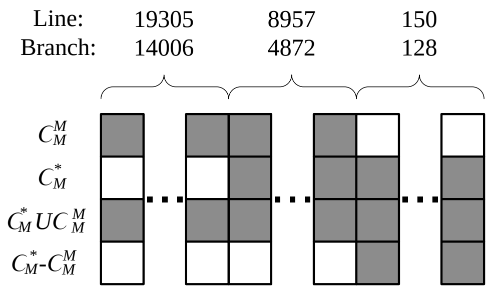
Table 2: Induced Coverage of APR and MPICH2(%)

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

Table 3: Induced Coverage Distribution of APR and MPICH2's users (%)



(a) Ubuntu



(b) FreeBSD

Figure 6: Induced Coverage of MPICH2

The results of this analysis show that the user components together achieved substantial coverage, but not as much coverage as the test suites of the base components did. However, the user components provided at least some additional coverage not achieved by the base components' tests. The extra coverage for MPICH2 is small, because MPICH2 is an implementation of a industry standard with a standard set of well-documented and widely-used API, and as a result has a very thorough test suite.

Our second analysis examined how the overall coverage achieved by testing multiple user components broke down by the component that was doing the testing, i.e., by which component's tests were being run. Table 3 shows the fraction of APR's and MPICH2's code that are covered by one, two, three or four of the upper level components, respectively, for branch and line coverage on two different platforms.

The results of this analysis suggest that for the APR example while there was some overlap in coverage from different users of APR, the tests of different users tended to cover APR's functionality in different ways. More specifically, among all the lines or branches covered by the test suites of the users of APR, about half of them were covered by the tests of only one component. For MPICH2, since it is an implementation of the MPI standard, there are a set of functions that are used by almost all MPI programs, such as MPI_init and $MPI\_finalize$. Thus we observed more overlap of coverage among user components of MPICH2, compared to the overlap between user components of APR. However, there was still around 20% of induced coverage from the users that were covered by only one user's tests, which again shows that different users have different ways of using the base component.

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

5.3 Parameter Value Coverage

APR	flood	svn	serf	managelogs		Joint
# of value range-extended parameters	13	38	11	11		62
Total # of numeric parameters	62	85	59	35		123
MPICH	PETSc	FreePooma	ParMETIS	SLEPc	TAO	Joint
# of value range-extended parameters	13	1	10	3	18	26
Total # of numeric parameters	247	55	269	140	246	302

Table 4: Parameter values passed from user component tests are sometimes outside the range of values tested by the test suites for base components.

To analyze values of individual parameters passed to functions in base components (i.e., APR and MPICH2), we instrumented all functions of APR and MPICH2 if they have at least one numeric parameter. We then ran the test suites of APR and MPICH2, and also the test suites of their users (See Figures 3 and 4). We collected the information about the parameter values passed into the instrumented functions, to see how the patterns of such values differed across the various test suites.

We observed that the test cases for user components often invoked functions in APR and MPICH2 (the base components) with values outside the range of values covered by the base component developers' test suites. In Table 4, for each base component, we show the total number of numeric parameters in the functions invoked while running test cases for the user components, and also show the number of parameters for which values tested by the user components were outside the range of values covered by the test suites of the base components. The rightmost column (Joint) in the figure shows the total number of parameters tested by at least one user component.

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

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

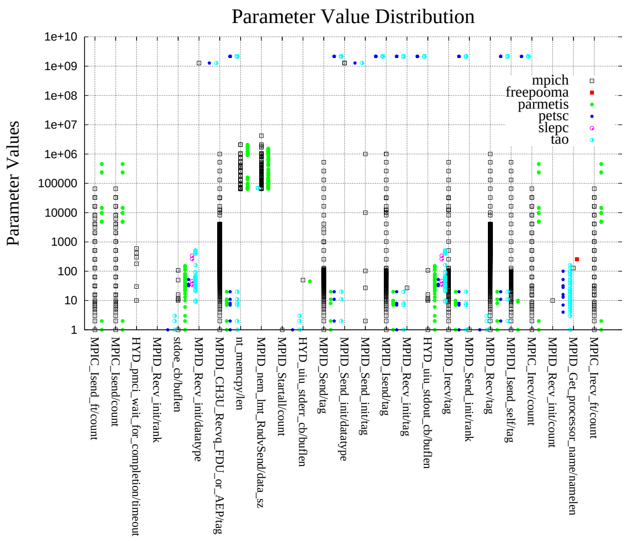


Figure 8: MPICH2 Parameter Value Distribution

Table 4 shows the number of value range-extended parameters, but that information by itself does not say that a broader set of test values is used by user components than by the base component test suite. For example, an user component may test a base component function with only one value outside the value range covered by the test suite of the base component. Therefore to look in more details at the actual parameter values covered by the base component test suites and by the user components' test suites, we show the distribution of parameter values covered by individual subject components in Figures 7 and 8.[4]

In the figures, the x-axis is pairs of function-name/parameter-name in the base components, while the y-axis is the values passed into the function from running the test suites of the base and user components. For presentation purposes, we only showed parameters for which value coverage was extended by user components. The y-axis is log-scale because of the wide range of parameter values, so we also do not show parameters values less than or equal to 0. In each graph, the parameter values covered by the base component

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

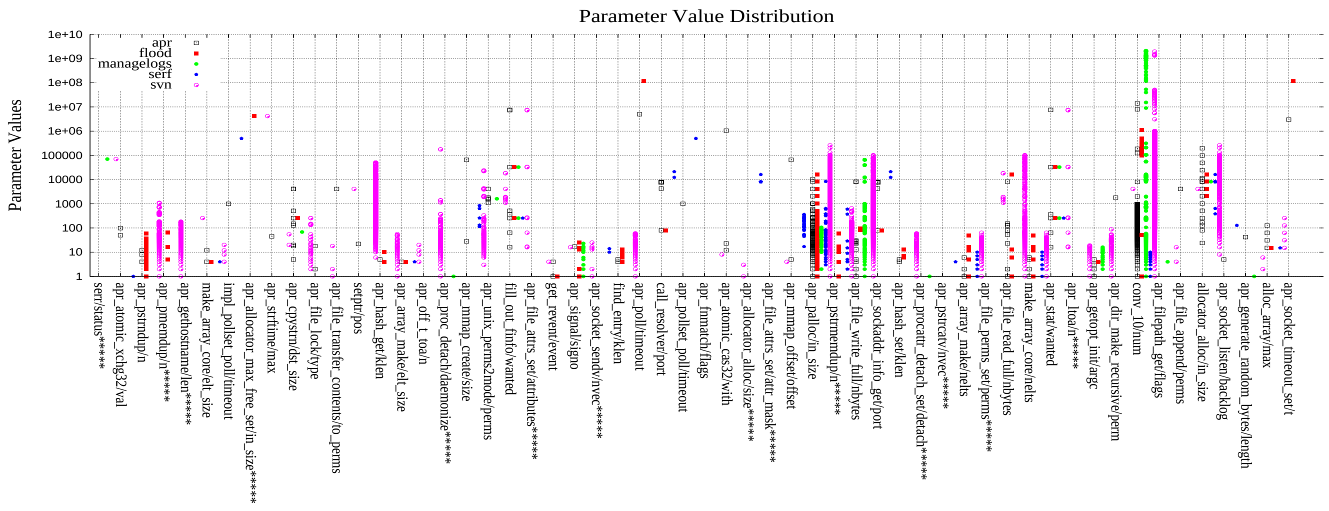


Figure 7: APR Parameter Value Distribution

are aligned to a vertical grid line, and values covered by user components are depicted on the right side of the line in the order shown in the graph legend.

In the figures, we observe that user components often test functions in the base components with a larger set of parameter values. For the APR component, *svn* tested many APR functions with diverse values, compared to the values tested by APR's test suites. For example, the APR test suite used only -1 and 5 as the values of the *klen* parameter for the function *apr_hash_get*, but the *svn* test suite used 35 different values between -1 and 56, including 0. For the MPICH2 component, we see that the MPICH2 test suites test itself uses many parameter values. This is not surprising, because, as previously noted, MPICH2 developers must do rigorous testing to ensure compliance with the MPI standard. While it is true that the number of different parameter values tested does not always increase overall test quality, sharing test results from component users with component developers can help the developers identify faults, especially if specific parameter values are associated with the faults.

We also observe that there are APR functions tested by only user components. In Figure 7, such function-parameter pairs are indicated by appending "*****" to the x-axis labels. For example, the function *apr_pmemdup* in the APR component is not tested by the APR test suite, although the function is invoked with many values by the test suites of *flood*, *serf*, and *svn*.

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

5.4 Fault Detection

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

Given that a significant number of faults must be seeded and tested individually to ensure the validity of our study, and that each round of testing for all components may be very time consuming (as long as an hour for each fault in our study), our subject system for this analysis was limited to a sub-CDG from Figure 4. The sub-CDG included MPICH2 as the base component, PETSc as a component that directly uses MPICH2, and TAO and SLEPc as components that indirectly use MPICH2 through PETSc. Since MPICH2 is a parallel computing library, two categories of faults were seeded to simulate real-world faults, which were:

1. **operator faults**: a change of an operator in the source code, including both arithmetic operators ('+', '-', '*' and '/') and comparison operators ('>', '<', '!=', '>=', '<=' and '==').

2. **constant faults**: a change of a constant value defined in macros in the source code. Non-zero constants are changed to zero, and vice versa.

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

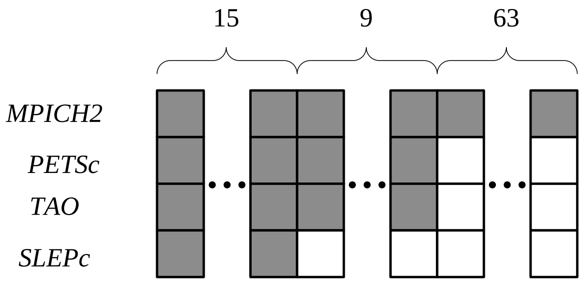


Figure 9: Test results from all components on 112 faults seeded into the MPICH2 source code.

and 0.0 for constant faults. We included all opportunities for constant faults given that their total number was very small. As a result, 96 opportunities for operator faults and all 16 opportunities for constant faults were chosen.

The testing results are shown in Figure 9. Each four boxes in a column represents the test results for the four components built on/from the source code of MPICH2 which has a single fault seeded. A filled box means the fault is detected by testing the corresponding component, while a blank box means the fault is not detected. Table 5 presents a summary of the results. Several observations can be made from the results:

1. Among all 112 faults seeded, 87 of them were detected by MPICH2's own test suite, 24 of them were detected by testing its direct user (PETSc), and 24 of them were detected by testing its indirect users (TAO & SLEPc).
2. All faults detected by direct users were also detected by indirect users.
3. All faults detected by users were also covered by the test suite of the base component.

The first observation shows that testing MPICH2's users alone can detect about 20% of all seeded faults. Remember all faults were seeded in the code that were covered by at least one component. Considering the fact that testing users alone are covering about 15.6% lines of MPICH2's source code, while the joint of all components' induced line coverage is 48.0% (refer to Table 2), The probability that a fault is seeded in the code covered by users is about 30%. Thus the result of fault detection is roughly consistent with code coverage.

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

Though no new faults are detected by users comparing to MPICH2's own tests, The third observation itself is not hard to explain. From Table 2 we can see that only 0.2% of MPICH2's code is covered exclusively by its users, while its own tests are covering 47.8% lines. Thus the probability that a fault is seeded in the code which is exclusively covered by users is about 0.004. When choosing around 100 faults uniformly among the code covered by either component, it is fully possible that no faults are seeded in such portion of MPICH2.

However, it is worth mentioning that we are only testing one of MPICH2's direct users and two indirect users for fault detection. In the real-world high performance community, far more components are using MPICH2, and we can expect more code to be covered exclusively by its users. In future work we will include more subject components and estimate the possibility of detecting new faults by users.

5.5 Threats to Validity

Like any other experiment, this empirical study has limitations that one should consider when evaluating its results and conclusions. The primary threats to validity for this study are external, involving subject and process representativeness. We chose APR, most of whose users are lightweight network applications, and MPICH2, whose users are mostly scientific computing libraries and applications. Those two categories of components do not necessarily represent the features of all loosely-coupled communities. Also, we test all components using their published default test suites, most of which are unit tests, and some of them are very limited (e.g., those for PETSc and SLEPc). More intensive testing with wider coverage is likely by the component developers.

Threats to internal validity includes factors that affect testing components without the developers' knowledge. It is the nature of component-based systems that their functionality is affected by how their base components were configured and built, as well as the compilers they used and operating systems they were built on. To limit this threat, we tried to use the same set of tools and base components, with the same versions, to build MPICH2 and APR on both Ubuntu and FreeBSD.

Last, threats to validity may exist when we chose the three metrics. Code coverage is not the only measurement of effectiveness of testing, a broader parameter value range does not necessarily lead to more thorough testing, and the results of detecting seeded faults cannot fully simulate fault coverage in real-world development.

6. RELATED WORK

Our work focuses on component-based software systems. The components are owned and developed by loosely-coupled developer teams, so overall control over the components is distributed. Testing of product line architectures has similarities to our work. Those architectures provide designs for families of related applications that share common components; these are exactly the types of applications that we are targeting [21, 11, 13]. The difference between product lines and our notion of loosely coupled self-organizing software development communities is that the component structures and architecture are imposed in a top-down manner in product lines, whereas our work evolves a bottom-up structure. Product line components are owned by one organization, so there is central control. Most often, any information repositories for these components are centralized, making their management simpler.

Many popular projects distribute regression test suites that end-users run to evaluate installation success. Well-known examples include GNU GCC [12], CPAN [5], and Mozilla [24]. Users can, but frequently do not, return the test results to project staff. Even when results are returned, however, the testing process is often undocumented and unsystematic. Developers therefore have no record of what was tested, how it was tested, or what the results were, resulting in the loss of crucial information.

Auto-build scoreboards monitor multiple sites that build/test

	Faults Seeded	Base	PETSc	TAO & SLEPc	Joint	Extra
Operator Faults	96	75	21	21	75	0
Constant Faults	16	12	3	3	3	0
All Faults	112	87	24	24	87	0

Table 5: Summary of Fault Detection Results

a software system on various hardware, operating system, and compiler platforms. The Mozilla Tinderbox [25] and ACE+TAO Virtual Scoreboard [8] are examples of auto-build scoreboards that track end-user build results across various volunteered platforms. Bugs are reported via the Bugzilla issue tracking system [23], which provides inter-bug dependency recording and integration with automated software configuration management systems, such as CVS or Subversion. While these systems help to document multiple build processes, deciding what to put under system control and how to do it is left to users.

Online crash reporting systems, such as the Netscape Quality Feedback Agent [19] and Microsoft XP Error Reporting [18], gather system state at a central location when fielded systems crash, simplifying user participation by automating parts of the problem reporting process. Orso et al. [20] developed GAMMA to collect partial runtime information from multiple fielded instances of a software system. GAMMA allows users to conduct a variety of different analyses, but is limited to tasks for which capturing low-level profiling information is appropriate. One limitation of these approaches is their limited scope, *i.e.*, they capture only a small fraction of interesting behavioral information. Moreover, they are *reactive*, meaning the reports are only generated *after* systems crash, rather than *proactive* in attempting to detect, identify, and remedy problems *before* users encounter them.

Distributed continuous QA environments are designed to support the design, implementation, and execution of remote data analysis techniques such as the ones described above. For example, Dart [17, 7] and CruiseControl [6] are continuous integration servers that initiate build and test processes whenever repository check-ins occur. Users install clients that automatically check out software from a remote repository, build it, execute the tests, and submit the results to the Dart server. A major limitation of Dart and CruiseControl, however, is that the underlying QA process is hard-wired, *i.e.*, other QA processes or other implementations of the build and test process are not easily supported.

Although these projects can provide some insight into fielded component behavior, they have significant limitations. For example, many existing approaches are reactive and have limited scope (*e.g.*, they can be used only when software crashes or focus only on a single, narrow task), whereas effective measurement and analysis support needs to be much broader and more proactive, seeking to collect and analyze important information continuously, before problems occur. Prior approaches also limit developer control over the measurement and analysis process. Although developers may be able to decide what aspects of their software to examine, some usage contexts are evaluated multiple times, whereas others are not evaluated at all.

Although we are, to our knowledge, the first to investigate the idea of loosely-coupled communities and their potential synergies, other researchers have 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 [3]. Leveraging the tools enabled by social media, individual developers can self-organize within and across organizational boundaries; software development communities may emerge centered around new technologies, common processes and target markets. Social media will also allow companies consisting of individuals to conceive of, design, develop, and deploy successful and profitable product lines.

7. CONCLUSIONS

This work is driven by our research conjecture – that in the context of loosely-coupled software development communities it may be profitable to optimize testing processes across multiple projects, rather than within individual projects as is generally done today.

To begin exploring this speculative idea, we have conducted an initial study of two open source software development communities. Overall the results of these various analyses suggest that the test cases designed and run by component users can be individually less comprehensive than those designed and run by component providers, but in some cases can exhibit new behaviors not covered by the original provider's test cases. Although preliminary, this finding is interesting and justifies experimenting with other existing testing techniques. One example of such a technique is carving [10], which could be used to extract test cases, specific to a lower-level component from the test cases designed for a higher-level component that uses the lower-level one. In addition, we have found that testing done at different levels in a CDG by different components for the same base component appear to be complementary. It remains to be seen, however, whether and how those differences might or might not dissipate as more higher-level components are added to a community. That is, as the number of higher-level components using a given lower-level component increases, the combined higher level test cases may begin to behave in the aggregate like a high quality test suite for the lower-level component. Finally, these preliminary results suggest that test results from the higher level components might provide useful feedback for understanding usage patterns or operational profiles from a component user's perspective. Component developers could use this feedback to improve their own test suites.

In our future work we will investigate the benefits of sharing test artifacts across additional software development communities, both for build testing and for various types of functional testing. We also plan to build testing tools and data repository infrastructure, and modify existing testing tools, to enable both component providers and users to benefit from all their testing efforts. The overall goal is to enable leveraging *all* the testing efforts within a community to improve the quality of their components.

8. REFERENCES

[1] *CBSE '11: Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*, New York, NY, USA, 2011. ACM. 5941101.

[2] *WCOP '11: Proceedings of the 16th international workshop on Component-oriented programming*, New York, NY, USA, 2011. ACM. 5941101.

[3] 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.

[4] A. Brown and K. Wallnau. The current state of cbse. *Software, IEEE*, 15(5):37 –46, sep/oct 1998.

[5] Comprehensive Perl Archive Network (CPAN). *http://www.cpan.org*, 2010.

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

[7] Dart: Tests, reports, and dashboards. *http://public.kitware.com/dart/HTML/index.shtml*, 2011.

[8] Doc group virtual scoreboard. *http://www.dre.vanderbilt.edu/scoreboard/*.

[9] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker. A message passing standard for MPP and workstations. *Commun. ACM*, 39(7):84–90, July 1996.

[10] S. G. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *SIGSOFT FSE'06*, pages 253–264, 2006.

[11] E. Engström and P. Runeson. Software product line testing - a systematic mapping study. *Inf. Softw. Technol.*, 53:2–13, January 2011.

[12] GNU GCC. *http://gcc.gnu.org*, 2010.

[13] M. F. Johansen, O. Haugen, and F. Fleurey. A survey of empirics of strategies for software product line testing. *IEEE International Conference on Software Testing Verification and Validation*, pages 266–269, 2011.

[14] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Perform. Eval.*, 67:634–658, August 2010.

[15] S. Mahmood, R. Lai, and Y. Kim. Survey of component-based software development. *Software, IET*, 1(2):57 –66, april 2007.

[16] S. Mahmood, R. Lai, Y. Soo Kim, J. Hong Kim, S. Cheon Park, and H. Suk Oh. A survey of component based system quality assurance and assessment. *Inf. Softw. Technol.*, 47:693–707, July 2005.

[17] 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 International Conference on Software Maintenance 2003*, September 2003.

[18] Microsoft XP Error Reporting. *http://support.microsoft.com/?kbid=310414*, 2010.

[19] Netscape Quality Feedback Agent. *http://www.mozilla.org/quality/qfa.html*, 2010.

[20] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 65–69. ACM Press, 2002.

[21] K. Pohl and A. Metzger. Software product line testing. *Commun. ACM*, 49:78–81, December 2006.

[22] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI–The Complete Reference, Second Edition*. Scientific and Engineering Computation Series. MIT Press, 1998.

[23] The Mozilla Organization. bugs. `www.mozilla.org/bugs/`, 1998.

[24] The Mozilla Organization. Mozilla. www.mozilla.org/, 1998.

[25] Tinderbox. *http://www.mozilla.org/tinderbox.html*, 2010.

[26] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Towards incremental component compatibility testing. In *Proceedings of 14th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE-2011)*. ACM Press, June 2011.

[27] I.-C. Yoon, A. Sussman, A. M. Memon, and A. Porter. Direct-dependency-based software compatibility testing. In *ASE '07: Proceedings of the 22nd IEEE international conference on Automated software engineering*, Washington, DC, USA, 2007. IEEE Computer Society.

[28] I.-C. Yoon, A. Sussman, A. M. Memon, and A. Porter. Effective and scalable software compatibility testing. In *ISSTA '08: Proceedings of the International Symposium on Software Testing and Analysis*, Washington DC, USA, 2008. IEEE Computer Society.