Mutual Testing on Combined Software Components

M.Chandrasekhar Varma
Asst. Prof.,
Department of Computer Science & Engineering,
DNR College of Engineering & Technology,
Bhimavaram.

Abstract:- **All the systems are based on the software built components, most of the large systems contain same built in components but individual component systems contain different components and different working model based on system developed, maintained and tested and the test cases will vary from system to system. In practice, testers of different systems rarely collaborate as a result, redundancy of testing common components by obtaining information of each and every component and properly design test cases for avoiding redundancy. In this paper I have developed Mutual influence of component-based systems with shared components. 1. Related testing of common based systems. 2. Infrastructure and related tools to make easy communication and data sharing between testers. 3. Testing process to implement different collaborative testing. By this i can achieve better efficiency in testing and discover inter-component compatibility faults within a minimal time window after they are introduced.**

Keywords: Collaboration, Regression Testing, Data sharing, Efficiency.

1. INTRODUCTION

Now a day's companies are rely on third-party software components, Combine them together to implement their system. Each component in a component-based system may have multiple versions, thus there can be a large number of version configurations for a single software system. Independent component based systems may have new versions continuously released, and new end-user. If developers use Agile Testing or 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 effort by collaboration. Collaborative testing can not only boost test efficiency comparing to testing in isolation, but also provide opportunities to improve the quality of individual components. Our supposition is based on characteristics of component-based systems. The characteristic is that components in component-based software systems have dependency relationships between them, i.e., some components use or depend on other components. Opportunity 1: There should be relation among the provider and tester to test the individual and shared components and the information shared between them to intimate about changes in software or there results so testers can keep more concentrate on good testing easily.2: Distribute test effort and share results for common components to improve test quality. More specifically, when two or more component-based systems use at least one common component, developers of the systems can collaborate in the testing of the common component. i) Improve the quality of compatibility testing of component-based systems; and ii) boost the efficiency of testing software system configurations. The goal of this research is to develop automated collaborative testing theories and tools for individual developers of shared software components, so that their testing practice can be more efficient and with Third, develop different collaborative testing processes upon the infrastructure, so that testers can rely on information shared by others to coordinate their own testing and improve the efficiency and effectiveness of their local tests. As the initial step, i conjecture that overlap and synergy exist in testing functionally related components. Developed two collaborative testing processes that coordinate the local testing procedures of component developers. The first process is called ad-hoc collaborative testing, which requires minimal modification to the current practice of isolated component developers conducting their testing. In this process, automated tools of isolated developers will query Conch before building any configuration, or running any functional test. If there are any prebuilt configurations or results shared for the same testing task, the developers can just reuse them and avoid redundant effort. Otherwise, they can continue with their original procedure, and share their prebuilt artifacts and/or test.

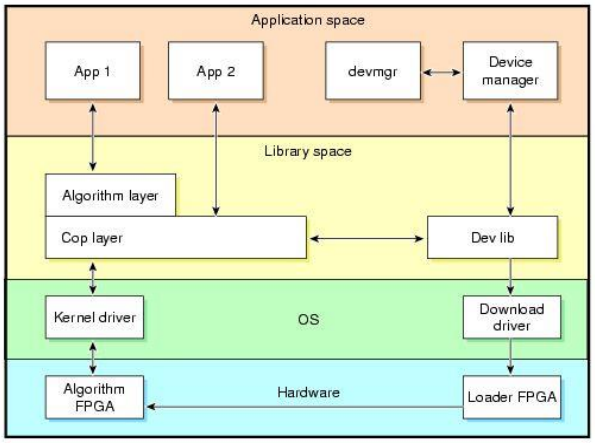

Fig 1:System with common Componenets

2. RELATED WORK

2.1 Distributed Software Development There has been much research of techniques on designing regression test cases and regression tests selection. Component-based

software systems, components are usually developed and maintained by different groups, each of which develops regression tests for their own component only. As a result, even though individual components may be well tested, the system consisting them may suffer compatibility faults across components. Many research target to address this problem by creating better cross-component regression tests. The research in this dissertation differs from the previous efforts, because we are relying on the regression tests created for individual components to improve the overall compatibility of component-based systems. This research is based on observations like; creating cross-component regression tests may not be feasible in some scenarios. Nilsson et al. developed a visualization technique for visualizing end-to-end testing activities involved in the continuous integration processes within projects or companies, so that such activities can be better arranged to support more efficient integration testing. However, Users install agents that automatically check out software from a repository, build the software, execute functional tests, and submit the results to the server.

2.2 Continuous Integration and Testing

Elbaum et al. designed algorithms to pre-select and prioritize test cases from test suites to make continuous integration processes more cost-efficient. Nilsson et al. developed a visualization technique for visualizing end-to-end testing activities involved in the continuous integration processes within projects or companies, so that such activities can be better arranged to support more efficient integration testing An important part of our work is the tools and infrastructure i provided to support coordinated collaborative testing, as part of the continuous integration. There are some distributed continuous quality assurance (QA) environments. However, the underlying QA process is hard-wired in Dart and Cruise Control and therefore other QA processes or implementations of the build and test process is not easily supported.

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

3. COLLABORATION TESTING

I describe my initial study of searching for overlaps in the testing processes of functionally related components. In this model i used to study how components get exercised by their user components in a component assembly. ***Induced Coverage:*** Suppose component a directly or indirectly uses component b, and a has a test suite T_a. In a system where a is successfully built on b, when running a's test suite, T_a, the fraction of b's coverage elements (lines, branches, functions, parameter values, faults, etc.) that get covered is called b's induced 25 coverage from a, represented as C^{a}_{b} To demonstrate the concept of induced coverage, i take the sub-CDG 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. Each column represents a line in E's source code, and each row shows the corresponding coverage. A filled block means the line is covered, and a blank one means that it is not. Union of Induced Coverage: When both components a and b use c, the union of their induced coverage for c ($C^{a}_{c} \cup C^{b}_{c}$) is defined as the fraction of c's elements that is covered by either a or b.

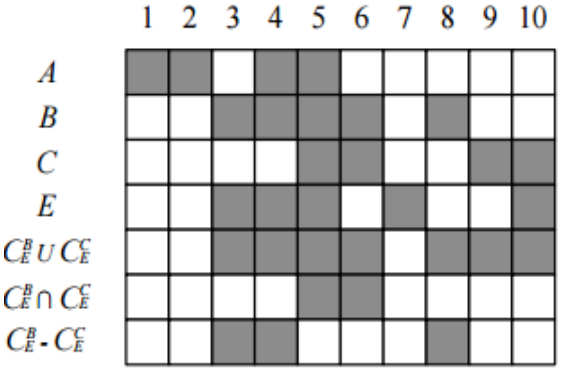

Fig 2: Induced Coverage Example

3.1 Line/Branch Coverage This analysis of functional testing examined how line and branch coverage changed depending on which component's tests were being run. Other coverage metrics, such as method coverage, dataflow coverage, etc., could also be used to measure the effectiveness of user components' test suites on testing the provider components.

3.2 Fault Detection i seeded faults in one provider component, and observed whether such faults were detected when running the test suites of both the component that contained the seeded faults and the component(s) that directly or indirectly used it. 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.

4. INFRASTRUCTURE AND RELATED TOOLS

The core component is a web-service based data sharing repository called Conch, which allows testing tools for different component developers and testers to share their testing artifacts and results. To support scalable caching and sharing of testing artifacts in the format of virtual machine images, i built an ad-hoc collaborative testing process upon this infrastructure, and evaluated its effectiveness as well as performance over a set of example components.

4.1 Environment Model In order to leverage independent testing efforts of component-based software systems, it is necessary to control the test environment in which a component is built and tested so that test results will be comparable across different testers. Thus, i provide a notional definition of a test environment as follows: Definition 1: An environment where a component built and tested in includes all pre-built component instances in a system, the tools to be used to build the new component, all source code needed by the build, and all other controllable factors known to determine the result of the component's build process and the correct functioning of the component. Controlling the environment in this way maximizes the likelihood that two testers building and testing the same component can share and combine their test results. That is, any differences in results should be attributable only to differences in how the components were tested, not in where or by whom they were tested. To gain this control, i attempt to standardize the test environment used by each tester. i have identified several factors that may affect the build and functional testing of components, and therefore must be captured by the test environment. These factors include: • Hardware parameters (processor type, memory system, etc.) • Operating system (architecture, kernel version, system core libraries, etc.) • Build environment (compiler, compiler options, extra instrumentation inserted, etc.) • Provider components (versions, their build settings and installation options, etc.) Of course, this approach is not bullet-proof. i cannot, for example, account for unknowable or random factors, such as transient hardware faults in one tester's computing device, which surely affect how a component behaves. A Virtual Machine (VM) with an installed operating system and pre-built core components is an intuitive way to encapsulate an environment, and sharing of prebuilt environments then becomes sharing of VM images. The description contains information about the hardware parameters of the VM, operating system information, pre-built components and their build options, and other information that may affect the test results. When accessing the repository, test tools search for VM images instantiating specific environments based on the description files. In this environment there are six components, including the operating system and a compiler.

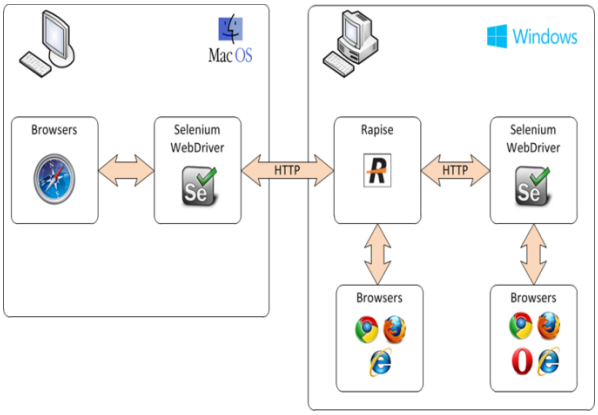

Fig 3: Environment Model

5. TESTING PROCESS

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

5.1 Strategy for Coordinated Collaboration

When a component is shared by multiple developer groups and a new version of the component is released, sets of regression configurations defined for its user components have to be tested by the groups. Because the component is shared, there must be overlaps in the regression configuration sets, and the overlaps – a set of partial configurations must be tested first. Conch selects one of the developer groups to test those partial configurations without causing test redundancy, based on the following factors: Availability: a binary value that indicates whether a developer group can immediately start testing a set of new regression configurations • Performance: how fast a developer group can complete testing on their testing resources • Reliability: how likely a developer group can complete assigned testing tasks The performance factor of a developer group G is defined as the ratio of the execution time required to run a sample test suite using the testing resources of the group and the resources at the Conch repository, as shown in Equation

$$PF(G) = T_G / T_{Conch}$$

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.

$$T\ F\ R(G) = F\ C_G / T\ C_G$$

In Equation , $T\ C_G$ is the total number of test suite execution requests that have been assigned to the group G, and $F\ C_G$ 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, i define the Expected Performance Factor (EPF) of the group as:

$$EP\ F(G) = P\ F(G)/\ (1 - T\ F\ R(G))$$

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, i first determine the user components for which functionality might be affected by the updated provider component. Then i 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, will then be requested to build and test the updated component over the partial configuration set.

Algorithm:CoordinateTester(C,CDG,A,PFs,FRs)

Data:
C: updated provider component
CDG: componenet dependency graph
A: availability of groups
PFs: performance factor values of groups
FRs: failure rate value of groups
Groups← available direct user comp. Developer groups;
Sort groups by EPF:
While groups!= Ø do
group← groups.getNext();
result← assigntask(group.C);
update FR of the group;
if result== Sucess then
update result in conch;
conch notifies subscribers of C's result;
break;
end
end

Algorithm 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 at result. If the group completes (or fails to complete) the test, the FR value of the group will be updated accordingly.

5.2 Regression Testing based on Cross-Component Coverage i have presented a strategy to coordinate multiple developer groups, while avoiding redundant test effort. However, in the end i 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. i showed that developers can save test effort up to 70% by selectively running regression test cases based on the mapping between the individual test cases of user components and the code coverage of provider components. In this part of the dissertation, coverage-based regression testing is conducted at two different granularity levels. If Conch maintains the code coverage mappings between each user component test case and each provider component, only a subset of the test cases that cover the updated regions of the provider component must be run. If Conch maintains the mappings between the test suite of a user component and each provider component and if a provider component update is relevant to one or more test cases of the user component, i rerun the whole test suite at a provider component update.

6. CONCLUSION

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

7. FUTURE WORK

My dissertation research is an initial study to search for benefits of collaborative testing. Several possible extensions and improvements can be made based on the current work. Improve Scheduling Algorithm is the coordinated collaborative testing process; Conch can schedule a common testing task to one of the affected component developers to avoid redundant testing. Improve Tests of Individual Components shows that testing of user components can test extra parts of provider components that are not covered by the test suites of the provider components themselves. Need to improve the Security and Consistency.

BIBLIOGRAPHY

[1] Collaborative Testing Across Shared Software Components by Teng Long
[2] Floris Erich, Chintan Amrit, and Maya Daneva. A mapping study on cooperation between information system development and operations. In Andreas Jedlitschka, Pasi Kuvaja, Marco Kuhrmann, Tomi Mnnist, Jrgen Mnch, and Mikko Raatikainen, editors, Springer International Publishing, 2014.
[3] Robert Cecil Martin. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR, Upper Saddle River, NJ, 2003.
[4] Ilchul Yoon, Alan Sussman, Atif Memon, and Adam Porter. Towards incremental component compatibility testing., 2011.

Web Sites
[1] http://essay.utwente.nl/63988/
[2] http://drum.lib.umd.edu/handle/1903/18149