Community-Based, Collaborative Testing and Analysis

Atif Memon, Adam Porter and Alan Sussman
Department of Computer Science, University of Maryland, College Park, MD
{atif,aporter,als}@cs.umd.edu

ABSTRACT

This article proposes a research agenda aimed at enabling optimized testing and analysis processes and tools to support component-based software development communities. We hypothesize that *de facto* communities—sets of projects that provide, maintain and integrate many shared infrastructure components—are commonplace. Currently, community members, often unknown to each other, tend to work in isolation, duplicating work, failing to learn from each other's effort, and missing opportunities to efficiently improve the common infrastructure. We further hypothesize that as software integration continues to become the predominant mode of software development, there will be increasing value in tools and techniques that empower these communities to coordinate and optimize their development efforts, and to generate and broadly share information. Such tools and techniques will greatly improve the robustness, quality and usability of the common infrastructure which, in turn, will greatly reduce the time and effort needed to produce and use the end systems that are the true goal of the entire community.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Testing tools

General Terms

Experimentation,Measurement,Verification

Keywords

Component-based software development communities

1. INTRODUCTION

Over the past few years we have worked extensively with multiple space physics research groups. To support their cutting edge science research, each of these groups has had to develop and evolve their own specialized software systems. As is increasingly common these days, these systems use a component-based architectural approach in which mission-specific components are integrated from large numbers of infrastructure components, including compilers, math libraries, data management packages, communication frameworks and simulation models.

As software engineers looking in from the outside, we see individual groups working largely in isolation from each other, even though the different underlying systems are composed from many of the same infrastructure components. The inevitable result is that the groups frequently duplicate integration effort, reinvent similar code fixes, and miss out on system knowledge gained by other groups.

We argue that these groups form a *de facto* **component-based software development community**. Moreover, we argue that development processes optimized for the community could be much more efficient and effective than the current practice in which individual projects work completely in isolation. That is, if communities can intelligently coordinate their efforts and share hard-won information, then they can greatly improve the robustness, quality and usability of the common infrastructure. This improved infrastructure will likewise greatly reduce the time and effort needed to develop and deploy the mission-specific components, which, in the end, is the true goal of the entire community. Finally, we believe that development communities are common, forming naturally where common needs, requirements, hardware and industry standards lead software developers to draw on common infrastructure components.

We, the authors of this paper, have personal experience with communities in investigation-based sciences, in telecommunications and in university administration. We believe that many others exist as well. Therefore, models, tools, techniques and collaboration platforms that help development projects: (1) identify the communities to which they belong, (2) express their development goals, (3) coordinate effort, and (4) share information, could save a great deal of time and effort for component-based software developers.

To achieve this vision we first need to formally define component-based software development communities, observe and document how they evolve and interrelate, and study their development processes to understand where process optimizations and sharing are most appropriate. Initially, we will focus our attention on testing and analysis processes for the space physics community we have already worked with. We see no reason, however, that this approach could not be extended to other development processes and communities in the future. Second, we will focus on cre-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.
Copyright 2010 ACM 978-1-4503-0427-6/10/11 ...$10.00.

ating powerful new community-based, collaborative testing and analysis processes, with shared information repositories and collaborative tools at their core. These new collaborative processes will require research advances and empirical evaluation of tools, algorithms and computing infrastructure to 1) model software development communities and their systems, 2) intelligently pool, coordinate and optimize decentralized community resources and testing and analysis efforts, 3) generate timely, detailed and accurate information about the components, their inter-dependencies, configurations, compile- and run-time constraints and runtime behaviors and 4) feed this information back to community members along with tools that allow members to conduct their own analyses in support of their own specific needs.

2. A NOTIONAL RESEARCH AGENDA

Software engineering techniques and tools have historically been designed to apply to **THE** system. Increasingly, however, there is no single system. Instead there are multiple systems emerging from a complex component assembly, organized around a large and complex *design space* [7]. A system's design space refers to the dimensions of controlled variation that it supports, i.e., there is typically a software base with hooks that allow controlled variation in features, versions, algorithms, platforms, architectures, standards implementations, and so forth. We believe that design spaces have identifiable structure, and that this structure can be leveraged to define powerful testing and analysis algorithms. These algorithms can then be executed to efficiently and effectively test across the system's design space. When the systems under test are used by communities, the entire community can define the testing goals and algorithms, and can share the effort and fruits of executing those algorithms.

To realize this vision the research community must advance in multiple directions. We will need to (1) define modeling formalisms that capture component structures and testing goals, (2) design new testing and analysis algorithms for testing systems with large and complex design spaces (3) create decentralized execution platforms that enable de facto communities to easily work together, (4) create information repositories for managing shared data about community components, and for keeping track of hardware configurations, test cases, and test results, and (5) add higher level algorithms and tools specifically aimed at optimizing testing across the community.

At a high level, these advancement will likely support at least the following key steps:

1. Explicitly **model the system design space.** Component providers create design space models for their individual components. Each model exposes dimensions of variability for that component, such as hardware platforms, operating systems, feature sets, compile- and run-time options, etc. Infrastructure tools then automatically integrate each individual component model with the models of the components it depends on to create an integrated model of the system under test.

2. Define test coverage criteria and generate test plans. System models implicitly define all configurations of the system to test. Since exhaustive testing is generally infeasible because of the tremendous number of possible configurations, we must define sampling strategies over the design space. Applying a sampling strategy to the model yields the specific set of configurations to be tested, called the test plan.

3. Execute the test plan in parallel across a virtual computing grid. Executing the test plan involves decomposing the test plan into independent test jobs, where each job typically focuses on one configuration or group of equivalent configurations. Numerous optimizations can be applied to limit duplicated effort, and to coordinate the activities of multiple test plans. Tools then distribute the jobs to client machines on the grid. The computing grid may contain different resource types, such as an ad-hoc federation of community-provided resources (e.g., a desktop grid), generic centrally managed resources, or specialized resources such as a high-end supercomputer. The choice of resources will depend on multiple factors, such as the component's maturity, testing goals, deadlines and budget, and the need for specific hardware environments. However, the desktop grid option will be extremely important for small- to mid-size communities that lack a resource-rich central authority.

4. Execute and measure the system under test. As the test jobs execute, execution data is collected, and the data is returned to the information repository. The data can include test results, coverage information, detailed crash reports, etc. Depending on how the test plan is defined, incremental results may be merged and analyzed to guide subsequent iterations of the test process.

5. Store, analyze and publish results. Data from test processes are stored in the community-accessible information repository. Besides being used by ongoing test processes, such data could also be analyzed and published via standardized visualizations, which show the stability of particular configurations, the test status of the latest system version, the effect of particular configuration parameters on standard performance benchmarks, etc. There should also be tools that enable community members to develop and share their own data analyses.

3. SOME FIRST STEPS

Our research vision has three related focus areas. Each area supports testing and analysis at a different level of granularity. These levels are (1) **testing individual systems**, (2) **testing component assemblies**, and (3) **testing across the community**. Testing individual systems focuses on techniques and tools for testing systems across their design spaces. The second focus, testing component assemblies, breaks open a system, leveraging knowledge of its architecture structural. That is, we view a component-based system as an assembly of individual components and then hierarchically test its dependent sub-assemblies up to and including the entire component-based system under test. Given the many different ways in which systems can be built and configured, reuse of previously generated test artifacts can greatly reduce overall effort. The final focus area extends component-based testing to the community by creating tools and processes that optimize testing across the common components used by the entire community. The results of all testing activities are stored in an information repository that is accessible both for use in ongoing testing processes and for use by community members.

3.1 Testing individual components

This area focuses on tools and techniques for testing individual components across their design spaces. In this con-

Option	Settings	Interpretation
COMPILER	{gcc4.1.2, SUNCC5_1}	compiler
AMI	{1 = Yes, 0 = No}	Enable Feature
CORBA_MSG	{1 = Yes, 0 = No}	Enable Feature
run(T)	{1 = True, 0 = False}	Test T runnable
ORBCollocation	{global, per-orb, NO}	runtime control
Constraints		
AMI = 1 → CORBA_MSG = 1		
run(Multiple/run_test.pl) = 1 → (Compiler = SUNCC5_1)		

Figure 1: Some QUASI options and constraints.

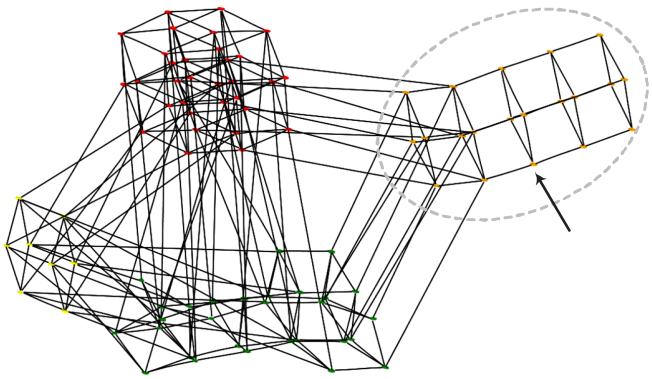

Figure 2: Nearest Neighbor Search.

text an individual component need not be truly monolithic. It could contain other sub-components, but such architectural information would not be used in this focus area. In our own research we have started by developing a system and approach currently called QUASI (Quality as a Service Infrastructure) [11, 5, 6, 9]. QUASI's analytical cornerstone is a model of the design space that implicitly captures all configurations on which test jobs might run. Configurations then serve as parameters to generic test jobs. In actual systems, not all option setting combinations are legal, so QUASI also supports inter-option constraints that limit the setting of one option based on the settings of others. Table 3 presents some sample options and constraints we have previously taken from 2M+ lines of code from the ACE+TAO CORBA project [3]. The sample options refer to things like the end-user's compiler (COMPILER); whether or not to compile in certain features, such as support for asynchronous messaging invocation (AMI); whether certain test cases are runnable in a given configuration (run(T)); and the level at which to set a run-time optimization (ORBCollocation). One sample constraint shows that if the AMI support option is turned on, then the CORBA messaging option must be turned on. However this modeling is done, design space models need to become first class development artifacts.

Given a design space model and generic test jobs that operate on design space points, test processes are defined by (1) creating programs that systematically "visit" points in the design space, which means having a client execute a test job in the configuration defined by that point and having it return the results to a QUASI server, (2) defining analysis techniques that merge and analyze incremental test results, and (3) creating decision rules to dynamically and intelligently steer the visitor programs based on incoming results. In general, component providers implement application-specific test programs, while the visitor programs that define the test process are predefined by QUASI (advanced users can extend the set of test processes).

At execution time, QUASI test processes run on client machines making calls to a QUASI server whenever the client is available to perform test tasks. When contacted, QUASI uses planning technology to assign the current best test job to that client, where "best" is application-specific and is defined by the visitor programs discussed above, the state of the test process, and the characteristics of the client machine offering service. QUASI bundles the code artifacts, assembly parameters, build instructions, and test-specific code associated with the selected job. This data is sent to the client, which executes it and returns the results for collection and analysis, also causing decision rules to be triggered to effect steer the overall process.

We have developed several novel test processes using the QUASI infrastructure, some of which we not touch on. The key point to note is that the design space model provides a great deal of information that can be used in structuring the test processes. Our fault characterization process systematically tests system configurations, feeds the test results to a machine learning algorithm and outputs a model describing the configuration options and settings that cause the observed test failures. These models help developers quickly narrow down the causes of specific failures. As one implementation of this process we developed a search-based strategy that uses distance measures to identify and test the "nearest neighbors" of failing configurations, allowing quick characterization of specific problems. We applied this process to one version of the ACE+TAO CORBA project with ∼115K possible configurations, using 120 CPUs in our dedicated evaluation testbed. This process is depicted in Figure 2. Nodes represent configurations and arcs connect pairs of configurations that differ in the setting of exactly 1 configuration option. The dotted line surrounds a neighborhood of ACE+TAO configurations that failed for the same underlying reason. In another implementation [8, 4], we developed techniques for systematically sampling the design space, using mathematical *covering* arrays [2]. A covering array induces a configuration sample in which all t-way interactions between options are observed at least once. In a third effort, we developed algorithms based on Design of Experiments (DoE) theory [1] to quicky determine whether recently changed software had suffered performance degradations in any system configuration [10].

3.2 Testing Component-Based Systems

Component-based systems present new challenges over and above those found in individual components. One reason is that the number of deployable component configurations is very large since each component in the system can have multiple versions, and can have complex dependencies on multiple other components. Collectively, a system's many end users might deploy a large number of these configurations. Additionally, components and their dependencies can change without notice, especially if components are developed and maintained by separate groups of developers. Finally, developers are under pressure to to support a broad set of configurations if they want their systems to be used as widely as possible.

In previous work, we developed *Rachet*, a process and in-

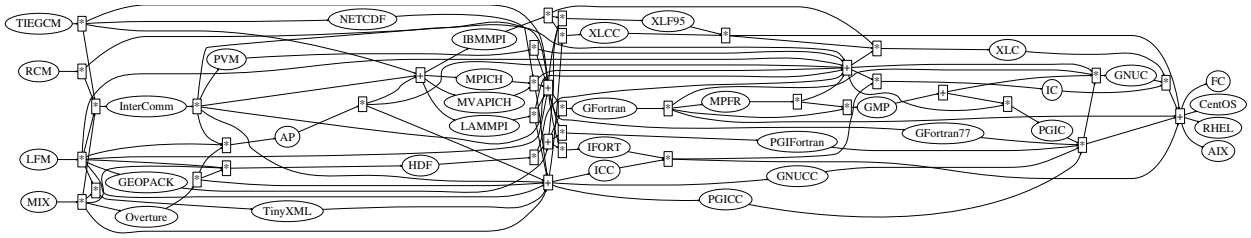

Figure 3: The (partial) CDG for 4 space physics systems.

frastructure for testing component-based systems [13, 14, 12]. At a high level the process has several parts. First, developers model the system under test (SUT) using a formal representation with two parts: a directed acyclic graph called the *Component Dependency Graph (CDG)* and a set of *Annotations*. A CDG specifies the components making up the system and specifies inter-component dependencies by connecting components with AND and XOR relationships. *Annotations* include version identifiers for components, and constraints between components or over configurations, written in first-order logic. Together, the CDG and annotations define the set of all valid configurations for the SUT. An example CDG for four space physics simulation components is shown in Figure 3.

Next, developers specify the component versions to test and specify a coverage criterion. Given a coverage criterion, Rachet automatically produces a set of configurations that satisfies it. Rachet then combines the produced configurations into a tree data structure called a *prefix tree*. The prefix tree acts as a *test plan*; each path from the root node to a leaf node corresponds to one desired configuration. The rationale behind combining configurations is that it may be possible to reduce the overall test time by reusing partially completed configurations on a given test machine. An example test plan is shown in Figure 4.

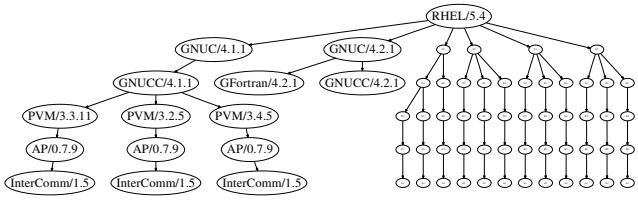

Figure 4: An Example Test Plan

Finally, Rachet executes the test plan by scheduling configurations to multiple client machines and collecting results. Instead of distributing full configurations, Rachet distributes partial configurations, called *tasks*, to increase reuse of partial tests that share sub-configurations (prefixes in the tree). Again, whatever tools and techniques are finally produced, they will need to leverage architectural dependencies and support compositional approaches to testing and analyzing component-based systems.

3.3 Community-Driven Testing

Community-driven testing is largely a new research area with some tentative initial steps being taken by different research and industrial groups. Substantial work will need to be done and the best models and approaches are currently unclear. However, we believe that any solution will likely include system modeling, test plan generation, distributed test plan execution, data collection, and data analysis and publication. Below we focus on some areas in which extensions are needed.

System modeling. In a community there is no central actor who models the entire system design space, so system modeling must be compositional. For example, component providers first create and publish models for their own components, providing information such as the source code repository location, dependent sub-components, component versions, configuration options and settings, configuration constraints, etc. These component models are then published for use by other components that depend on them. Next, component owners create system models, similar to that described earlier, but describing only the component's direct dependencies. They must also specify any applicable configuration constraints.

Test plan generation. Once modeling is done, component owners can start a test process for a given component. Tools will instantiate a test process from the component's models and from all the dependent component's models. The component provider can then test the system and each of its components across the full range of their configurations or can restrict the test process to specific parts of the design space, e.g., by limiting testing to specific versions of specific components, or to specific configuration settings for some of the dependent components. Component owners will then select test coverage criteria and given this information can then generate a test plan to be executed across the testing grid, using virtual machines, real machines or both. Here various optimizations are possible. For instance we could generate the test plan to minimize new work, taking advantage of the fact that some tests jobs may already have been performed in other test processes. Or we could optimize for broader coverage of the design space. More specifically, the test plan avoids already tested configurations in favor of ones not previously tested.

System execution and monitoring. Once the test plan is generated, the test process can start. Because many test processes are long-running, new results can be computed between the time the test plan is generated and the time that a specific job starts. Therefore, component providers can specify whether to include or exclude community results from their long-running test processes. Should they choose to include them, then prior to executing each job, the information repository is queried for existing results. Only if no existing results are found would the job be executed.

Analysis and visualization. All results from executing the test processes will be stored into the community-accessible information repository both for later test processes and for analysis and visualization. Based on the needs of the community members standard visualizations can be created for popular analyses. Community members will be able to developer their own visualizations as well.

4. REFERENCES

[1] G. E. P. Box, W. G. Hunter, and J. S. Hunter. *Statistics for experimenters : an introduction to design, data analysis, and model building*. 1978.

[2] R. Brownlie, J. Prowse, and M. S. Padke. Robust testing of AT&T PMX/StarMAIL using OATS. AT&T Technical Journal, 71(3):41–7, 1992.

[3] DOC Group. ACE and TAO. deuce.doc.wustl.edu/Download.html/, 2004.

[4] S. Fouché, M. B. Cohen, and A. Porter. Incremental covering array failure characterization in large configuration spaces. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 177–188, 2009.

[5] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan. Skoll: Distributed Continuous Quality Assurance. In *Proceedings of the 26th IEEE/ACM International Conference on Software Engineering*, Edinburgh, Scotland, May 2004. IEEE/ACM.

[6] Skoll – a distributed continuous quality assurance environment, 2010. http://www.cs.umd.edu/projects/skoll.

[7] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–108, 2001.

[8] C. Yilmaz, M. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 31(1):20–34, 2006.

[9] C. Yilmaz, A. Krishna, A. Memon, A. Porter, D. C. Schmidt, A. Gokhale, and B. Natarajan. Main Effects Screening: A Distributed Continuous Quality Assurance Process for Monitoring Performance Degradation in Evolving Software Systems. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, St. Louis, MO, May 2005.

[10] C. Yilmaz, A. Porter, A. S. Krishna, A. M. Memon, D. C. Schmidt, A. S. Gokhale, and B. Natarajan. Reliable effects screening: A distributed continuous quality assurance process for monitoring performance degradation in evolving software systems. *IEEE Transactions on Software Engineering*, 33(2):124–141, 2007.

[11] C. Yilmaz, A. Porter, and D. C. Schmidt. Distributed continuous quality assurance: The Skoll project. In *Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, Portland, Oregon, May 2003. IEEE/ACM.

[12] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Prioritizing component compatibility tests via user preferences. In *ICSM '09: Proceedings of the 23rd IEEE International Conference on Software Maintenance*, Alberta, Canada, 2009. IEEE Computer Society.

[13] 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.

[14] 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.