Direct-Dependency-based Software Compatibility Testing

Il-Chul Yoon, Alan Sussman, Atif Memon, Adam Porter
Dept. of Computer Science
University of Maryland
College Park, MD, 20742 USA
{iyoon,als,atif,aporter}@cs.umd.edu

ABSTRACT

Software compatibility testing is an important quality assurance task aimed at ensuring that component-based software systems build and/or execute properly across a broad range of user system configurations. Because each configuration can involve multiple components with different versions, and because there are complex and changing interdependencies between components and their versions, it is generally infeasible to test all potential configurations. Therefore, compatibility testing usually means examining only a handful of default or popular configurations to detect problems, and as a result costly errors can and do escape to the field.

This paper presents an improved approach to compatibility testing called RACHET. We formally model the configuration space for component-based systems and use the model to generate test plans covering user-specified portion of the space – the example in this paper is covering all *direct dependencies* between components. The test plan is executed efficiently in parallel, by distributing work so as to best utilize test resources. We conducted experiments and simulation studies applying our approach to a large-scale data management middleware system. The results showed that for this system RACHET discovered incompatibilities between components at a small fraction of the cost for exhaustive testing without compromising test quality.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Design, Experimentation

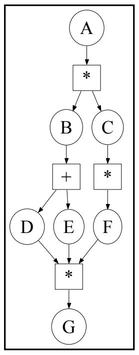
Keywords

component-based software system, compatibility testing

1. INTRODUCTION

Over the last decade software engineering researchers have created tools and techniques that facilitate assembling large systems from independent components. These advances are now embodied in technologies such as configuration management systems, interconnection standards, middleware frameworks and product-line and service-oriented architectures. Despite their many advantages, these technologies also tend to push many problems and complexities into configuration and integration activities. Consider for instance, the InterComm system [2, 4]. InterComm is data management middleware that supports large coupled scientific simulations. For example, a simulation studying the effect of solar weather patterns on cell phone performance in the U.S. might involve multiple simulation modeling applications: solar activity on the sun's surface, radiation propagation in the region between the sun and the earth, the effects of the solar wind on earth's ionosphere, etc. InterComm couples the applications, which may be written in different languages and run in parallel on diverse operating systems, and enables data to be transferred between them at appropriate times and at appropriate simulation scales. To support that, InterComm requires several *system components* including multiple C, C++ and Fortran compilers, parallel data communication libraries, process management libraries and a structured data management library. Each component has multiple versions and there are complex dependencies between components and their different versions.

Developers of component-based systems like InterComm are bedeviled by configuration and integration problems. First, the sheer number of combinations of components and their versions makes it very difficult to test all possible configurations. Second, the components and their dependencies can change without notice. Finally, while it might appear that these issues could be avoided by radically restricting the set of supported configurations, in reality that would unacceptably restrict the potential user base. In practice, developers often approach this problem by conducting *compatibility testing* [3]. This involves selecting a set of *configurations* – each configuration is an ensemble of component versions that respects known dependencies – and testing whether each configuration behaves (builds and functions) properly. Because the set of possible configurations is very large and because automated support is limited, to perform compatibility testing, developers often pare down the set to a handful of popular configurations [5] or use only configurations that can be realized in test machines [1]. This means that the software is released with nearly all of its possible



Version Annotations

Component	Versions
A	A_1
B	B_1, B_2, B_3, B_4
C	C_1, C_2, C_3
D	D_1, D_2
E	E_1, E_2, E_3, E_4
F	F_1, F_2, F_3, F_4, F_5
G	G_1

Constraints
$(\text{ver}(C) = C_3) \rightarrow (\text{ver}(F) \geq F_3)$

Figure 1: An Example CDG and Annotations

field configurations untested. So costly errors can and do escape to the field. To improve this situation, we develop a process, algorithms and tools to perform compatibility testing effectively and efficiently. For this paper, we restrict our testing focus to determining whether the software system under test builds (compilation and deployment) successfully without any error, but we can and will extend the work to functional and performance testing.

2. THE RACHET PROCESS

This section describes the steps needed to perform compatibility testing of a given component-based system under test (SUT) using RACHET.

1. Model the software system

To define the configuration space of a SUT, developers model the components, versions, inter-component dependencies and constraints. This information is captured into a formal model containing a *Component Dependency Graph (CDG)* and *Annotations*. The CDG is a directed acyclic graph that encodes components and their inter-dependencies. The example CDG depicted in Figure 1 shows the dependencies for a SUT called `A`. The figure shows that `A` requires two components `B` *and* `C` (captured using an `AND` node represented by a `*`). Component `B` requires one of `D` *or* `E` (captured using an `XOR` node represented by a `+`); `C` requires `F`; and `D`, `E` and `F` all require `G`, the bottom node that does not depends on any other component. (The bottom node may represent an operating system, but that is not required.) Additional information not encoded in the CDG is specified as *Annotations*. Annotations may include (1) version identifiers to be considered for each component, (2) inter-component constraints and configuration constraints described in first-order logic. For example, in Figure 1, component `C` has three version identifiers and component `C's` version C_3 may only be built with `F`'s versions F_3 and higher.

2. Determine coverage criteria

The model from Step 1 represents a configuration space, containing a large number of ways in which the SUT may be legally configured. Developers must now determine exactly which parts of the space will be tested for compatibility. For example, they may decide to test every possible configuration for the SUT. Since that is often not feasible they may opt for more practical criteria that *systematically sample* the space.

Our approach starts from the observation that a successful component installation is most influenced by the components on which it *directly* depends. In this paper we say that component `A` is *directly dependent* on component `B` if and only if there is a path from `A` to `B` in a CDG that does not contain any component node, and we propose a coverage criterion called *DD coverage* that tests all *version combinations* (VCs) among a component and other components on which it directly depends.

3. Develop test configurations

RACHET automatically produces configurations satisfying the user-specified coverage criteria. Each configuration is a set of VCs where each VC is a (c_v, dep) pair. c_v denotes a component version to build and *dep* is a set of component versions used to build c_v. Since component versions represented by *dep* are used to set up the proper environment for c_v, each VC represents a partial order to build components.

To satisfy the DD coverage criterion, each VC must be covered by at least one configuration. To do that, we first generate all possible VCs for each component in a CDG, considering direct dependencies of the component. Then, for each uncovered VC (c_v, dep) of components in the CDG, a test configuration is produced by choosing (covering) appropriate VCs for the components specified in *dep* while traversing the CDG in depth-first order recursively, until all VCs necessary to build c_v are selected.

In addition, developers need to determine how to test each component in a configuration. If, for example, the goal is to determine whether a component can be built without any error on top of components it depends on, then a set of instructions to build each component should be specified. In practice, such instructions may be implemented as generic build scripts parameterizable for components to be built.

4. Generate test plans

From the observation that multiple configurations share partial orders represented by identical sets of VCs, a test plan is synthesized from the configurations produced in Step 3. To execute the configurations efficiently on multiple machines, each configuration is linearized into a totally ordered component build sequence, respecting the partial orders represented by VCs. Then, linearized configurations are merged into a *prefix tree* called a *test plan*. The VCs in configurations are mapped into nodes in the tree and the work to build identical prefixes (build sequences) may be shared for testing multiple configurations.

5. Execute test plans

RACHET executes a test plan by intelligently distributing work for each node to multiple machines and collecting the results. The work for each node is a build sequence of components represented by the VCs corresponding to the nodes in the path from the root to the node, and work may be reused if it is shared by multiple configurations. Components are built in a *virtual machine* (VM) running on each machine, without contaminating persistent machine state. Since we focus on testing successful component builds, RACHET checks successful component installation by monitoring whether the installation process for the component finishes without any error.

A novel aspect of RACHET is that it supports contingency strategies, which dynamically modify test plans if needed so as not to lose test coverage. During compatibility testing, failures in low-level components prevent testing higher-level dependent components. In this case, RACHET creates new configurations that try to build the higher-level components from other successfully built low-level ones.

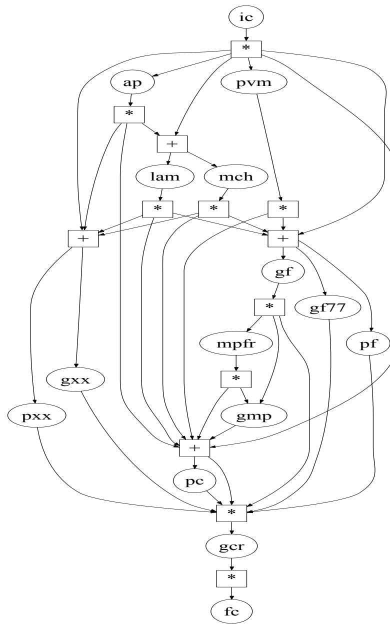


Figure 2: CDG for InterComm

3. EMPIRICAL STUDIES

We implemented the RACHET process into a tool with a client/server architecture. The server produces test configurations, synthesizes a test plan and distributes build sequences to a set of clients, each running a VM to test correct component builds. In this section, we present results from several empirical studies of our approach. In these studies we apply the RACHET process to the InterComm middleware system described in Section 1. In particular, we focus on examining: (1) the costs and benefits of the RACHET process, comparing direct-dependency-based testing to exhaustive testing, and (2) the RACHET process' behavior as we modify key process design parameters, such as the number of test machines used and the amount of space dedicated to caching partial results.

3.1 Experimental Setup

To execute RACHET process, we first created a model working with the InterComm developers. It consists of the CDG shown in Figure 2, version annotations depicted in Table 1, and additional constraints described below.

These constraints apply to individual configuration appearing in a test plan. First, if multiple GNU compilers are used (gcr, gxx, gf and gf77), they must have the same version. Second, only a single MPI component (i.e., lam or mch) can be used in a given configuration. Third, only one C++ compiler and version (gxx version X or pxx version Y) can be used. Fourth, in a single configuration if C and C++ compilers are used, they must be developed by the same vendor (i.e., GNU Project or PGI).

Based on the model, two test plans are generated by RACHET. The first test plan is based on exhaustive coverage of the model. This test plan contains 3552 configurations, requiring a total of 39,840 components to be built. Of these, 9919 are distinct nodes in the combined test plan. The other test plan, called DD, covers all direct dependencies in the model. This plan contains 211 configurations, requiring 1897 component builds, of which 649 are distinct in the plan.

Comp.	Version	Description
ic	1.5	InterComm, the SUT
ap	0.7.9	High-level array management component
pvm	3.2.6, 3.3.11, 3.4.5	Parallel data communication component
lam	6.5.9, 7.0.6, 7.1.3	A library for MPI (Message Passing Interface) standard
mch	1.2.7	A library for MPI
gf	4.0.3, 4.1.1	GNU Fortran 95 compiler
gf77	3.3.6, 3.4.6	GNU Fortran 77 compiler
pf	6.2	PGI Fortran compiler
gxx	3.3.6, 3.4.6, 4.0.3, 4.1.1	GNU C++ compiler
pxx	6.2	PGI C++ compiler
mpfr	2.2.0	A C library for multiple-precision floating-point number computations
gmp	4.2.1	A library for arbitrary precision arithmetic computation
pc	6.2	PGI C compiler
gcr	3.3.6, 3.4.6, 4.0.3, 4.1.1	GNU C compiler
fc	4.0	Fedora Core Linux operating system

Table 1: Version Annotations for InterComm CDG

We executed the test plans on three clients, each with a P4 2.4GHz CPU, 768MB memory and 150GB of disk space, all running Linux Fedora Core 4. The server is equipped with a P4 2.4GHz CPU and 512MB memory, running Red Hat Linux 3.2.3. All machines were connected via Fast Ethernet. The cache size for each client is set to 64, which means that each client is capable of storing up to 64 VMs simultaneously.

3.2 Cost-Benefit Assessment

As stated previously, the exhaustive test plan involves 3552 configurations, while the DD test plan contains only 211. Since in our setup building a configuration for InterComm from scratch takes can take up to 3 hours, we estimate that a naive execution plan where each configuration is built completely from scratch would take up to 10,600 CPU hours for the exhaustive test plan and about 630 CPU hours for the DD test plan. As there is no work shared across builds, the worst-case turnaround time given n machines would then be 3 hours times the ceiling of the number of configurations divided by n. For our 3 machine situation then, the expected worst case turnaround time is 3552 hours for the exhaustive test plan and 210 hours for the DD test plan. However, RACHET shares build effort across configurations and machines and many component versions in configurations failed to build. Therefore the actual turnaround time for the exhaustive test plan was 110.1 hours, during which 1148 of the 9919 nodes built successfully. For the DD test plan, the turnaround time was 36 hours, with 320 of 649 nodes built without any error.

In addition to cost savings we must also consider any lost effectiveness due to testing only direct dependencies. To do this we first mapped all component builds in the exhaustive test plan onto component builds in the DD test plan. We examined all failures in the exhaustive test plan and for each failure, noted the specific component version that failed to build and the other component versions on which it directly depended. Then, we checked whether the same component version failed in the same context when we tried to build it during the DD test plan execution. We found that for the system studied each failure in the exhaustive test plan is covered by one in the DD test plan during this analysis.

In summary these results suggest that RACHET using the DD test plan was able to detect build incompatibilities at

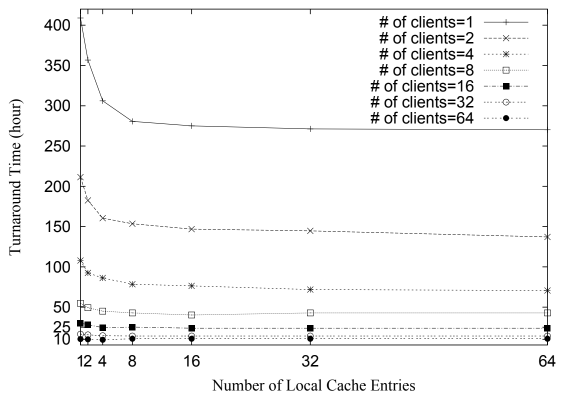


Figure 3: Turnaround times, varying the number of machines used for the exhaustive test plan.

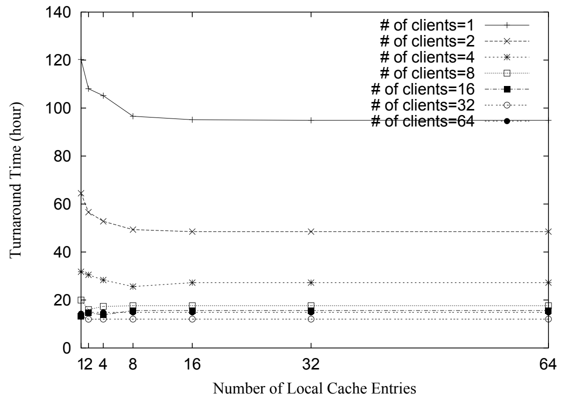


Figure 4: Turnaround times, varying the number of machines used for the DD test plan.

greatly reduced cost when compared to exhaustive testing. Furthermore, for this SUT direct-dependency-based testing achieved these savings without compromising test effectiveness. We note that these results are specific to this SUT.

3.3 Understanding Process Trade-offs

The experimental results presented in Section 3.2 showed that direct-dependency-based testing quickly and accurately detected build incompatibilities between components. However, the experiment explored only a single instance of the RACHET process – i.e., executing across 3 machines with a local cache size of 64. To better understand the trade-offs between these process parameters and to help pinpoint areas for further improvement we developed a discrete event simulator that mimics the behavior of the key components found in the actual RACHET tool. The output of the simulation is the turnaround time for completing a test plan using a given number of machines, each with a given cache size. The work times used by the simulation are computed from the average work times seen during the experiments from Section 3.2. These work times include the time needed to build a component, to transfer a VM over the network, to compress or decompress a VM, and to start up a VM. The time to handle events and to manage the test plan are not modeled in the simulation. The simulation was performed on a machine with a P4 2.4GHz CPU and 512MB memory, running Red Hat Linux 3.2.3.

As a control, we compared the simulated turnaround times to the actual turnaround time for the setup used in the experiments from Section 3.2, and we found that the simulated times were roughly 10% less than the actual times for both the exhaustive and DD test plans. Based on this we believe that the simulated times are accurate enough to help us understand the behavior of the real RACHET system.

Figures 3 and 4 show simulated turnaround times using different numbers of machines and cache sizes for the exhaustive and DD test plans, respectively. Each line corresponds to different numbers of machines (clients). One observation is that the turnaround times decrease a fair amount (about 20%) as cache sizes increase from 1 to 8 for all system sizes (number of machines). Beyond a cache size of 8, however, little benefit is seen. Increasing the system size from 1 to 64 has a more substantive effect. In general, we see that doubling the number of machines decreases turnaround time by somewhat less than half until 16 machines – some overhead is seen due to dependencies between components that make some machines remain idle, and some extra time is needed to transmit VMs across the network, as local cache hit rates drop when the workload is spread across more machines.

4. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an improved approach called RACHET to test compatibility of component-based systems. Our experimental results for the system studied show that RACHET can detect *inter-component incompatibilities* rapidly and effectively, at a fraction of the cost of exhaustive testing and without compromising test quality. We also explored several tradeoffs among process parameters, such as the number of computers used and the size of the cache on each machine. Based on these results we plan to examine diverse strategies to execute test plans efficiently in a parallel environment, to extend our work to new SUTs, new test scenarios, including functional and performance testing, and to investigate other cost-effective test coverage criteria.

Acknowledgments

This research was supported by the National Science Foundation under grants #CCF-0447864, #ATM-0120950 and #CNS-0615072, the Office of Naval Research under grant #N00014-05-1-0421, and NASA under grant #NNG06GE75G.

5. REFERENCES

[1] A. Duarte, G. Wagner, F. Brasileiro, and W. Cirne. Multi-environment SW testing on the Grid. In *Proc. of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging*, Jul. 2006.

[2] J.-Y. Lee and A. Sussman. High performance communication between parallel programs. In *Proc. of HIPS-HPGC 2005*, Apr. 2005.

[3] W. E. Lewis. *Software Testing and Continuous Quality Improvement.* CRC Press LLC, Apr. 2000.

[4] A. Sussman. Building complex coupled physical simulations on the Grid with InterComm. *Eng. with Computers*, 22(3–4):311–323, 2006.

[5] VMware Inc. Streamlining software testing with IBM® Rational® and VMware^{TM}: Test lab automation solution - whitepaper, 2003.