

# Automating Software Compatibility Testing

Il-Chul Yoon  
Dept. of Computer Science  
University of Maryland  
College Park, MD, 20742 USA  
iyoon@cs.umd.edu

## 1. INTRODUCTION

Many software systems rely on multiple components, each with multiple versions, and the potential field environments for the components and their various versions can be very heterogeneous. Therefore, to ensure that a software system behaves (builds and functions) properly on a range of field environments, the compatibility of the software system with the environments needs to be validated.

To perform software compatibility testing, developers first need to select a set of *configurations* (expected field environments), where each configuration is an ensemble of component versions with dependencies on other components in the configuration. However, it is rarely feasible to test all potential configurations, since the number of configurations can be large, and since the components used for the system have complex dependencies that may change without notice, especially if reused components are developed by separate groups of developers. Moreover, there has been limited support to automate software compatibility testing. Consequently, in practice, developers have examined only a handful of popular configurations to perform compatibility testing. This virtually guarantees that the software system will be released with many possible configurations untested. As a result, costly errors can escape to the field.

The goal of this research is to investigate methods to create an automated framework supporting software compatibility testing, and efficiently utilize limited test resources. This involves modeling the configuration space for software systems, selecting a small set of effective configurations and testing the behavior of components in those configurations.

As an initial effort to achieve this goal, I encode the entire configuration space of a software system into a graph-based model and used the model to generate a test plan consisting of configurations that cover all *direct dependencies* between components in the model. The test plan is executed in parallel on a cluster of test resources, leveraging hardware virtualization technology. The results from empirical studies show that incompatibilities between components were effectively detected for two large-scale scientific middleware systems we studied.

Related to this work, an approach to test software on heterogeneous field environments was studied in [3]. Instead of provisioning intended environments without modifying the physical state of test resources, they used available environments on the Grid as-is or modified physical resource

states, if needed, to provision the required environments. Based on combinatorial test design, Cohen et al. [1] investigated methods to test variability of software product lines. My approach differs by modeling inter-component relations in a graph and sampling configurations that cover relations between directly-dependent components.

## 2. PROPOSED APPROACH

The proposed approach [5, 6] includes a compatibility test process, configuration sampling and execution methods, and an automated tool to support this process. Current work is restricted to testing that components build properly with various configurations of other components on which they depend.

### 2.1 Compatibility Testing Process

**1. Model software configuration space:** Developers first need to define the configuration space that encodes ways in which the System Under Test (SUT) may be legitimately configured. To do this, dependency relationships between components are encoded as an acyclic directed graph called a *Component Dependency Graph (CDG)*. If a component is packaged in a well-known format (e.g., RPM), dependencies may be automatically extracted. The example CDG depicted in Figure 1 shows that the SUT component, called A, requires component D and one of B or C (captured via an XOR node represented by +). Components B and C require E; D requires F; and E and F require G. Dependencies may also be captured via other means such as a feature model or rule-based representation [2, 4].

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 two version identifiers and component C's version C<sub>2</sub> may only be built with E's versions E<sub>3</sub> and higher.

**2. Determine coverage criteria:** The specified model may encode a large number of ways in which components in the model may be legally configured, and developers must determine which parts of the space need to be examined. One possibility is to test *all* configurations, which is infeasible in many cases. Instead, developers may prefer more practical criteria that systematically sample the space.

The proposed approach is based on the observation that a successful component build and its behavior are most influenced by the components on which it *directly* depends. In

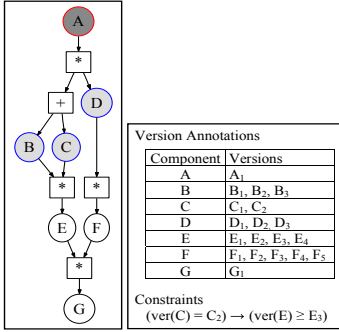


Figure 1: CDG and Annotation for Example System

this approach, a component  $c$  in a CDG directly depends on a set of other components when there exists a path from the node representing  $c$  to the node representing each component in the set, where no other component node is contained in the path. If a component version  $c_v$  is properly built with a version combination of components in the set,  $c_v$  with the combination is considered adequate for testing the build of other components relying on  $c_v$ .

This observation should be reliable, since in practice component developers usually describe only a list of components required for their component, and widely-used component build/deployment tools such as *Autoconf/Automake* and *yum* typically check the appropriateness of an environment to build a component by examining the existence or functionality provided by components on which the component to build directly depends.

Based on this observation, I have proposed a coverage criterion called *Direct Dependency (DD) coverage* that tests all direct dependencies of components in a CDG. Each direct dependency is a tuple  $t = (c_v, d)$  where  $c_v$  is a version  $v$  of a component  $c$ , and  $d$  is the dependency information used to build  $c_v$  – a version set of components on which  $c$  directly depends. For example,  $(A_1, \{B_1, D_1\})$  is one of 15 direct dependencies for the component **A** in the example in Figure 1.

**3. Produce configurations and test plan:** Given the model and coverage criteria, configurations satisfying the coverage are produced automatically. A configuration is a sequence of direct dependencies where component versions included in the direct dependencies need to be built, respecting their dependencies. For example, to test the compatibility of  $A_1$  with  $\{B_1, D_1\}$ ,  $B_1$  and  $D_1$  must be built in advance as specified in each of their direct dependencies.

Then, to reduce the number of components to build and to reuse the efforts to test identical partial configurations, the produced configurations are combined into a *prefix tree*, called the *test plan*. The direct dependencies in configurations are mapped into nodes in the tree and the work to build identical prefixes (sub-sequences) among configurations may be shared and reused for testing multiple configurations.

**4. Execute test plan:** A test plan is executed by distributing work defined for nodes in the plan to multiple machines. The work for a node is the sequence of direct dependencies encoded by the nodes in the path from the root to the node, and may be reused if it is shared by multiple configurations. Actual component builds are performed inside a virtual environment running on each resource, without contaminating persistent host machine state.

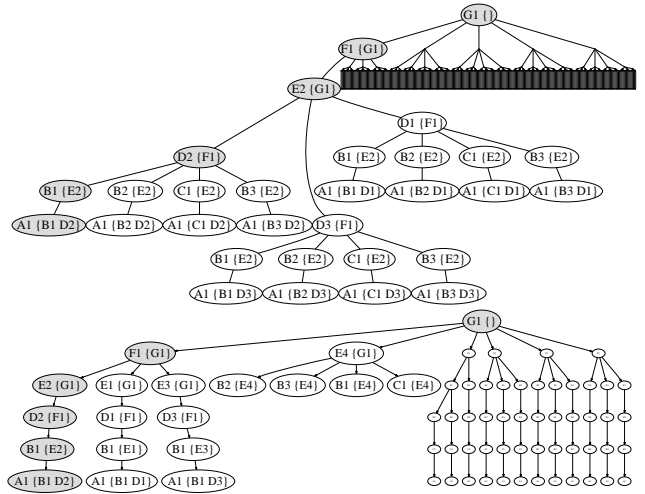


Figure 2: EX-plan (top) and DD-plan (bottom) for example model

The test plan can be modified dynamically if needed, so as not to lose test coverage. If building a component fails, it will prevent testing direct dependencies to build higher-level components depending on the component. In this case, new configurations are generated for the affected direct dependencies and added to the plan, to test the direct dependencies with low-level ones successfully built in an alternate way.

## 2.2 Producing Reduced Set of Configurations

For practical considerations, developers must sample the configuration space for a software system, instead of testing entire space. I have proposed a method to produce a reduced set of cost-effective configurations.

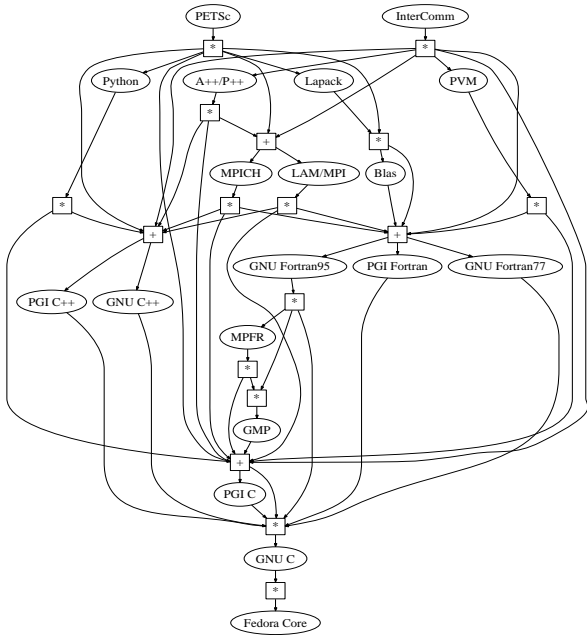
Based on the observation mentioned before, the proposed method produces a set of configurations where direct dependencies for all components in a CDG are contained in *at least* one configuration. Direct dependencies contained in a configuration are considered to be *covered* by the configuration, and the compatibility of component versions with other components can be obtained by testing the configuration. The method to produce configurations is briefly described below.

For an uncovered direct dependency  $(c_v, d)$  of each component in the CDG, a configuration initially has one element: the direct dependency  $(c_v, d)$  to cover. Then, we need to determine how to build the component versions contained in  $d$ . This is achieved by recursively selecting appropriate direct dependencies for the component versions and expanding the configuration with the selected direct dependencies, while traversing the CDG in depth-first order until the configuration under construction contains all necessary direct dependencies to examine the initial direct dependency.

As a result, a set of configurations is produced where each configuration is a set of direct dependencies, then a test plan is synthesized from the configurations as explained before. Figure 2 shows two test plans for the example model; one from the configurations produced exhaustively and the other from the configurations that cover all direct dependencies of the components. An example configuration contained in both plans is shaded in the figure.

## 2.3 Test Plan Execution

Executing a test plan means that for each node contained



**Figure 3: CDG for InterComm and PETSc**

in the plan the component version encoded by the node must be built on top of all the component versions it directly depends on. To do that, it is necessary to have a machine on which all component versions encoded by the ancestor nodes are built properly. Therefore, we define a *task* for a node as the sequence of direct dependencies encoded by the nodes in the path from the root of the plan to the node.

Actual *execution* of the task for a node uses a *virtual machine (VM)* environment, so as not to contaminate the persistent state of a physical test resource. VMware server is used as the hardware virtualization layer. Then, if the task execution is successful, which means that all component versions are built without any error, the VM state may be cached and *reused* to execute tasks for the descendant nodes. In that case, only additional components need to be built by reusing the VM state.

I have designed three plan execution strategies to maximize task parallelism and reuse of cached tasks: *parallel depth-first*, *parallel breadth-first* and *hybrid strategy*. For all strategies, I assume that a single server controls the plan execution and distributes tasks to multiple clients, and that each client has disk space available to store VMs (completed tasks) for reuse after executing its assigned tasks.

**Parallel Depth-First Strategy:** The parallel depth-first strategy is designed to maximize the reuse of locally cached tasks at each client. When a client completes executing a task for a node and subsequently requests a new task, the server first tries to assign a new task based on the previously executed task or based on the completed tasks (VMs) locally cached in the client machine. That is, it searches the plan to find an untested node in depth first order starting from the nodes corresponding to the locally available tasks. And, if there is no such node, the plan is searched from the root node. For the search, the nodes currently being executed by other clients, and their subtrees in the plan, are not searched.

**Parallel Breadth-First Strategy:** The parallel breadth-first strategy focuses on increasing the number of tasks being

executed in parallel by multiple clients. To achieve that, a node queue is maintained and the task for the first untested node in the queue is assigned to a requesting client. The server initializes the queue with nodes by pre-traversing the plan until the number of nodes in the queue exceeds the number of clients so that tasks for nodes in the queue are assigned immediately to the clients at the beginning of plan execution. If a task execution is successful, the server appends to the queue the child nodes of the node corresponding to the task. To reduce the time to execute a task, the server always finds the best cached task to initialize the state of the VM to execute the task and the cached task need to be transferred across the network, if needed.

**Hybrid Strategy:** The drawback of the depth-first strategy is that it may not fully achieve maximum task parallelism during the early stages of plan execution, since many clients could be idle waiting for tasks to become available. On the other hand, the breadth-first strategy has high task reuse cost due to many VM transfers across the network. The hybrid strategy is designed to balance both the locality of reused tasks and task parallelism, by combining the features of both strategies. As in the breadth-first strategy, a node queue is used to increase task parallelism during the early execution stage. And, to maximize the locality of reused tasks, this strategy assigns new task by searching the plan as does the depth-first strategy.

### 3. INITIAL RESULTS

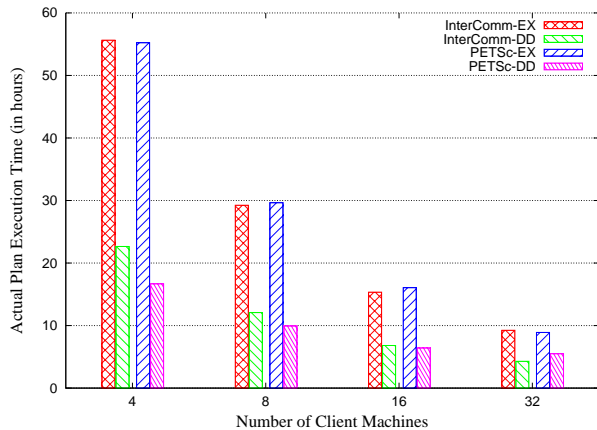
The proposed approach has been implemented as an automated tool with a client-server architecture, and applied to two software systems used in high-performance computing community: InterComm and PETSc. Currently, the application context is restricted to test component builds.

As shown in Figure 3, I modeled component dependencies to encode the configuration space for the two systems. The model consists of components (up to 4 versions each) necessary for the systems, including compilers for C (PGI C, GNU C), C++ (PGI C++, GNU C++) and Fortran (GNU Fortran95, PGI Fortran, GNU Fortran77), numerical libraries (Blas, Lapack, GMP, MPFR), data communication libraries (PVM, MPICH, LAM/MPI) and a parallel array management library (A++/P++). The bottom node represents an operating system (Fedora Core). Additional constraints are also specified. For example, only a single C++ compiler version can be used in a configuration.

For each system, I generated a test plan (EX-plan) from the configurations that covers the entire configuration space and another plan (DD-plan) from configurations that cover all direct dependencies. (EX-plan contains roughly 10 times more nodes than DD-plan for the systems.) The plans are executed using the plan execution strategies. I present results from both real executions and simulations.

#### Cost-effectiveness of testing direct dependencies:

The results in Figure 4 show that the DD-plan executions are up to 3 times faster than the EX-plan executions for the subject systems with 4, 8, 16 and 32 client machines. The speedup was not 10 (the plan size difference) due to many build failures occurring during plan execution. The execution times decrease by half as the number of test resources doubles, up to 16 machines. This means that additional resources are fully utilized to maximize the number



**Figure 4: InterComm and PETSc plan execution times with parallel depth-first strategy.**

of tasks executed in parallel. The achieved benefit with 32 machines were smaller, since many machines are idle due to component dependencies, and the overall cost for task reuse increases when tasks are spread across more machines.

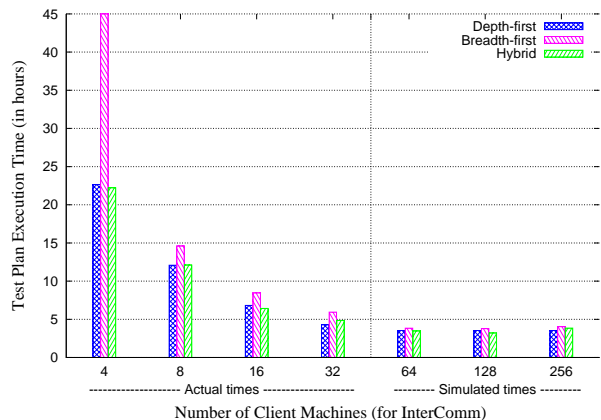
I also examined the potential loss of test effectiveness from using the DD-plan instead of the EX-plan. To do that, for all component build failures identified by the EX-plan, I noted the component and its version that failed to build, and its dependency information – i.e., a direct dependency of the component. Then, I checked whether building the component version failed in the same context in the DD-plan. For the InterComm example, all contexts of successful and failed builds were perfectly matched. For the PETSc example, due to a missing constraint, there were 8 mismatches out of 160 direct dependencies. For these cases, the DD-plan execution reported successes and the EX-plan reported both successes and failures for an identical direct dependency.

#### Tradeoffs between plan execution strategies:

Figure 5 shows the combined results from both actual and simulated plan executions with different strategies for InterComm. Since we only had 32 machines available for actual executions, for the cases with large number of machines, we performed simulations based on the information obtained from actual executions. The simulated execution times were, on average, about 18% less than the actual times for up to 32 clients. The results for PETSc were similar.

Although the breadth-first strategy is designed to reduce the number of idle clients throughout the plan execution, it performed worst for most cases, mainly due to many expensive VM transfers between test clients. Especially with 4 machines, the breadth-first strategy works very poorly. In that case, the task to be reused for executing a new task had already been replaced by another task in the client cache hosting the VM that performed the task, and as a result many components in the task had to be rebuilt.

Similar performance was observed with the parallel depth-first and hybrid strategy, since many build failures that occurred during plan execution negate the benefits of the hybrid strategy achieved by maximizing the number of tasks started early in the plan execution. In addition, little benefit is achieved with more than 32 machines for all strategies. In those cases, additional machines remained idle waiting for task assignment since all available tasks were already assigned to other machines. In another experiment performed



**Figure 5: InterComm DD-plan execution times, with different plan execution strategies.**

with the assumption of no failures during the plan execution, I observed that the hybrid strategy performs best and balances well both task reuse locality and task parallelism across all numbers of clients.

## 4. DISCUSSIONS AND ONGOING WORK

I have presented an approach to automate software compatibility testing and shown the results of applying the approach to scientific middleware systems.

In this approach, configurations to test compatibility between components are automatically generated, and experimental and simulation results show that those configurations can effectively detect incompatibilities between components and can be tested efficiently in parallel, utilizing available resources. Currently, I am investigating methods to prioritize produced configurations under time constraints, and I plan to adapt graph-based pair-wise test design to produce more cost-effective configuration sets, and also to extend the work to functional and performance testing.

## 5. REFERENCES

- [1] M. B. Cohen, M. B. Dwyer, and J. Shi. Coverage and adequacy in software product line testing. In *Proc. of the Workshop on Role of Software Architecture for Testing and Analysis*, pages 53–63, Jul. 2006.
- [2] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, Jan./Mar. 2005.
- [3] A. Duarte, G. Wagner, F. Brasileiro, and W. Cirne. Multi-environment software testing on the Grid. In *Proc. of the Workshop on Parallel and Distributed Systems: Testing and Debugging*, Jul. 2006.
- [4] T. Syrjänen. A rule-based formal model for software configuration. Technical Report A55, Helsinki University of Technology, Dec. 1999.
- [5] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Direct-dependency-based software compatibility testing. In *Proc. of the 22st Int’l Conf. on Automated Software Engineering*, Nov. 2007.
- [6] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Effective and scalable software compatibility testing. In *Proc. of the Int’l Symposium on Software Testing and Analysis*, Jul. 2008.