Effective and Scalable 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

Today's software systems are typically composed of multiple components, each with different versions. Software compatibility testing is a quality assurance task aimed at ensuring that multi-component based systems build and/or execute correctly across all their versions' combinations, or *configurations*. Because there are complex and changing interdependencies between components and their versions, and because there are such a large number of configurations, it is generally infeasible to test all potential configurations. Consequently, in practice, compatibility testing examines only a handful of default or popular configurations to detect problems; as a result costly errors can and do escape to the field.

This paper presents a new approach to compatibility testing, called Rachet. We formally model the entire configuration space for software systems and use the model to generate test plans to sample a portion of the space. In this paper, we test all *direct dependencies* between components and execute the test plan efficiently in parallel. We present empirical results obtained by applying our approach to two large-scale scientific middleware systems. The results show that for these systems Rachet scaled well and discovered incompatibilities between components, and that testing only direct dependences did not compromise 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

1. INTRODUCTION

When developers produce software, one of their top concerns is the compatibility of their software with real field environments that are equipped with different software components, including multiple compilers and third-party libraries. If the software is released with undetected incompatibilities, it can make users spend valuable time resolving incompatibilities, and it may also make it difficult to rationally manage support activities for the software.

Many tools and techniques have been created over the last decade to reduce potential incompatibilities between software components, and these advances are now embodied in technologies such as interconnection standards, configuration management tools, service-oriented architectures and middleware frameworks. Despite these advances, it is still difficult to guarantee the compatibility of the components in a software system with the expected field environments, for several reasons.

First, field environments for software system can be extremely heterogeneous. The expected field environments for build and/or execute the developed software may have multiple inter-operable components, each with multiple versions, and software system can depend on all those components, so that in many cases it is infeasible to test all possible field environments. Second, the individual components and the dependencies between them can change without notice, especially if components are developed and maintained by separate groups of developers. Finally, developers do not want to restrict their potential user base by forcing users to limit their environments to only a set of supported (tested in-house) environments.

Developers perform *compatibility testing* [7] to ensure that a software system behaves (builds and functions) properly across a broad range of heterogeneous field environments. Compatibility testing involves selecting a set of *configurations* (field environments), where each configuration is an ensemble of component versions that respects known dependencies. However, as described above, the large number of possible configurations and the lack of automated testing support have limited developers to performing compatibility testing on a set of popular configurations [16], or on a set of configurations physically realized in field environments available over a network of machines [6]. For example, InterComm, one of our example applications, has been extensively tested in only three configurations, where each configuration is a development environment for a different operating system. This implies that often the software system is released with nearly all of its possible configurations untested. So costly errors can and do escape to the field.

In the previous short paper [17], we discussed the Rachet process to perform software compatibility testing, and presented initial empirical results for the InterComm scientific

ISSTA'08, July 20–24, 2008, Seattle, Washington, USA.

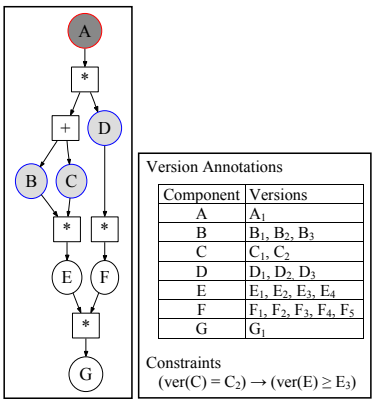

Figure 1: An Example ACDM

middleware library [12] that incompatibilities between components can be effectively detected by Rachet. In this paper, we extend our previous work and describe in detail new algorithms and strategies to improve the efficiency and effectiveness of compatibility testing. To validate our approach, we have performed experiments and simulations on two widely-used middle libraries from the field of high-performance computing. For this paper, we restrict the compatibility testing to the build process (i.e. compilation and deployment) of a component with other components on which it depends, but we will extend the work to functional and performance testing in future work.

Our approach makes several contributions, including: (1) a formalism for modeling software system's potential configuration space precisely; (2) algorithms for producing exhaustive and sampled test configurations to detect incompatibilities between components; (3) strategies to test configurations systematically in parallel, dealing with test failures dynamically to allow greater coverage; (4) an implementation that realizes the Rachet process, utilizing multiple computational resources that employ platform virtualization to enable the results from building lower-level components to be reused in building multiple higher-level components; and (5) results from empirical studies showing that the Rachet approach enables fast and effective compatibility testing across a large configuration space.

The rest of the paper is organized as follows. We explain a high-level overview of the steps needed to perform compatibility testing in Section 2. In Section 3 we describe a model to represent the software configuration space. Our approach to generate effective test configurations is presented in Section 4. Section 5 presents strategies to test the configurations, utilizing a large set of test resources. Section 6 outlines the Rachet architecture and presents results from empirical studies. Section 7 describes related work and in Section 8 we conclude with a brief discussion and future work.

2. Rachet PROCESS OVERVIEW

This section provides a high-level overview of the steps needed to perform compatibility testing for a given software under test (SUT) using Rachet.

1. Model software configuration space: To define the configuration space, i.e., the ways in which the SUT may be legitimately configured, developers identify the components needed to build the SUT. This information can often be obtained, at least in part, from the component providers. Dependency relationships between components are then encoded as a directed acyclic graph called *Component Dependency Graph (CDG)* and constraints between components are described as *Annotations* of the graph; together they form the model called *Annotated Component Dependency Model (ACDM)*. Relationships between components may also be encoded using other formalisms such as feature-based [4] or rule-based models [13]. The example CDG depicted in Figure 1 shows dependencies for a SUT component called `A`. The figure shows that `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`. Additional information such as version identifiers for components is also specified as annotations to the CDG. We formally describe our model in Section 3.

2. Determine coverage criteria: The above model encodes the configuration space for the SUT. For non-trivial software, this space can be quite large. Developers must determine which part of the space need to be tested for compatibility testing. For example, they may decide to test configurations exhaustively, which is often infeasible. Instead, they may choose more practical criteria that *systematically cover* the space. In this paper, we propose a coverage criterion that tests all combinations of each component with other components on which it *directly depends*; the definition and rationale behind this criterion is further explained in Section 3.

3. Produce test configurations and test plan: Given the model and coverage criteria, Rachet produces test configurations automatically; each configuration describes a set of components to build and dependency information used to build the components. Then, a test plan is synthesized from the configurations, which specifies the schedule to build components in the configurations.

4. Execute test plan: Rachet chooses tasks (sub configurations) in the plan and distributes them to multiple execution nodes according to a plan execution strategy described by the developers. If the goal is to determine whether a component can be built without any error on top of other components on which it depends, then a set of instructions to build each component should be specified. In addition, Rachet applies contingency plan to handle task failures dynamically. If testing a component in a task fails, it may prevent testing other components in the plan depending on it. In this case, Rachet dynamically modifies test plan so as not to lose test coverage, by creating additional configurations that try to build the components in different ways.

3. CONFIGURATION SPACE MODEL

Components, their versions, inter-component dependencies, and constraints define the configuration space of an SUT. The ACDM models the configuration space with two components, a CDG and a set of annotations (Ann). A CDG has two types of nodes – component nodes and relation nodes; directed edges represent architectural dependencies between them. For example, Figure 1 depicts an SUT `A` that requires components `B` and `D` or `C` and `D`, each of which depend in turn on other components. As shown in the figure, inter-component dependencies are captured by relation nodes that are labeled either "$*$" or "$+$", which are interpreted respectively as applying a logical AND or XOR over the relation node's outgoing edges.

ACDM annotations provide additional information about components in the CDG. The first set of annotations for this example is an ordered list of version identifiers for each component. Each identifier represents a unique version of the corresponding component. In Figure 1, component `B` has three version identifiers: $\texttt{B}_1$, $\texttt{B}_2$ and $\texttt{B}_3$.

Version-specific constraints often exist between various components in a system. For example, in Figure 1 component `C` has two versions and it depends on component `E`, which has 4 versions. Lets assume that component `C's` version $\texttt{C}_2$ may only be compiled using `E`'s versions $\texttt{E}_3$ and higher. This "constraint" is written in first order logic and appears as $(\texttt{ver(C)} = \texttt{C}_2) \rightarrow (\texttt{ver(E)} \geq \texttt{E}_3)$. Global constraints may be defined over entire configurations. For instance, in the case study in Section 6, we require all components depending on a C++ compiler to use the same version of C++ compiler in any single configuration.

We now formally define the ACDM:

DEFINITION 1. *An ACDM is a pair (CDG, Ann), where CDG is a directed acyclic graph and Ann is a set of annotations.*

DEFINITION 2. *A CDG (Component Dependency Graph) is a pair (V, E), where: (1) $V = C \cup R$. C is a set of labeled component nodes. Component node labels are mapped 1-1 to components required to test the SUT. R is a set of relation nodes whose labels come from the set $\{\text{“}*\text{”} | \text{“}+\text{”}\}$. Relation nodes are interpreted as applying a logical function, AND or XOR, across their outgoing edges, and (2) E is a set of dependency edges, with each edge connecting two nodes. Valid edges are constrained such that no two component nodes are connected by an edge: $E = \{(u,v)|u \in C, v \in R\} \cup \{(u,v)|u \in R, v \in R\} \cup \{(u,v)|u \in R, v \in C\}$. This enables the relationships between components defined solely by relation nodes.*

*Furthermore, valid CDGs obey the following properties: (i) There is a single distinguished component node with no incoming edges called **top**. Typically **top** represents the SUT. (ii) There is a single distinguished component node with no outgoing edges called **bottom**. This component is not dependent on any other component. (The bottom node may represent an operating system, but that is not required.) (iii) All other component nodes, $v \in \{C/\{\textbf{top,bottom}\}\}$, have exactly one incoming edge and one outgoing edge.*

DEFINITION 3. *The annotation set, Ann used in this paper contains two parts: (i) For each component $c \in C$, a set that defines the range of elements (versions) over which c may be instantiated. (ii) A set of constraints between components and over configurations. The constraints are specified using boolean operators ($\vee$, $\wedge$, $\rightarrow$, $\neg$) and relational operators ($\leq$, $\geq$, $==$, $<$, $>$) between component versions; the version numbers are used to evaluate these expressions.*

Except for the bottom node, all other components in a CDG depend on functionalities provided by other components on any path from the node encoding the component to the bottom node. However, many common build tools (e.g., GNU Autoconf and Automake [15]) assume that a successful build of the component is influenced by other components on which it *directly depends*. Hence, they check only for the functionalities provided by components on which the component to build depends, by generating and testing a simple program during the build process. Definition 4 defines a set of components on which a component *directly depends*.

DEFINITION 4. *In a CDG, a component c directly depends on a set of components if there exists a path that does not contain any other component node, from the component node for c to each component node for the components in the set.*

From this definition, component `A` in the previous example directly depends on `B`, `C` and `D` although it depends on functionalities provided by the component `B` through `G`. Similarly, `B` and `C` directly depends on `E`.

We annotate each component in a CDG with a set of relations called *direct dependencies*, between each version of the component and versions of other components on which it directly depends. A direct dependency is a tuple $t = (c_v, d)$ where c_v is a version v of a component c. d is the dependency information used to build c_v and is a set of versions of components on which c directly depends. When multiple relation nodes lie on a path between a component and other components on which it directly depends, we take into account the semantics of the nodes by applying set operations recursively; Union for XOR nodes and Cartesian product for AND nodes. For example, $(\texttt{A}_1,\{\texttt{B}_1,\texttt{D}_1\})$ is one of 15 direct dependencies for the component `A` in the previous example.

As mentioned, we restrict the application context for compatibility testing to error-free build of components.[1] In this context, testing a direct dependency (c_v, d) is to examine the build-compatibility of c_v with component versions in d. However, to build c_v with d, in advance we need to build each component version in d as specified by one of the direct dependencies for the component. This leads to defining a *configuration* as a partially ordered set of direct dependencies. In this paper, we totally order a configuration in reverse topological order, starting from the bottom node. For example, a configuration to test the direct dependency $(\texttt{A}_1,\{\texttt{B}_1,\texttt{D}_1\})$ above is: $((\texttt{G}_1,\emptyset)$, $(\texttt{E}_3,\{\texttt{G}_1\})$, $(\texttt{F}_2,\{\texttt{G}_1\})$, $(\texttt{B}_1,\{\texttt{E}_3\})$, $(\texttt{D}_1,\{\texttt{F}_2\})$, $(\texttt{A}_1,\{\texttt{B}_1,\texttt{D}_1\}))$. In the next section, we describe methods to generate configurations.

4. PRODUCING CONFIGURATIONS

Although the configuration space for a software system can be tested exhaustively, that may be very expensive for a large system. For practical considerations, developers must sample the space. In this section, we describe two methods to generate configurations to either test the entire space or to sample the space and cover all direct dependencies.

4.1 Exhaustive-Cover Configurations

The most straightforward way to build-test the range of configurations in which the SUT is build-compatible is to build it on the exhaustive set of possible configurations. To compute an exhaustive configuration set, we start from the *bottom node* of the CDG (one that does not depend on any other components), and for each node type, do the following:
•*Component node:* compute new configuration set by extending each configuration in the configuration set of its child node (a relation node) with proper direct dependencies of the component. That is, for each direct dependency (c_v, d) of the component, identify configurations from the configuration set of the child node, where each configuration contains all matched component versions specified in d.

[1]In many Unix-based operating systems, building a component commonly includes three steps – *configuring, compiling and deploying* the component.

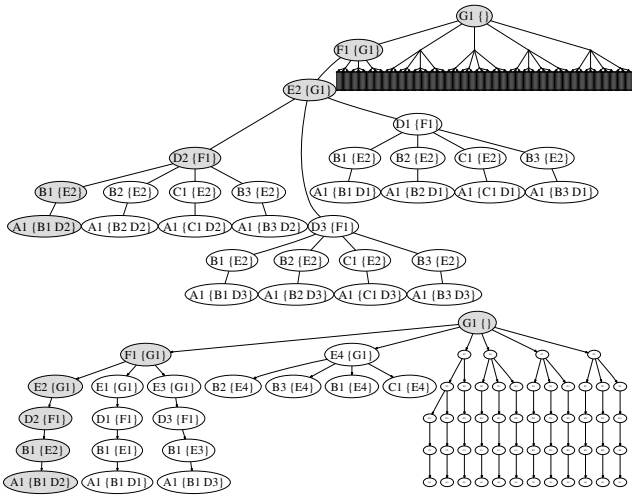

Figure 2: EX-plan (top) and DD-plan (bottom) for example model

Then, extend the configurations with (c_v, d). For the *bottom node*, return the bottom component's versions, each is paired with the empty set, as it has no dependences.

•*AND node:* compute all combinations of configurations taken from each configuration set of its child nodes, then combine the configurations in each combination. We enforce two rules implicitly in computing the combinations of configurations for an AND node. First, only one version for each component is allowed in a configuration. That is, we do not allow to combine configurations when each contains a direct dependency for the component but with different version. Second, we allow a single way to build a component in a configuration. This means that we need to check whether dependency information in direct dependencies for a component are same across the configurations to combine.

•*XOR node:* the result set is simply the union of the configuration sets of its child nodes.

Even for the simple CDG in the Figure 1, the number of test configurations in the exhaustive set for component A is 270. Since a CDG for a real application can be more complex and contain many more components, as shown in Figure 4, the number of configurations in exhaustive cover may be large. Taking into account the potentially long time required to build each complete configuration, for many CDGs it would be infeasible to test all possible configurations.

4.2 Direct Dependency-Cover Configurations

Instead of testing all configurations exhaustively, we propose an approach to generate a reduced set of test configurations, while still maintaining test effectiveness. Our approach is based on the observation that a successful component build is most influenced by the components on which it *directly* depends, as described in Definition 4. We now describe a method to produce a set of configurations where direct dependencies for all components in a given CDG is contained in *at least* one configuration. We say that direct dependencies contained in a configuration are *covered* by the configuration and the information on build-compatibility of component versions encoded by each direct dependency can be obtained by testing the configuration.

For a yet-uncovered direct dependency (c_v, d), the configuration under construction initially has only one element, i.e., the direct dependency itself. Next, we need to determine how to build component versions in d, since the component version c_v must built on top of the component versions. This is achieved by recursively selecting an appropriate direct dependency for each component version in d where the selected direct dependency encodes how to build it; the recursion ends at the bottom node. During the selection process, the rules explained in the previous section are applied and constraint are checked. We apply this selection process for the component nodes in topological order starting at the *top* component, since more direct dependencies may be covered from multiple direct dependency sets when applied to a direct dependency of components close to the *top* component in the CDG, compared to those closer to a *bottom*.

In earlier work, when multiple direct dependencies were available for a component version, we selected the first-fit direct dependency [17]. This made multiple configurations repeatedly use the same direct dependency for a component version, although other direct dependencies for the component version may also have been covered by the configurations. This increased the number of configurations unnecessarily to cover uncovered direct dependencies. In this paper, we use a heuristic that selects an *uncovered direct dependency first*. The example system has 58 direct dependencies; applying the new heuristic we obtain only 19 configurations containing 102 components to build; this is much smaller compared to the 40 configurations containing 165 components obtained with first-fit direct dependency selection.

4.3 Test Plan Synthesis

Generated configurations may be tested one at a time by building each component in each configuration on a machine according to a build order. However, the total number of component versions that must be built may be reduced by utilizing the fact that multiple configurations may contain identical direct dependency sequence. For example, all configurations contain the same first direct dependency that decides how to build an operating system if only one version of operating system is used in the model.

To reduce the number of components to build, we merge the configurations into a single *prefix tree*. Initially, the tree contains only one node, the root of the tree. For each configuration, nodes representing direct dependencies in the configuration are in order added to the prefix tree, called a *test plan*. For a node n in the tree, we use the notation $n.c_v$ and $n.d$ to specify the component version and dependency information for the component. Since common prefix are shared among configurations in the plan, the total number of component versions to build is less than the sum of the component versions in the configurations. Figure 2 shows two test plans, one from the configurations produced exhaustively and the other from the configurations that cover all direct dependencies of the components in the previous example. An example configuration contained in both plans are shaded in the figure. The exhaustive test plan contains 270 configurations, consisting of 626 component versions to build, which is 62% less compared to 1620, the sum of component versions in produced configurations. In the next section, we describe our strategies to execute a test plan.

5. TEST PLAN EXECUTION

The test plan created by the process described in Section 4 may be executed in several ways. Test plan execution visits all nodes contained in the plan, and when a node is visited

we test the build compatibility of the direct dependency it encodes. In this section, we describe three test plan execution strategies that attempt to maximize both parallelism and reuse of cached tasks. We also describe how to handle component build failures during the plan execution.

5.1 Plan Execution Strategies

Executing a test plan means that for each node we must build c_v on top of all the component versions contained in d. However, to test a node n, we need a machine on which all component versions encoded by n's ancestor nodes have been built, since the component versions contained in $n.d$ may also depend on other lower-level components. Therefore, the component versions in the sequence of direct dependencies encoded by the nodes in the path from the root of the plan to the node n must be built before testing the build compatibility of the direct dependency encoded by n. We call this sequence the *task* for the node n.

When we *execute* the task for n, all component versions in the task must be built in a proper order, respecting all dependencies. We use a *virtual machine* (VM) environment, called VMWare, for the builds so as not to contaminate the persistent state of a physical test resource (machine). Then, if the task execution is successful, which means that all component versions were built without any error, the modified machine state has the correct state for the task and the machine may be *reused* to execute tasks for n's descendant nodes. When we execute those tasks, we need only to build additional components by reusing the VM state, which is encapsulated in the file system of the physical machine hosting the VM. For all test plan execution algorithms we describe, we assume that a single test server controls the plan execution and distributes tasks to multiple test clients. We also assume that each client (a physical machine) has disk space available to store VMs (completed tasks) for reuse.

Parallel Depth-First Strategy: The parallel depth-first strategy is designed to maximize the reuse of locally cached tasks at each client during the plan execution. When a client completes executing a task for a node n and subsequently requests a new task, the server assigns a task according to following rules, attempting to maximize cached task reuse.

First, if the node n is a non-leaf node in the plan, the task for one of n's unassigned child nodes is chosen as the next task for the client. The client will then reuse the VM state from its previously executed task, so only have to build one additional component (the one specified by the last direct dependency in the new task). This is typically the least expensive way to execute a new task.

Second, if the node n is a leaf node, tasks already stored in the cache space of the client are utilized to assign new task. Starting from the node for the most recently cached task, the algorithm searches for an unassigned descendant node in depth-first order. The nodes currently being executed by other clients, and their subtrees in the plan, are not visited by the search. In this case, the test client must build the *difference* between the assigned task and the reused task.

Finally, if the algorithm cannot find an unassigned node using the first or second rule, the plan is searched in depth-first order from the root node. As for the second rule, the nodes currently being executed, and their subtrees, are not visited. In this case, to reduce the time to execute the assigned task, the test server finds the best cached task for the assigned task (i.e. the one with the longest matching prefix), so the VM for the cached task must be transferred across the network from the client that produced the cached task, which can take a significant amount of time (a cached VM can be large, up to 1GB or more). The difference between the assigned task and the cached task must then be built.

For the depth-first strategy, the decision to cache a task that has just been executed is based on the number of children its node has in the plan. If the node has two or more children, the task may be reused to execute tasks for the children, so the test server requests the client to cache the task. However, if the node has only one child, the task for the child node is assigned to the same client by the first rule, so there is no reason to cache the task.

Since the depth-first strategy tries to utilize locally cached tasks, the number of locally reused tasks is maximized, minimizing the number of tasks that require task transfers between clients. However, the cost to build the components in a task will be high if the difference between an assigned task and a locally cached task is large. In addition, when a large number of test clients are available and the test plan does not have many nodes near the root of the plan, many clients could be idle during the early stage of plan execution, waiting for enough tasks to become available.

Parallel Breadth-First Strategy: The parallel breadth-first strategy focuses on increasing task parallelism. This strategy tries to maximize the number of tasks being executed simultaneously, and secondarily tries to maximize the reuse of locally cached tasks. To assign tasks in breadth-first order, the server maintains a priority queue of plan nodes ordered according to their depth in the plan.

At the initialization step, the algorithm initializes the priority queue by traversing the plan in breadth-first order, adding nodes to the queue until the number of nodes exceeds the number of test clients. When a leaf node in the plan is traversed, it remains in the queue. On the other hand, when a non-leaf node is traversed, it is removed from the queue and instead its child nodes are added to the queue. That is, we increase the number of tasks that can be executed in parallel by assigning tasks for the child nodes.

When a task is requested by a client, the test server assigns the first unassigned task in the queue. Then, if a task is executed by the client successfully, the algorithm locates the node corresponding to the task in the queue, and appends the child nodes to the queue. To reduce the time to execute a task, the test server always finds the best cached task to initialize the state of the VM to execute the task, although the cost to transfer the VM across the network may be high.

Unlike the depth-first strategy, for the breadth-first strategy a completed task is cached if its corresponding node in the plan is a non-leaf node. The rationale behind this choice is that in many cases the tasks for the child nodes will not be assigned to the same client. This strategy will keep all clients busy as long as there are unassigned nodes in the queue throughout the plan execution. Therefore, we expect high-level of parallelism. However, we also expect increased network cost compared to the depth-first strategy, because of transferring many cached tasks across the network.

Hybrid Strategy: We have described costs and benefits of the depth-first and breadth-first strategy. Although the depth-first strategy tries to maximize the locality of reused

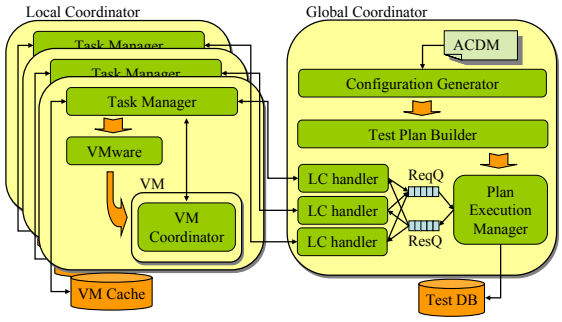

Figure 3: Rachet Software Architecture

tasks, during the early stage of plan execution it may not maximize the parallelism that could be obtained by executing tasks on all available clients. On the other hand, the breadth-first strategy may achieve high level of parallelism, but may also increase the network cost to execute tasks.

The hybrid strategy is designed to balance both the locality of reused tasks and task parallelism throughout plan execution, by combining the features of both strategies. As in the breadth-first strategy, a priority queue of plan nodes is created by traversing the test plan, and is used to increase task parallelism during the early execution stages of the plan. That is, for the initial task requests from test clients, tasks for nodes in the queue are assigned to *all* available clients immediately at the beginning of plan execution. To maximize locality for reused tasks, the first and second rules for the depth-first strategy are subsequently applied to assign tasks to requesting clients. If both rules fail to find an unassigned node, the test plan is traversed in breadth-first order from the root node to find an unassigned node. This is based on the heuristic that a node closer to the root node will likely have a larger subtree beneath it than nodes deeper in the tree, which will lead to more work being made available for a client reusing locally cached tasks.

5.2 Dynamic Failure Handling

If building a direct dependency encoded by a node n fails, $n.c_v$ could not be built correctly on top of the component versions contained in $n.d$. We say that $n.c_v$ is *incompatible* with $n.d$, and use the failure information to guide further test plan execution. A build failure for a direct dependency for node n also prevents testing of all direct dependencies represented by the nodes in n's subtree. This is because we need a VM on which all direct dependencies of the ancestor nodes have been built to test those direct dependencies. However, the failure does not imply failure for *all* direct dependencies affected by the failure. In this situation, instead of regarding the direct dependencies as *not able to be tested*, we adjust the test plan dynamically to test the direct dependencies in alternate ways, if possible, by producing additional configurations to test them, and merging the new configurations into the test plan. This enables the test plan execution algorithms to maximize direct dependency test coverage.

Since one direct dependency can be used in multiple configurations, it can appear as multiple nodes in different branches of a test plan. If one of the (identical) test plan nodes fails, we expect the others to also fail, since direct dependency testing assumes that the success of a component build is mainly determined by the components on which it directly depends. Thus, the test plan nodes affected by a build failure are not confined to the descendant nodes in the subtree of the failed node, but also include all descendant nodes in the subtrees of the nodes encoding the same direct dependency. Therefore the contingency planning algorithm must generate new configurations to cover all the affected direct dependencies.

To reduce the number of newly generated configurations, we apply the algorithm described in Section 4.2 for the direct dependencies represented by the descendant nodes under the subtrees in a post-order tree traversal. As configurations are produced for components in the CDG in topological order, we expect that a direct dependency represented by a node will be covered while generating configurations for the direct dependencies represented by its descendant nodes.

6. EMPIRICAL STUDIES

We have developed an automated test infrastructure that supports the Rachet process, and have performed empirical studies with two scientific middleware libraries that are widely used in the high-performance computing community. In this section, we describe the overall architecture of the Rachet infrastructure and present results from the studies.

6.1 System Architecture

The Rachet infrastructure is designed in a client/server architecture, as illustrated in Figure 3, utilizing platform virtualization technology.

Global Coordinator (GC): The GC is the centralized test manager that directs overall test progress, interacting with multiple clients. It first generates configurations that satisfy the desired coverage criteria (e.g., direct dependencies) and also produces a test plan from the configurations, using the algorithms described in Section 4. Then, the GC dynamically controls test plan execution by dispatching tasks and the ancillary information necessary to execute the tasks to multiple clients, according to one of the test plan execution strategies described in Section 5.

The GC contains a *testmanager* thread and a set of *lchandler* threads, one for each client machine. The *testmanager* is responsible for creating configurations and a test plan. During test execution, the testmanager satisfies requests from clients and inserts test results into a database. When a client first requests a task, the GC creates an *lchandler* thread for the client and that *lchandler* is responsible for all communication with the client.

Local Coordinator (LC): The LC controls task execution in a test client. One LC runs on each test machine and interacts with the GC to receive information on the tasks it must execute and also to report execution results.

As describe previously, task execution in the current Rachet design and implementation means *building* the components in a task, taking into account the dependency information needed to build each component. To do the builds, the LC employs hardware virtualization technology. The components are built within a *virtual machine (VM)*, which provides a virtualized hardware layer. This design is advantageous since the persistent state of the test machine is never changed, so a large number of tasks can be executed on a limited number of physical test machines. The Rachet implementation currently uses *VMware Server* as its virtualization technology, since it handles virtual machines reliably and also provides a set of well-defined APIs to control the

VM. A key feature of VMware Server is that the complete state of a VM is stored as files on the disk of the test machine, so can be saved and later reused (i.e. the VM can be stopped and copied, and the original VM and the copy can be restarted independently).

Virtual Machine Coordinator (VMC): The VMC is responsible for the actual component build process in a VM. When a VM is started by the LC, the VMC is automatically installed in the VM and started by the LC. The VMC then interacts with the LC to receive the build commands for the task assigned to the LC. The instructions to build each component are translated into appropriate *shell* commands by the VMC and executed in the VM to build the required components.

Interactions among GC, LC and VMC: The various coordinators in the Rachet system interact with each other to execute a test plan as follows:

1. Prepare test: The GC produces configurations and builds a test plan. Then, it listens for LC requests.

2. Assign a test case: When a LC requests a new task, the GC selects a task from the plan based on the desired plan execution strategy, also employing ancillary information, such as the VM provisioning method for the task.

3. Provision a VM: Each LC provisions a VM chosen to execute the assigned task. If a locally cached VM is to be reused, the cached VM is uncompressed into a directory. However, if a VM stored in a remote machine is chosen, the LC fetches the VM over the network and decompresses it.

4. Establish a communication channel with the VMC: The LC starts the provisioned VM, places and launches the VMC in the VM. The VMC automatically connects to the LC and establishes a communication channel.

5. Build components: The LC sends instructions to the VMC to build the components contained in the task, and the VMC translates the instructions into a series of *shell* commands and executes them on the VM.

6. Report Results and Cache VM: The LC reports the test result to the GC. The GC stores the result and uses it to guide further plan execution. If the task is executed successfully, and if the GC has requested the LC to cache the task, the LC requests a unique cache identifier from the GC and registers the cached task with the GC. The VM is compressed into a file and stored in the LC's local cache.

6.2 Subject Systems

We have applied the Rachet process to two widely used software libraries used to build high-performance computing applications: InterComm[2] and PETSc[3].

InterComm is a middleware library that supports coupled scientific simulations by redistributing data in parallel between data structures managed by multiple parallel programs [12]. To provide this functionality, InterComm relies on several components including multiple C, C++ and Fortran compilers, parallel data communication libraries, a process management library and a structured data management library. Each component has multiple versions and there are complex dependencies and constraints between the components and their versions.

[2] *http://www.cs.umd.edu/projects/hpsl/chaos/ResearchAreas/ic*

[3] *http://www-unix.mcs.anl.gov/petsc/petsc-as*

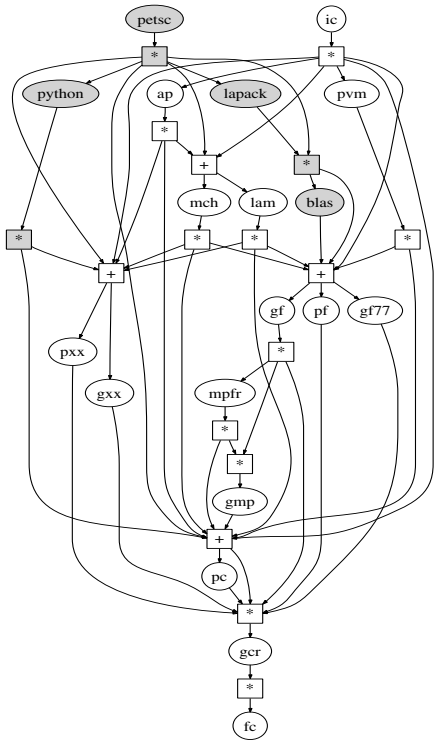

Figure 4: CDG for InterComm and PETSc

PETSc (Portable, Extensible Toolkit for Scientific computation) [2] is a collection of data structures and interfaces used to develop scalable high-end scientific applications. Similar to InterComm, PETSc is designed to work on many Unix-based operating systems and depends on multiple compilers and parallel data communication libraries to provide interfaces and implementations for serial and parallel applications. To enhance the performance of application developed using PETSc, PETSc also relies on third-party numerical libraries such as BLAS [5] and LAPACK [1], and uses Python as deployment driver.

6.3 Experimental Setup

To perform compatibility testing for the subject systems, we first modeled component dependencies, working directly with the InterComm developers and carefully inspecting documentation provided by the PETSc developers. In Figure 4, we show the component dependencies captured for InterComm and PETSc in a single CDG. The nodes specific to PETSc are shaded in the figure. Version annotations for the components used in the CDG are depicted in Table 1.

In addition to component versions, following constraints are specified and must be satisfied by each configuration. First, if multiple GNU compilers are used (gcr, gxx, gf and gf77) in a configuration, they must have the same version identifier. Second, only a single MPI component (i.e., *lam* or *mch*) can be used in a configuration. Third, only one C++ compiler, and only one of its versions can be used in a configuration. Fourth, if both a C and a C++ compiler are used in a configuration, they must be developed by the same vendor. For PETSc, we applied one additional constraint: compilers from the *same* vendor must be used to build the PETSc or MPI component. With these constraints, we obtained 302 and 160 direct dependencies for the components contained in the InterComm and PETSc models.

Comp.	Version	Description
petsc	2.2.0	PETSc, the SUT
ic	1.5	InterComm, the SUT
python	2.3.6, 2.5.1	Dynamic OOP language
blas	1.0	Basic linear algebra subprograms
lapack	2.0, 3.1.1	A library for linear algebra operations
ap	0.7.9	High-level array management library
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: Component Version Annotations for InterComm and PETSc

For each subject system, we generated two test plans. Table 2 summarizes the number of produced configurations, and the number of components contained in those configurations and in the test plan. The first test plan, called EX-plan, was generated using the exhaustive coverage criteria, and the other test plan, called DD-plan, only covers all direct dependencies identified for the components in a model. For example, the PETSc EX-plan has 1,184 configurations, containing 14,336 components to be built. However, the number of components in the final test plan is only 3,493, since configurations are merged to produce the test plan.

We first conducted experiments to measure the costs and benefits of DD-cover compared to EX-cover, and also examined the behavior of Rachet as the overall system scales. To do that, we executed both the EX-plan and DD-plans with 4, 8, 16 and 32 client machines, using the parallel depth-first plan execution strategy. To compare the various test plan execution strategies, we also executed the DD-plans for both subject systems using the parallel breadth-first and hybrid strategies with the same range of client machines.

For all experiments, we ran the GC on a machine with Pentium 4 2.4GHz CPU and 512MB memory, running Red Hat Linux 2.4.21-53.EL, all LCs run on Pentium 4 2.8GHz Dual-CPU machines with 1GB memory, all running Red Hat Enterprise Linux version 2.6.9-11. All machines were connected via Fast Ethernet. One LC runs on each machine, and each LC runs one VM at a time to execute tasks. The number of entries in the VM cache for each LC is set to 8, because in previous work [17] we observed little benefit from more cache entries for the InterComm example, and also because test plans for PETSc are smaller than test plans for InterComm in this scenario. In addition to these experiments on the real system, we ran simulations using our event-based simulator that mimics the behavior of the key Rachet components, described in Section 6.1, to better understand the performance characteristics of the Rachet on larger sets of resources than we were able to use for the real experiment (both because of limited resource availability and the time required to perform experiments).

System	Type	Cfgs	Comp_{cfgs}	Comp_{plan}
InterComm	Ex-Cover	3552	39840	9919
InterComm	DD-Cover	158	1642	677
PETSc	Ex-Cover	1184	14336	3493
PETSc	DD-Cover	90	913	309

Table 2: Test Plan Statistics

6.4 Cost-benefit Assessment

As shown in Table 2, the EX-plans for both systems have a large number of configurations compared to the DD-plans. Since it takes up to 3 hours to build a configuration for either InterComm or PETSc, it requires about 10,600 and 470 CPU hours to execute the InterComm EX-plan or DD-plan, respectively, and 3,500 or 270 CPU hours for the corresponding PETSc plans. With a naive plan execution strategy where each configuration is always built from scratch, with 8 machines it would still take 1,325 or 438 hours, respectively for the EX-plans with perfect speedup, and 59 or 34 hours for the DD-plans. However, since our plan execution strategies reuse build effort across configurations, the plan execution times for both plans are expected to be much smaller than times with the naive execution strategy. In our experiments, execution times were further shortened due to many build failures.

The cost savings obtained by executing the DD-plans are shown in Figure 5. With 8 machines, the InterComm EX-plan took about 29 hours with the parallel depth-first strategy, during which 461 of the 9,919 component builds (the number of nodes in the plan) were successful and 687 failed. All other builds could not be tested due to observed failures. For the PETSc EX-plan, about 29 hours were needed, during which we observed 724 build successes and 407 build failures for the 3,493 components in the plan, with the rest not able to be tested. Compared to the EX-plans, the InterComm DD-plan took 12 hours with 275 successful component builds, and the PETSc DD-plan took 10 hours with 216 successful builds. In our experiments, the execution times for the EX-plans took only 2.5 – 3 times more than those for the DD-plans, because many build failures occurred during plan execution, especially for the components close to the bottom node in the CDGs. Note that the difference in execution times between the EX-plans and DD-plans decreases as more clients are used, since the Rachet system always tries to best utilize the machines for plan execution and therefore a larger plan can benefit more when many clients are available.

The results show that Rachet was able to achieve large performance benefits by testing only configurations covering direct dependencies, and also was able to execute the test plans efficiently using the depth-first execution strategy. However, we also need to examine the potential loss of test effectiveness from using the the DD-plan, that only samples a subset of the configurations that are tested by the EX-plan. To do that, for all component build failures identified by the EX-plan, we examined the component and its version that failed to build, and its dependency information. We then checked whether building the same component version failed in the same context in the DD-plan.

We found that each component build failure in the InterComm EX-plan exactly maps to a corresponding failure in the DD-Plan. However, for the PETSc EX-plan, we observed 8 instances where a PETSc component build failure or success depended on the exact compilers used to build

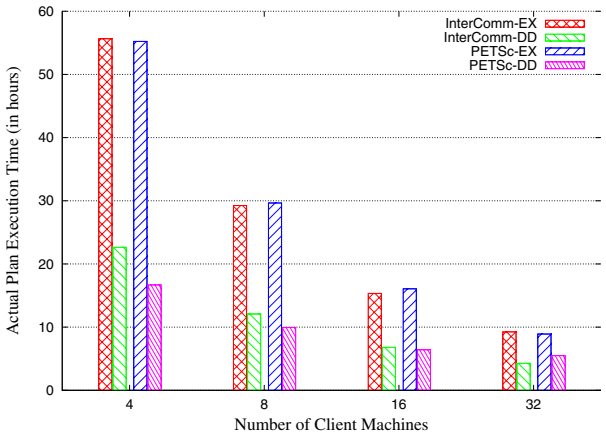

Figure 5: Actual plan execution times for InterComm and PETSc EX-plans and DD-plans using depth-first plan execution strategy. The DD-plan takes much less testing time, and Rachet scales well as the number of available machines increases.

components on which the PETSc component depends (e.g., when Rachet tries to build a version of the PETSc component with the GNU compilers on a VM, the MPICH component might have been previously built on the machine with the GNU compilers *or* with the PGI compilers.) Unfortunately, all those instances were reported as successful builds during the DD-plan execution. We observed that this happened because there were missing constraints in the model. For these instances, the missing constraint was that compilers from the same vendor must be used to build the components on which the PETSc component directly depends. PETSc developers might have simply assumed this constraint. However, users do not always have complete information on the compilers used to build those components on their system, especially if the system is managed by a separate system administrator. Another observation for the PETSc component is that it was never able to be built successfully using the LAM MPI component. It seems that some undocumented method is required to build PETSc using that MPI implementation.

For InterComm, due to many build failures of the components in the model, we were only able to test build compatibility for 7 direct dependencies out of its 156 direct dependencies for the InterComm component. However, they were not the ones on which InterComm has been tested before. The results show that InterComm can be successfully built with the combinations of PGI C/C++ compiler version 6.2, all versions of the GNU Fortran77 or GNU Fortran90 compilers, and MPICH version 1.2.7. This is a larger set of components than what the InterComm developers had previously tested, as documented on the InterComm distribution web page. The direct dependency with GNU C/C++ compiler version 3.3.6 and with the PGI Fortran compiler version 6.2 failed to build. The failure occurred because the InterComm *configure* process reported a problem in linking to Fortran libraries from C code. This result is interesting since the InterComm web page claims that InterComm was successfully built with GNU C/C++ version 3.2.3 and PGI Fortran version 6.0. We reported all the results to the InterComm developers and a bug fix for the failed direct dependency is being investigated.

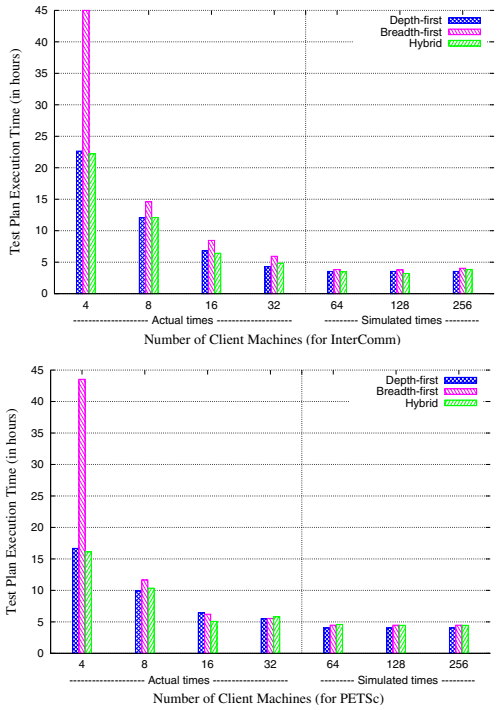

Figure 6: InterComm and PETSc DD-plan execution times, with different plan execution strategies. Breadth-first shows poor performance on few machines. Depth-first and hybrid show similar performance due to many build failures.

6.5 Comparing Plan Execution Strategies

As seen in Figure 5, Rachet scales very well as the number of machines used to run Rachet clients increases from 4 to 32. When we double the number of available machines, the execution time decreases by almost half, up to 16 machines. This means that Rachet can fully utilize additional resources to maximize the number of tasks executed in parallel. However, Figure 5 shows results obtained by executing the DD and EX plans for the subject systems using only the parallel depth-first strategy. To analyze the performance behavior of the different plan execution strategies, we also executed the DD-plans for InterComm and PETSc using the other strategies.

Figure 6 shows the combined results from both actual and simulated plan executions with different strategies. For both systems, we ran actual experiments with 4, 8, 16 and 32 clients. For larger numbers of clients, we ran simulations to compute expected plan execution times. The data used for the simulations, including the component build successes/failures, the average times needed to manage VMs and to build components, were all obtained from real experiments. The simulated times were, on average, about 18% less than the actual times for up to 32 clients.

We found that the breadth-first strategy performed worst for most runs. As described before, with the breadth-first strategy, Rachet tries to utilize as many machines as possible throughout the plan execution, and always reuses the best cached virtual machine to execute each task. However, the time to transfer the VMs across the network was a performance bottleneck, even though the clients were con-

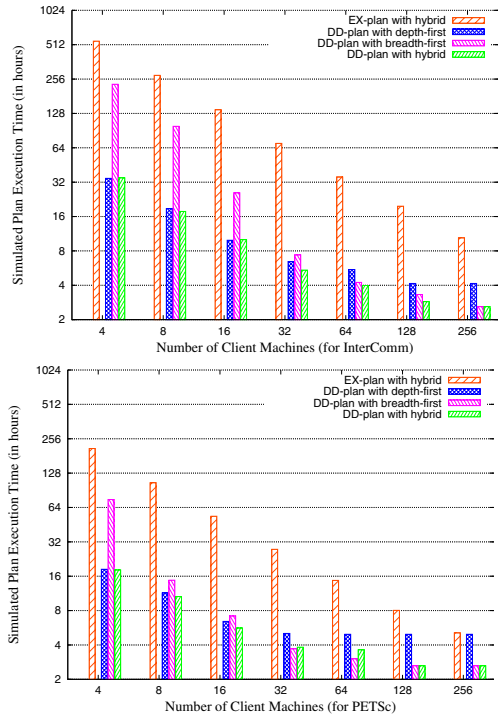

Figure 7: InterComm and PETSc EX-plan simulated execution times with hybrid execution strategy, and DD-plan with all strategies, assuming no build failure. The hybrid strategy achieves good task reuse locality and task parallelism.

nected via Fast Ethernet[4]. Breadth-first performed especially poorly with 4 machines compared to the other strategies, because in many instances the best cached task for executing a new task had already been replaced in the VM cache before it was needed, and as a result all components in the task had to be built. For our experiments and simulations, we used a Least-Recently-Used (LRU) cache replacement policy to manage the VM cache on each machine. We also experimented with a First-In-First-Out (FIFO) cache replacement policy, but did not see a significant performance difference compared to LRU.

Many build failures occurred during plan executions are responsible for the similar performance between the hybrid and depth-first strategy in Figure 6. With a small number of clients, the depth-first strategy can maximize the number of tasks executing in parallel shortly after starting the plan execution, and with many clients, build failures negate the benefits of the hybrid strategy achieved by maximizing the number of tasks started early in the plan execution.

The second observation is that little benefit is achieved with more than 32 machines for all strategies, since many machines remained idle waiting to be assigned tasks, since all available tasks were already assigned to other machines. Moreover, the execution times may even increase slightly with a large number of machines, because the local cache hit rate drops when tasks are spread across the machines, and also because additional time is needed to transfer cached tasks across the network, negating the benefit of greater parallel task execution. In spite of these overheads, we expect that the hybrid strategy will achieve the best performance as we increase the number of machines, if a test plan has fewer failures, and therefore more work to do with task reuse.

For our final experiment, we examined how Rachet behaves as the number of successfully built components grows. As previously described, many direct dependencies for components could not be tested in the earlier experiments, because at least one of the components on which it directly depends could not be built successfully. If developers were to fix some of these problems, many more direct dependencies would be testable, greatly increasing the effective size of the test plan. We ran simulations to examine this scenario and measured the benefit of testing only direct dependencies, under the strong assumption that no build failures occurred during plan execution. Figure 7 shows expected execution times for the test plans for the subject systems. Both the EX-plan and the DD-plan are executed with the hybrid strategy, and we also applied the other strategies for the DD-plan. We observe that the hybrid strategy balances well both task reuse locality and task parallelism across all numbers of clients. The hybrid strategy is competitive with the depth-first strategy for small numbers of clients, because it tries to maximize reuse of locally cached tasks. And the hybrid strategy also achieves good performance for a large number of clients, since the extra costs for caching tasks and reusing VMs during the early parts of test plan execution are avoided, compared to the depth-first strategy. Although the breadth-first strategy shows good performance with 32 or more machines, such performance relies on the availability of a fast network connecting *all* client machines.

7. RELATED WORK

GridUnit/InGrid [6] is a framework that enables distributing JUnit software tests for distributed applications onto multiple machines. The software and test suites for the software are transmitted to a Grid of heterogeneous machines for execution. Deployment and configuration are handled by a system called SmartFrog [11], which uses model-based approaches to describe software configurations. The work has a similar goal to ours, but differs in several ways. In particular, it does not support sharing across machines; all tests are independent. Also, the system does not analyze and sample the configuration space, but simply runs all tests given to it.

The Skoll [9] and BuildFarm [10] projects use heterogeneous and distributed resource to build and/or execute software across large configuration spaces. One way in which these efforts differs from ours is that they focus on configuration spaces as defined by traditional compile- and run-time options. Our approach is focused on configuration spaces defined more by architectural concerns.

Techniques to test highly configurable components have been extended to testing of software product-lines. Cohen et al. [3] apply combinatorial interaction testing methods to define test configurations that achieve a desired level of coverage, and identify challenges to scaling such methods to large, complex software product lines. Although not directly related to our idea of sampling configuration spaces, they too illustrate how software product line modeling notations can be mapped onto an underlying relational model that captures variability in the feasible product line instances. They use the relational model as the basis for defining a family of coverage criteria for product-line testing.

[4]The percentage of VM reuse to execute the InterComm and PETSc DD-Plans was on average 53% for the breadth-first and 80% for the depth-first and hybrid strategies.

Our work is broadly related to component installation managers that deal with dependencies between components. Opium [14] and EDOS [8] are two example projects. Opium makes sure a component can be installed on a client machine, while EDOS checks for conflicting component requirements at the distribution server. Both projects assume that component dependencies are correctly specified by the component distributors. Rachet differs in that we test component compatibility over a large range of configurations in which the components may be installed.

8. CONCLUSIONS AND FUTURE WORK

We have presented Rachet, a process, algorithms and infrastructure to perform compatibility testing. Our work makes several novel contributions: a formal model for describing the configuration space of software systems; algorithms for computing a set of configurations and a test plan that test all *direct dependencies* for components in the model; test plan execution strategies focused on minimizing plan execution time, while allowing contingency management that improves test coverage over static approaches if and when an attempt to build a component fails; and automated infrastructure that executes a test plan in parallel on a set of machines, distributing tasks to best utilize resources.

The results from our empirical studies on two large software systems demonstrate that Rachet can detect *incompatibilities* between components rapidly and effectively without compromising the test quality, compared to the exhaustive approach. We also examined the tradeoffs between plan execution strategies on different system scales, by running both actual experiments and simulations. The results suggest that the hybrid strategy can achieve the best performance by attaining high locality to optimize task reuse and high task parallelism, for both small and large system scales.

Based on these results, we plan to work on several issues. First, we will further optimize the plan execution strategies. Information on the expected cost to execute tasks and benefits of reusing cached tasks might help Rachet to make better decision for task assignment and reuse. Second, we will explore new types of coverage criteria that produce fewer configurations, but still provide effective coverage. Third, we will investigate adding cost models into plan execution strategies to prioritize the order in which tasks are distributed so as to maximize the number of direct dependencies for the SUT tested within a fixed time period. Fourth, for components distributed using popular packaging methods (e.g., Autoconf/Automake), we plan to explore ways to extract dependencies automatically. Finally, we will extend our work to include functional and performance testing for a software system.

Acknowledgments

This research was supported by NASA under Grant #NNG06GE75G and NSF under Grants #CNS-0615072, #ATM-0120950, #CCF-0205265 and #CCF-0447864. This work was also partially supported by the the Office of Naval Research under Grant #N00014-05-1-0421.

9. REFERENCES

[1] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. D. Croz, S. Hammarling, J. Demmel, C. H. Bischof, and D. C. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proc. of Supercomputing '90*, pages 2–11, Nov. 1990.

[2] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.3.2, Argonne National Laboratory, Sep. 2006.

[3] M. B. Cohen, M. B. Dwyer, and J. Shi. Coverage and adequacy in software product line testing. In *In Proc. of ROSATEA '06*, pages 53–63, 2006.

[4] 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.

[5] J. Dongarra, J. D. Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, Mar. 1990.

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

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

[8] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package–based software distributions. In *Proc. of the 21st Int'l Conf. on Automated Software Engineering*, pages 199–208, Sep. 2006.

[9] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *Proc. of the 26th Int'l Conf. on Software Engineering*, pages 459–468, May 2004.

[10] PostgreSQL BuildFarm. http://www.pgbuildfarm.org.

[11] R. Sabharwal. Grid infrastructure deployment using SmartFrog technology. In *Proc. of the 2006 Int'l Conf. on Networking and Services*, Jul. 2006.

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

[13] T. Syrjänen. A rule-based formal model for software configuration. Technical Report A55, Helsinki University of Technology, Dec. 1999.

[14] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. OPIUM: Optimal package install/uninstall manager. In *Proc. of the 29th Int'l Conf. on Software Engineering*, pages 178–188, May 2007.

[15] G. V. Vaughan, B. Elliston, T. Tromey, and I. L. Taylor. *GNU Autoconf, Automake, and Libtool*. Sams, 1st edition, 2000.

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

[17] 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.