Towards Incremental Component Compatibility Testing

Ilchul 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 components are increasingly assembled from other components. Each component may further depend on others, and each may have multiple active versions. The total number of configurations—combinations of components and their versions—deployed by end users can be very large. Component developers, therefore, spend considerable time and effort doing compatibility testing – determining whether their components can be built correctly for all deployed configurations. In previous work we developed *Rachet* to support large-scale compatibility testing of components.

In this paper, we describe and evaluate methods to enable *Rachet* to perform incremental compatibility testing. We describe algorithms to compute differences in component compatibilities between current and previous component builds, a formal test adequacy criterion based on covering the differences, and cache-aware configuration sampling and testing methods that attempt to reuse effort from previous testing sessions. We evaluate our approach using the 5-year evolution history of a scientific middleware component. Our results show significant performance improvements over *Rachet*'s previous retest-all approach, making the process of compatibility testing practical for evolving components.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Design, Experimentation

Keywords

incremental testing, software component, compatibility

1. INTRODUCTION

Testing modern software components is extremely difficult. One particular challenge is that components may be built in a great many *configurations*. Consider, for example, InterComm [10], a component used to support coupled parallel scientific simulations. InterComm has complex dependencies on multiple third-party components, each of which in turn depend on other components, and every component has multiple active versions. Each possible combination of components and their versions is a configuration that might contain unique errors. To make matters worse, each component may evolve independently, and each configuration may need to be rebuilt and retested after each change.

In past work, we have addressed some of these challenges by creating *Rachet* [14, 15, 16], a process and infrastructure for testing whether a component can be *built* correctly for all its configurations. *Rachet* includes a formal, graph-based representation for encoding a component's *configuration space* – the set of all possible configurations. Using this representation, developers specify the components and versions, component dependencies, and constraints. *Rachet* then automatically computes the component's configuration space. *Rachet* also defines a test adequacy criterion and algorithms for generating a set of configurations that satisfy it. Finally, *Rachet* efficiently tests the selected configurations, distributing the *build* effort across a grid of computers. However, *Rachet* did not accommodate component evolution; it simply retested all configurations, including the ones that had already been tested in previous test sessions.

In this paper, we describe methods to enhance *Rachet* that test component compatibilities incrementally, taking into account various types of component changes. We evaluate the effectiveness of our approach on 20 actual builds for the InterComm component, developed over a 5-year period. Our results show that the incremental approach is more efficient than the previous *retest-all* approach. We also present optimization techniques to reduce testing time even further by reusing test artifacts and results inherited from previous test sessions. More specifically, the work described in this paper makes the following contributions:

- An *incremental compatibility testing* adequacy criterion;
- An algorithm to compute incremental testing obligations, given a set of changes to the configuration space;
- An algorithm to select *small* sets of configurations that efficiently fulfill incremental testing obligations;
- A set of optimization techniques that reuse test artifacts and results from previous test sessions, to greatly decrease testing time.

The next section provides an overview of *Rachet*. Section 3 defines a test adequacy criterion for incremental com-

CBSE'11, June 20–24, 2011, Boulder, Colorado, USA.

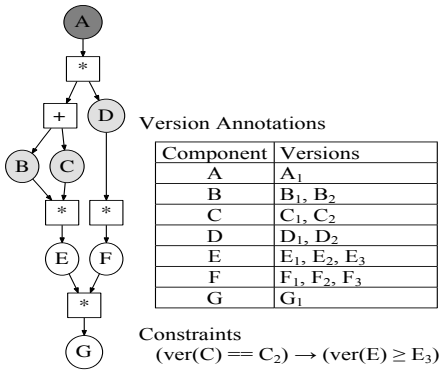


Figure 1: Example Configuration Model

patibility testing, and also describes algorithms to generate configurations for incremental compatibility testing. Section 4 presents the results of our empirical study. Section 5 describes related work and Section 6 concludes with a brief discussion of future work.

2. RACHET OVERVIEW

In our previous work [14, 15, 16], we developed *Rachet*, a process and tool to perform compatibility testing for components. The *Rachet* process has several steps. First, developers formally model the configuration space of the component under test. The model has two parts: (1) a directed acyclic graph called the *Component Dependency Graph (CDG)* and (2) a set of *Annotations*. As demonstrated in Figure 1, a CDG contains one node for each component and specifies inter-component dependencies by connecting component nodes through **AND** and **XOR** relation nodes. For example, component `A` depends on component `D`, which is represented by an **AND** node labeled `*` between component `A` and component `D`. Component `A` also depends on exactly one of either `B` or `C`, which is represented by the **XOR** node labeled `+`. The model's *Annotations* include one identifier for each component version and, optionally, constraints between components and/or over full configurations, expressed in first-order logic. A model must contain *all* component dependencies and constraints, and the model cannot change in a single test session.

Together, the CDG and Annotations implicitly define the configuration space. More formally, a configuration to build the component represented by the top node of the CDG is a sub-graph that contains the top node, the relation node connected to the outgoing edge of the top node, and other nodes reachable from the top node, where we pick one child node for an **XOR** node and all child nodes for an **AND** node. Each component node is labeled with exactly one valid version identifier. A *valid* configuration is one that does not violate any constraint specified in a model, and the configuration space is the set of all valid configurations. For the example in Figure 1, a sub-graph that contains the nodes `A`, `B`, `D`, `E`, `F`, `G`, along with intervening relation nodes, is a valid configuration to build the component `A`. However, for example, if the sub-graph does not contain component `G`, it is not a valid configuration.

Because it is often infeasible to test all possible configurations, *Rachet*'s second step is to select a sample set of configurations for testing. The default sampling strategy is called *DD-coverage*, and is based on covering all *direct dependencies* between components. In CDG terms, a component c *directly depends* on a set of components, DD, such that for every component, $DD_i \in DD$, there exists at least one path from c to DD_i not containing any other component node. In the running example, component `A` directly depends on components `B`, `C` and `D`, and component `A` has two direct dependencies – one is to *build* `A` with `B` and `D` and the other with `C` and `D`.

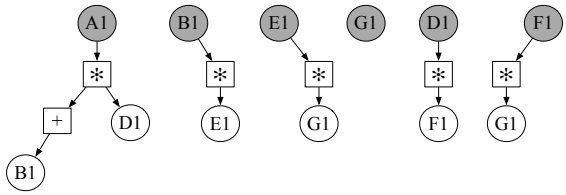


Figure 2: Example applying the BuildCFG algorithm to cover a *DD-instance* for component A.

From these direct dependencies, *Rachet* computes *DD-instances*, which are the concrete realizations of direct dependencies, specifying actual component versions. A *DD-instance* is a tuple, (c_v, d), where c_v is a version v of component c, and d is a set of component versions on which c directly depends. For example, there are 3 *DD-instances* for component `E` in Figure 1: (E_1,{G_1}), (E_2,{G_1}), (E_3,{G_1}). Once all *DD-instances* for all components in the model have been computed, *Rachet* computes a set of configurations in which each *DD-instance* appears at least once. *Rachet* implements this step with an algorithm called **BuildCFG**. The algorithm works greedily, attempting to generate each configuration to cover as many previously uncovered *DD-instances* as possible, with the goal of reducing the total number of configurations needed to cover all *DD-instances*.

The **BuildCFG** algorithm takes two parameters: (1) a set of *DD-instances* already selected for the configuration under generation, and (2) a set of component versions whose *DD-instances* must still be added to the configuration. To generate a configuration that is to cover a given *DD-instance* (call it $ddi_1 = (c_v, d)$), *Rachet* calls **BuildCFG** with the first parameter set to ddi_1 and the second parameter containing all the component versions in d. **BuildCFG** then selects a *DD-instance* for some component version in the second parameter. The configuration (the first parameter) is extended with that *DD-instance*, and component versions contained in the dependency part of the *DD-instance* are added to the second parameter, if *DD-instances* for those component versions are not yet in the configuration. **BuildCFG** then checks whether the extended configuration violates any constraints. If the configuration does not violate constraints, **BuildCFG** is called recursively with the extended configuration and the updated second parameter. If there has been a constraint violation, **BuildCFG** backtracks to the state before the *DD-instance* was selected and tries another *DD-instance*, if one exists. **BuildCFG** returns true if the configuration has been completed (i.e., the second parameter is empty) or false if it runs out of *DD-instances* that can be selected, due to constraint violations. If all of those calls return success, the configuration under construction contains all *DD-instances* needed for a configuration that covers ddi_1 (and all other *DD-instances* selected for the configuration).

The **BuildCFG** algorithm is applied first to generate a configuration that covers a *DD-instance* for a component at smaller depth in a CDG – the component at depth 0 is the top component, since the algorithm may then choose

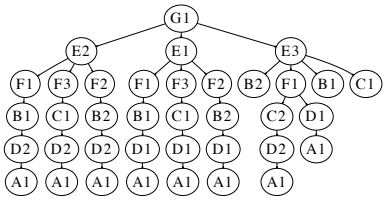


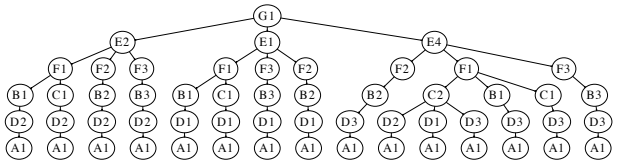
Figure 3: Test plan with DD-coverage. (Dependency part of each *DD-instance* is omitted.)

more *DD-instances* that have not been covered by any other configuration in the recursive process, and as a result, we may reduce the total number of configurations generated. Figure 2 illustrates the process of generating a configuration to cover a *DD-instance* for component `A`, ($\texttt{A}_1$, $\{\texttt{B}_1,\texttt{D}_1\}$). Starting from the leftmost sub-graph, the figure shows *DD-instances* selected for the configuration. For the example model in Figure 1, **BuildCFG** generates 11 configurations to cover all *DD-instances*.

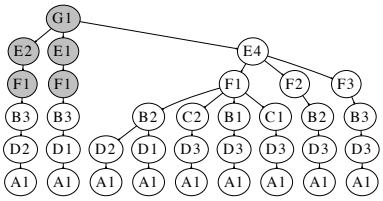
Rachet's third step takes each of the configurations and topologically sorts its component nodes to produce a build-ordered sequence of components. That is, the i^{th} component in the sequence does not depend on any component with an index greater than i. Therefore, *Rachet* can first build the 1^{st} component in the sequence, then build the 2^{nd} component, etc. *Rachet* combines the build sequences for each configuration into a *prefix tree*, by representing each common build prefix (a build subsequence starting from the first component) exactly once. Thus, each path from the root node to a leaf node corresponds to a single build sequence, but common build subsequences are explicitly represented. The rationale behind combining configurations is that many configurations are quite similar, so we can reduce test effort by sharing partially built configurations across multiple configurations. The prefix tree essentially acts as a *test plan*, showing all opportunities to share common build effort. Figure 3 depicts a test plan for the example. This test plan contains 37 nodes (components to be built), reduced from 56, the number contained in the 11 original configurations generated by applying **BuildCFG**.

Finally, *Rachet* executes the test plan by distributing component builds across multiple client machines and collecting the results. Instead of distributing complete configurations, *Rachet* distributes partial configurations (*prefixes* in the test plan). The partial configurations are built in virtual machines (VMs), which can be *cached* (since a VM is a (large) disk image), for reuse in building other configurations, – i.e., *Rachet* tries to build a prefix only once, reusing it to build other configurations, sometimes by transferring a VM image between machines across the network. In previous work [14, 15], we examined three different test plan execution strategies, where each uses a different method to select the next prefix to distribute to a machine that performs the build: a *depth-first* strategy, a *breadth-first* strategy and a *hybrid* strategy. In our studies, the *hybrid* strategy has generally performed best across a wide variety of execution scenarios. In the strategy, *Rachet* first distributes prefixes for non-overlapping subtrees of a test plan to each machine and then continues building components in depth-first order.

Rachet's final output is test results indicating whether each *DD-instance* was (1) tested and built successfully, (2) tested and failed to build, or (3) was untestable, meaning that there was no way to produce a configuration to test



(a) A test plan that retests all *DD-instances*



(b) An incremental test plan

Figure 4: Test plans: Retest-All (56 components) vs. Incremental (35 components). The shaded nodes can also be reused from the previous test session.

that *DD-instance*. For example, suppose that in testing for our example, all attempts to build $\texttt{B}_2$ with $\texttt{E}_1$, $\texttt{E}_2$, and $\texttt{E}_3$ fail. Then, two *DD-instances* of component `A`, ($\texttt{A}_1$,$\{\texttt{B}_2,\texttt{D}_1\}$) and ($\texttt{A}_1$,$\{\texttt{B}_2,\texttt{D}_2\}$), are untestable because the *DD-instances* require a successfully built B_2 to build $\texttt{A}_1$.

3. INCREMENTAL TESTING

Our previous work cannot be used to test evolving systems, because *Rachet* will generate configurations that test all *DD-instances* for components in a model after component changes, and the set can include unnecessary configurations that only test *DD-instances* whose results are already known in previous testing sessions. To support incremental compatibility testing, we have extended *Rachet* to (1) identify a set of *DD-instances* that need to be tested given a set of component changes, and (2) compute a small test set of configurations that cover those *DD-instances*.

Consider the running example from Figure 1. Suppose that during the last testing session, $\texttt{B}_2$ could not be built over any version of component `E`. As a result, all *DD-instances* in which component `A` must be built over $\texttt{B}_2$ have not been tested. Now suppose that new versions of components `B` and `D` become available, and that the latest version of `E`, $\texttt{E}_3$, has been modified. In this case the configuration model changes in the following ways. First, the new versions of `B` and `D` are added to the configuration model as version identifiers $\texttt{B}_3$ and $\texttt{D}_3$. Next, the modified component is handled by removing the old version, $\texttt{E}_3$, and then adding a new version, $\texttt{E}_4$. For this example, *Rachet* would previously produce a test plan with 56 component versions to build (Figure 4(a)), which is larger than necessary because some configurations involve only result-known *DD-instances*.

3.1 Computing Incremental Test Obligations

The types of changes can include adding/deleting components, component versions, dependencies or constraints. To deal with all such changes in a uniform way, we assign unique identifiers to each component and its versions, and compute the set of *DD-instances* for both the old and new configuration models and then use set differencing operations to compute the *DD-instances* to be tested. Using a Venn diagram, we can describe the relationship between the *DD-instances* for two successive builds. Figure 5 shows the

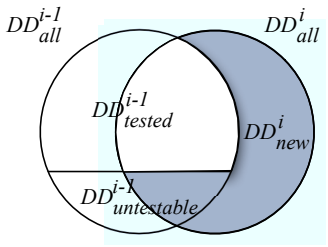


Figure 5: *DD-instances* for two consecutive builds, $build_{i-1}$ and $build_i$. The *DD-instances* represented by the shaded areas need to be tested in $build_i$.

set of *DD-instances* for two consecutive builds, $build_{i-1}$ and $build_i$. DD^{i-1}_{all} and DD^{i}_{all} represent the sets of all *DD-instances* in the respective builds. DD^{i}_{new} represents the *DD-instances* in DD^{i}_{all}, but not in DD^{i-1}_{all}. DD^{i-1}_{tested} is the subset of DD^{i-1}_{all} whose build status (success or failure) was determined in that testing session and $DD^{i-1}_{untestable}$ is the subset of DD^{i-1}_{all} whose build status is unknown – each of those *DD-instances* could not be tested because at least one of the component versions in the dependency part of the *DD-instance* failed to build in all possible ways.

Using this set view, the *DD-instances* that must be tested for $build_i$ are shown as the shaded area in the figure, and are computed as follows:

$$DD^{i}_{test} = DD^{i}_{all} - DD^{i-1}_{tested}$$

We include previously untestable *DD-instances* in the current testing obligations, since newly introduced component versions might provide new ways to build a given component, thus enabling previously untestable *DD-instances* to be tested.

The next step applies the **BuildCFG** algorithm as many times as needed to generate a set of configurations that cover all the *DD-instances* in DD^{i}_{test}. The algorithm generates configurations for *DD-instances* that have not yet been covered, starting from *DD-instances* for the component closest to the top node in a CDG. As previously discussed in Section 2, we expect to generate fewer configurations. An outline of this process is as follows:

1. Compute DD^{i}_{test}.
2. Select the *DD-instance* from DD^{i}_{test} that is to build the component closest to the top node of the CDG (if more than one, select one at random).
   - 2.1 Generate a configuration that covers the selected *DD-instance*, by applying **BuildCFG**.
   - 2.2 Remove all *DD-instances* contained in the generated configuration from DD^{i}_{test}
   - 2.3 If DD^{i}_{test} is not empty, go to step 2.
3. Merge all generated configurations into a test plan.
4. Execute the test plan.

On the running example, this new algorithm produces 9 configurations, reducing the test plan size from 56 (Figure 4(a)) to 35 components (Figure 4(b)). As the test plan executes, *Rachet* caches *partially-built configurations* (prefixes) on the client machines when a prefix can be reused later in the test process. As a result, for the running example, the total number of component builds is only 30, because the 5 components depicted by shaded nodes in Figure 4(b) have already been built in partial configurations and the configurations are cached in the previous test session (assuming those partial configurations were not deleted at the end of testing).

3.2 Cache-aware Configuration Generation

In our previous studies [14, 15, 16], we assumed that the cache space in each test machine is empty at the beginning of a test process. For incremental testing, however, previous efforts can and should be reused. On the other hand, just preserving the cache between test sessions may not actually result in reduced effort unless the cached prefixes are shared by at least one configuration generated for the new test session. We present a method that uses information about cached prefixes from previous test rounds in the process of generating configurations, to attempt to increase the number of configurations that contain cached prefixes. More specifically, step 2.1 in the configuration generation algorithm from Section 3.1 is modified as follows:

- 2.1.1 Pick the *best* prefix in the cache for generating a configuration that covers the *DD-instance*.
- 2.1.2 Generate a configuration by applying **BuildCFG**, using the prefix as an extension point.
- 2.1.3 Repeat from step 2.1.1 with the next best prefix, if no configuration can be generated by extending the best prefix.

To generate a configuration that covers a *DD-instance*, in step 2.1.1, we first pick the *best* prefix, which is one that requires the minimum number of additional *DD-instances* to turn the prefix into a full configuration. Then in the 2.1.2, the **BuildCFG** algorithm is used to extend the prefix by adding *DD-instances*. It is possible that **BuildCFG** will fail to generate a configuration by extending the best prefix, due to constraint violations. In that case, the new algorithm applies step 2.1.1 with the next best cached prefix, until one is found that does not have any constraint violations.

However, the best cached prefix can be found only *after* applying the **BuildCFG** algorithm to every prefix in the cache. That can be very costly, since the algorithm must check for constraint violations whenever a *DD-instance* is added to the configuration under construction. Therefore, we instead employ a heuristic that selects the best prefix as the one that requires the longest time to build all the components in the prefix. The rationale behind this heuristic is that fewer *DD-instances* should need to be added when a configuration is constructed by extending the cached prefix that takes longest to build. However, the downside of this heuristic is that a prefix could be regarded as the best prefix to cover a *DD-instance* only because it takes the longest time to build, even though many components in the prefix are not really needed. We currently overcome this problem by not considering a prefix as an extension point if it contains at least one component that appears later in the topological ordering of the components in the CDG than the component in the *DD-instance* to be covered.

Not all prefixes in the cache can be extended to generate a configuration that covers a *DD-instance*. To reduce the cost to generate configurations, we check whether any constraint is violated when the *DD-instance* is added to each cached prefix, before extending the prefix with the *DD-instance*. This can be achieved efficiently by maintaining an auxiliary data structure called a *cache plan*, which is a prefix tree that combines prefixes in the cache. (In Figure 6, the sub-tree

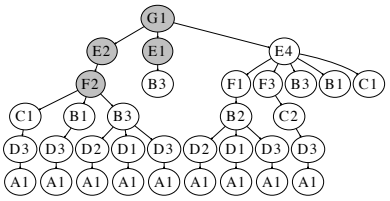


Figure 6: Test plan produced from configurations selected in a cache-aware manner. 34 component versions must be built. (Shaded area is cached, from the previous test session.)

reaching the shaded nodes is the cache plan for the example system, after the first test session completes.) For a *DD-instance* that is to be covered, the cache plan is traversed in depth-first order, checking for whether constraints are violated when the *DD-instance* is added to the path from the root node to a node in the cache plan. If there is a violation, we filter out all cached prefixes reaching any node in the subtree starting at the node.

Figure 6 shows a test plan created by merging the configurations produced by applying the cache-aware algorithm to the example system. The test plan has 13 configurations, which is 4 more than the test plan that does not consider cached prefixes (Figure 4(b)). The number of components that actually need to be built is 30 in both cases because we can reuse prefixes in the cache. However, the average build sequence length is smaller for the cache-aware plan by about 1 1/3 components compared to the cache-unaware plan, because almost half of its configurations are extended from cached prefixes. This factor can greatly decrease the turnaround time needed to complete the test plan.

3.3 Managing Cached Configurations

Because cache space is a limited resource, so when the cache is full we must discard a previously cached prefix when adding a new one. In previous work [14, 15, 16], we employed the commonly used *Least-Recently-Used* (LRU) cache replacement policy. However, during the execution of a test plan, *Rachet* can, for each prefix in the cache, compute how many times the prefix can be reused for testing additional *DD-instances*. This information can then be used to select the victim prefix to be replaced in the cache. For example, if all the plan nodes in the subtree rooted at the last node of a prefix have already been tested, the prefix can be deleted from the cache even though it has been recently used, without increasing overall test plan execution time.

To keep prefixes with more reuse potential longer in the cache throughout test sessions, we have designed a heuristic to estimate the reuse potential of prefixes in the cache. When we need to replace a prefix in the cache, we compute, for each prefix in the cache: (1) the expected time saving by reusing the prefix to execute the remaining portion of the current test plan, and (2) the average change frequency of components in the prefix across previous test sessions.

The expected time savings measures how useful a prefix can be for executing the current test plan. To compute the expected time savings for each prefix, we first identify, for each plan node, the cached prefix that enables saving the most time to test the node by reusing that cached prefix. Then, we multiply the number of nodes that benefit the most from reusing the prefix by the time required for building the prefix from an empty configuration. In Figure 6, prefixes <G_1,E_2> (call that p_1) and <G_1,E_2,F_2> (call that p_2) are cached during the first test session. When the test plan in the figure is executed in the next test session, the time savings expected from p_1 is 0, since p_2 is the best prefix for testing all plan nodes in the subtree starting from p_1.

We also estimate how likely a prefix cached during the execution of a test plan is to be helpful to execute test plans for subsequent test sessions, by considering change frequencies of components in the prefix. Component version annotations in the CDG can include both officially released versions of a component and also the latest states of development branches for a component from a source code repository, because developers often want to ensure compatibility of a component with the most recent versions of other components. To model an updated system build, a developer must specify modified component versions in version annotations, including patches for released versions or code changes in development branches. We regard such changes as version replacements in the CDG annotations, but also keep track of the test sessions in which the changes occurred.

The change frequency of a cached prefix is computed by counting the number of preceding test sessions in which a component version has changed. We do the counting for each component version contained in the prefix and compute the average across the components to compute the frequency for the whole prefix. Therefore, if a prefix in the cache contains only component versions that have not changed at all, the change frequency is 0, which means that components involved in the prefix are not likely to change in the future so that it may be worthwhile to keep the prefix in the cache. On the other hand, if a prefix contains only component versions that have changed often across test sessions, it is more likely that the prefix is not reusable in later test sessions.

When a cache replacement is necessary, the victim is the prefix that has the least time savings. The highest change frequency is used as a tie breaker. That is, we first focus on completing the test plan under execution more quickly and secondarily try to keep prefixes that may be useful for later test sessions.

The scheduling strategy for test plan execution cannot be considered separately from the cache replacement policy. For the *hybrid* scheduling strategy described in Section 2, when a client requests a new prefix to test, the scheduler searches the test plan in breadth-first order starting from the root node, or, if that client has cached prefixes available for the test plan, in depth-first order from the last node of the most recently used cached prefix.

For the new cache replacement policy, the prefix with the least reuse potential, call it p_1, is replaced when the cache is full. If the test plan is searched starting from the most recently used cached prefix, p_1 could be replaced before it is reused. If such a replacement happens, we must pay the cost to build p_1 from scratch later when we need p_1 for testing plan nodes beneath the subtree rooted at p_1. Hence, we search the test plan giving higher priority to prefixes with low reuse potential, because such prefixes are more likely to be reused for testing only a small part of the test plan. By testing those parts of the plan earlier, those prefixes can be replaced after they are no longer needed.

4. EVALUATION

Having developed a foundation for incremental compatibility testing between evolving components, we now evaluate

Comp.	Description
ic	InterComm, the SUT
ap	High-level C++ array management library
pvm	Parallel data communication component
lam	A library for MPI (Message Passing Interface)
mch	A library for MPI
gf	GNU Fortran 95 compiler
gf77	GNU Fortran 77 compiler
pf	PGI Fortran compiler
gxx	GNU C++ compiler
pxx	PGI C++ compiler
mpfr	A C library for multi-precision floating-point number computations
gmp	A library for arbitrary precision arithmetic computation
pc	PGI C compiler
gcr	GNU C compiler
fc	Fedora Core Linux operating system

Table 1: Description of components used in the InterComm model

our new approach and algorithms on a middleware component from the high performance computing community.

4.1 Subject Component

Our subject component for this study is *InterComm*[1], which is a middleware component that supports coupled scientific simulations by enabling efficient and parallel data redistribution between data structures managed by multiple parallel programs [4, 10]. To provide this functionality, InterComm relies on other components, including compilers for various programming languages, parallel data communication libraries, a process management library and a structured data management library. Some of these components have multiple versions and there are dependencies and constraints between the components and their versions.

For this study, we examined the change history of InterComm and the components required to build InterComm over a 5 year period. To limit the scope of the study, we divided this 5 year period into 20 epochs, each lasting approximately 3 months. We took a snapshot of the entire system for each epoch, producing a sequence of 20 *builds*.

4.2 Modeling InterComm

We first modeled the configuration space for each build. This involved creating the CDGs, and specifying version annotations and constraints. We considered two types of version identifiers – one is for identifying versions officially released by component developers, and the other is for the change history of branches (or tags) in source code repositories. Currently, the modeling is manual work and based on careful inspection of the documents that describe build sequences, dependencies and constraints for each component.

Figure 7 depicts dependencies between components for one build, and Table 1 provides brief descriptions of each component. The CDGs for other builds were different. For instance, GNU Fortran (`gf`) version `4.0` did not yet exist when the first version of InterComm (`ic`) was released. Therefore, the CDG for that build does not contain the Fortran component and all its related dependencies (shaded nodes in the figure).

Table 2 shows the history of version releases and source code changes for the components in each build. Each row corresponds to a specific build date (snapshot), and each column corresponds to a component. For each build, en-

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

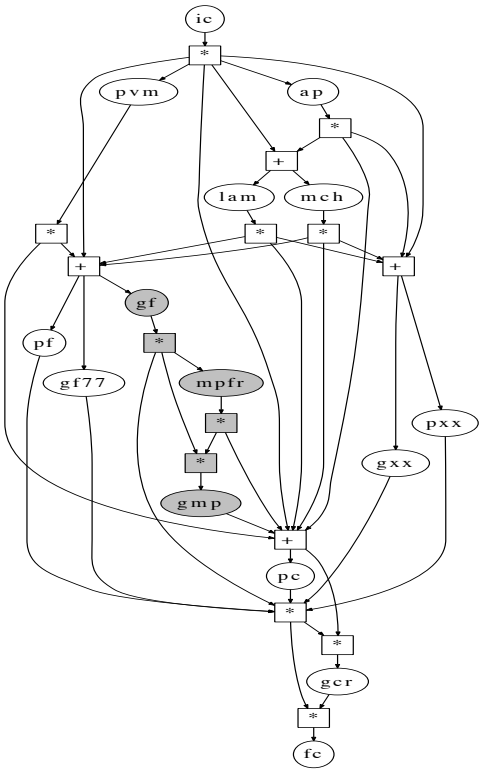


Figure 7: CDG for InterComm

tries in the last 8 columns of the table indicate official version releases of components. For example, InterComm (`ic`) version `1.5` was released between 02/25/2006 ($build_6$) and 05/25/2006 ($build_7$). We use a version released at a given build date to model that build and also for modeling all subsequent builds. Entries in the 6 columns labeled **Branches** contain version identifiers for development branches. We assign a unique version identifier for the state of a branch at a given build date by affixing to the branch name an integer that starts at 1 and is incremented when the branch state at a build date has changed from its state in the previous build.[2] For example, `1.1d2` in the third column of $build_5$ indicates that there were file changes in the branch `1.1d` between 08/25/2005 ($build_4$) and 11/25/2005 ($build_5$). Compared to a released version whose state is fixed at its release date, the state of a branch can change frequently and developers typically only care about the current state for testing. Therefore, for a branch used to model a build, we consider only the latest version identifier of the branch, so include the latest version identifier in the model and remove the previous version identifier for the branch.

Using this information, we define a build to contain all released component version identifiers available on or prior to the build date, and the latest version identifiers for branches available on that date. Note that Table 2 does not include several components: `fc` version `4.0`, `ap` version `0.7.9`, `mch` version `1.2.7`, and the PGI compilers (`pxx, pc, pf`) version `6.2`. For these components, we could access only one version or we considered only one version to limit the experimental effort (`fc`). For this study, we assumed that these versions were available from the first build. Also, we considered de-

[2] Branches are not used for modeling builds unless there has been at least one official version released from the branch.

		Branches						Released Versions							
Build	Date	ic	gcr gxx	gf77	gf	gmp	mpfr	ic	gcr gxx	gf77	gf	gmp	mpfr	lam	pvm
0	08/25/04		3.4d1	3.4d1				1.1	3.4.0 3.4.1	3.4.0 3.4.1				6.5.9 7.0.6	3.2.6 3.3.11
1	11/25/04	1.1d1	3.4d2	3.4d2					3.4.2 3.4.3	3.4.2 3.4.3					3.4.5
2	02/25/05		3.4d3	3.4d3											
3	05/25/05		3.4d4, 4.0d1	3.4d4	4.0d1				3.4.4 4.0.0	3.4.4	4.0.0	4.1.0,4.1.1 4.1.2,4.1.3 4.1.4	2.1.0 2.1.1		
4	08/25/05		3.4d5, 4.0d2	3.4d5	4.0d2				4.0.1		4.0.1		2.1.2		
5	11/25/05	1.1d2	3.4d6, 4.0d3	3.4d6	4.0d3				4.0.2		4.0.2		2.2.0		
6	02/25/06		3.4d7, 4.0d4	3.4d7	4.0d4				3.4.5	3.4.5					
7	05/25/06	1.1d3	3.4d8 4.0d5, 4.1d1	3.4d8	4.0d5, 4.1d1			1.5	3.4.6 4.0.3 4.1.0 4.1.1	3.4.6	4.0.3 4.1.0 4.1.1	4.2.0,4.2.1			
8	08/25/06	1.5d1	4.0d6, 4.1d2		4.0d6, 4.1d2										
9	11/25/06		4.0d7, 4.1d3		4.0d7, 4.1d3										
10	02/25/07	1.5d2	4.0d8, 4.1d4		4.0d8, 4.1d4		2.2d1		4.0.4		4.0.4		2.2.1	7.1.3	
11	05/25/07	1.5d3	4.1d5		4.1d5		2.2d2		4.1.2		4.1.2				
12	08/25/07	1.5d4	4.1d6		4.1d6										
13	11/25/07	1.5d5	4.1d7		4.1d7		2.3d1					4.2.2	2.3.0		
14	02/25/08		4.1d8		4.1d8		2.3d2						2.3.1		
15	05/25/08		4.1d9		4.1d9										
16	08/25/08		4.1d10		4.1d10		2.3d3					4.2.3			
17	11/25/08						2.3d4					4.2.4	2.3.2		
18	02/25/09		4.1d11		4.1d11		2.3d5								
19	05/25/09					4.3d1						4.3.0,4.3.1			

Table 2: History of version releases and source code changes for components in the InterComm build sequence

velopment branches for only 4 major GNU compiler versions (`3.3`, `3.4`, `4.0` and `4.1`), due to limited test resource availability and the time required to perform the experiments.

In addition to the CDGs and version annotations, InterComm places several constraints on configurations. For example, if compilers from the same vendor for different programming languages are used in a configuration (e.g., `gcr`, `gxx`, `gf` and `gf77`), they must have the same version identifier. These constraints eliminated configurations that we knew a priori would not build successfully.

4.3 Study Setup

To evaluate our incremental testing approach, we first gathered component compatibility data (i.e., success or failure to build the component version encoded by each *DD-instance*) and on the time required to build each component version. To obtain this data, we created a single configuration space model containing identifiers for all released component versions and all branch states that appear in any build. We then built every configuration using a single server (Pentium 4 2.4GHz CPU with 512MB memory, running Red Hat Linux) and 32 client machines (Pentium 4 2.8GHz Dual-CPU machines with 1GB memory, all running Red Hat Enterprise Linux), connected via Fast Ethernet. To support the experiment we enhanced *Rachet* to work with multiple source code repositories and to retrieve source code for development branches as needed. Currently, *Rachet* supports the CVS, Subversion and Mercurial [7] source code management systems.

By running the test plan for the integrated model, we obtained compatibility results for 15128 *DD-instances* needed to test the InterComm builds. Building components was successful for 6078 *DD-instances* and failed for 1098 *DD-instances*. The remaining 7952 *DD-instances* were *untestable* because there was no possible way to successfully build one or more components in the dependency part of the *DD-instances*. For example, all the DD-instances to build an InterComm version with the dependency to the PVM component version `3.2.6` could not be tested, because *all* possible ways to build that PVM version failed.

Using the data obtained from the experimental run described earlier, we simulated a variety of use cases with different combinations of client machines and cache sizes. For example, we used the average time required to build a component for calculating total build times for each simulation. Table 3 shows the number of *DD-instances* corresponding to each region in the diagram in Figure 5.

For the i-th build in the InterComm build sequence, the second column in the table is the total number of DD-instances represented by the annotated CDG. Note that for some builds the number of *DD-instances* does not differ from the previous build. This is because model changes between builds only involved replacing branch version identifiers with more recent ones. The last column is the number of nodes in the initial test plan for each build. In some cases, the number of nodes in a test plan is fewer than the number of *DD-instances* to cover (the sum of the 4th and 5th columns). That happens when a large number of *DD-instances* are classified as untestable when we produce the set of configurations that are merged into the test plan for the build.

We ran the simulations with 4, 8, 16 or 32 client machines, each having 4, 8, 16, 32, 64 or 128 cache entries. To distribute configurations to multiple machines, we used the modified plan execution strategy described in Section 3.3. For each machine/cache combination, we conducted multiple simulations to test the InterComm build sequence: (1) *retest-all*: retests all *DD-instances* for each build from scratch ($DD^i_{test} = DD^i_{all}$), (2) *test-diff*: tests builds incrementally ($DD^i_{test} = DD^i_{all} - DD^{i-1}_{tested}$) but without reusing configurations cached in prior builds, (3) *c-forward*: *test-diff* plus reusing cached configurations across builds but without applying any optimization technique, (4) *c-aware*: *c-forward* plus applying cache aware configuration production (Section 3.2), (5) *integrate-all*: *c-aware* plus applying the im-

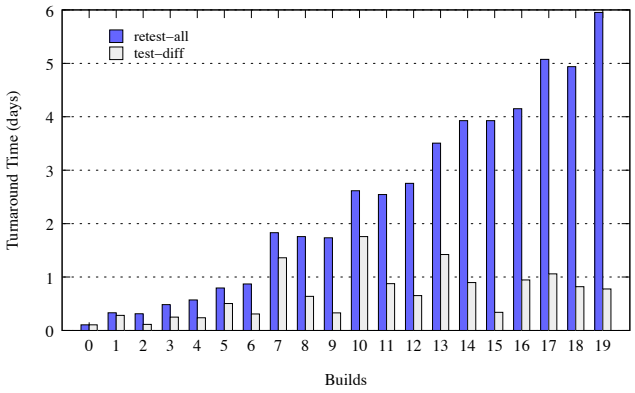


Figure 8: Turnaround times for testing DD^i_{all} and DD^i_{test} for each build (8 machines and 4 cache entries per machine)

i	dd^i_{all}	$dd^{i-1}_{tested} \cap dd^i_{all}$	$dd^{i-1}_{untestable} \cap dd^i_{all}$	dd^i_{new}	# of plan nodes
0	123	0	0	123	252
1	403	44	42	317	579
2	403	141	186	76	170
3	781	141	186	454	756
4	945	271	320	354	753
5	1129	287	255	587	1061
6	1229	411	498	320	561
7	2480	416	341	1723	2766
8	2921	981	1170	770	960
9	2921	1050	1488	383	758
10	4407	981	1170	2256	3243
11	4407	1450	1886	1071	1594
12	4407	1585	1940	882	915
13	5064	1585	1940	1539	2078
14	5296	2031	2514	751	1654
15	5296	2355	2622	319	706
16	5576	2193	2568	815	1754
17	6146	2586	2728	832	1414
18	6146	2877	2622	647	1684
19	7073	3301	2844	928	1704

Table 3: Numbers of DD-instances for the InterComm build sequence

proved cache management strategy (Section 3.3). We measured the turnaround time for each build in the sequence, for the different test strategies.

4.4 Retest all vs. Incremental test

The configuration space for InterComm grows over time because it incorporates more component versions. As a result, we expect incremental testing to be more effective for later builds. Figure 8 depicts the turnaround times for all 20 builds. The testing is done in two ways; by retesting all *DD-instances* for each build and by testing *DD-instances* incrementally. It is clear that turnaround times are drastically reduced with incremental testing. For example, for the last build, *retest-all* takes about 6 days, while incremental testing takes less than one day.

With *retest-all*, the turnaround time required for a test session increases as the number of *DD-instances* (DD^i_{all}) increases. However, for incremental testing, the test time varied depending on the number of *DD-instances* covered by generated configurations. For example, as seen in Table 3, the sizes of DD^i_{test} ($DD^i_{all} - DD^{i-1}_{tested}$) for build 11 and build 15 are comparable (2957 for build 11 and 2941 for build 15), but the required testing time for build 11 is

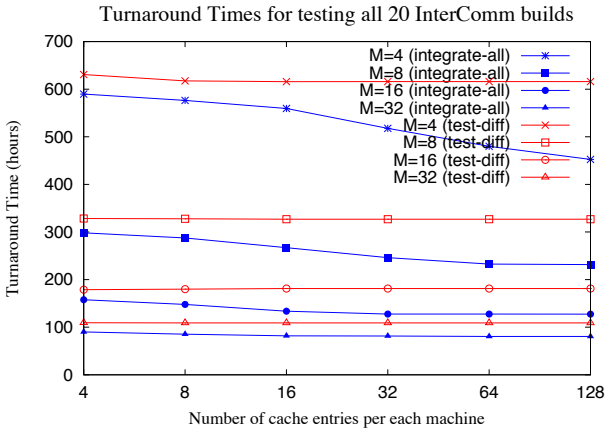


Figure 9: As the number of cache entries per machine increases, overall test cost decreases up to 24% when optimization techniques are applied, compared to the baseline incremental test.

twice as much as the time for build 15. The reason is that 857 *DD-instances* were covered by configurations generated for build 11, compared to 369 for build 15. The rest of the *DD-instances* were classified as untestable while generating configurations, because there was no possible way to generate configurations to cover those *DD-instances* because of build failures identified in earlier builds.

4.5 Benefits from Optimization Techniques

Figure 9 depicts the *aggregated* turnaround times for incremental testing without cache reuse across builds (*test-diff*) and for incremental testing with all optimization techniques applied (*integrate-all*). The x-axis is the number of cache entries per client machine and the y-axis is turnaround times. The simulations use 4 to 32 client machines and the number of cache entries per machine varies from 4 to 128.

As we increase the number of cache entries, we observe that the optimization techniques reduce turnaround times by up to 24%.[3] That is because a larger cache enables storing more prefixes between builds, so more configurations can be generated by extending cached prefixes and also cached prefixes can be more often reused to execute test plans for later builds. On the other hand, for *test-diff*, we see few benefits from increased cache size. These results are consistent with results reported in our previous study [15], that little benefit was observed beyond a cache size of 8 for InterComm. Also, as reported in the study, turnaround times decreased by almost half as the number of machines doubles.

We also observed that the benefits from the optimization techniques decrease as more machines are employed. With 4 machines, the turnaround time decreases by 24% when the number of cache entries per machine increases from 4 to 128, but decreases by only 10% when 32 machines are used. There are two reasons for this effect. First, with more machines the benefits from the increased computational power offset the benefits that are obtained via intelligent cache reuse. With 32 or more machines, parallel test execution enables high performance even with only 4 cache entries per machine. Second, communication cost increases

[3]For our subject component, turnaround times did not decrease further with more than 128 cache entries per machine.

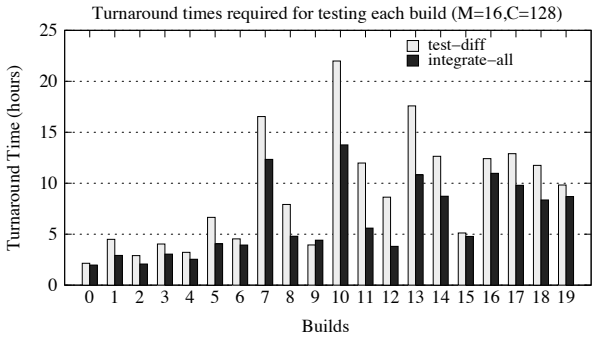


Figure 10: *test-diff* vs. *integrate-all*. There are significant cost savings for some builds from the optimization techniques. (16 client machines, 128 cache entries per machine)

for more machines, because each machine is responsible for fewer nodes in the test plan and machines that finish their work faster will take work from other machines. In many cases, the best cached configurations for the transferred work must be sent over the network.

As we previously noted, the cost savings vary depending on the component changes between two builds. In Figure 10, we compare turnaround times for each build, for *test-diff* and *integrate-all*. We only show results for 16 machines, each with 128 cache entries, but the overall results were similar for other machine/cache size combinations.

In the figure, we see significant cost reductions for several, but not all, builds (1, 5, 7–8, 10–13). We found that for those builds there were either new version releases or source code updates for InterComm, the top component in the CDG. Since we have to first build all other components before building InterComm, we can significantly reduce the execution time for the builds of interest by first extending configurations that took more time to build in the process of the configuration generation, and then reuse those configurations during test plan execution. From the results, we see a decrease of more than 50% in build time for builds 11 and 12 and a 40% time reduction on average for the other builds of interest.

The optimization techniques are heuristics, and do not always reduce testing time much. There were smaller cost reduction for builds 0–4 and 14–19. There are several reasons for that. First, test plans for builds 0–4 contain fewer nodes than for other builds, and therefore the plan execution times are dominated by the parallel computation. Second, for builds 14–19, as seen in Table 2, there were no changes for InterComm or for other components close to the top node in the CDGs. Although the test plan sizes for those builds, seen in Table 3, were comparable to those for other test plans where we achieved larger cost savings, for these builds we could only reuse shorter prefixes that can be built quickly from an empty configuration, because changes are confined to components close to the bottom node in the CDGs.

4.6 Comparing Optimization Techniques

Figure 11 shows turnaround times for testing each build using 16 machines, with cache sizes of 4 (top) and 128 (bottom) per client machine. We only show results for builds for which we saw large benefits in Figure 10, when both optimization techniques are applied. For each build of interest,

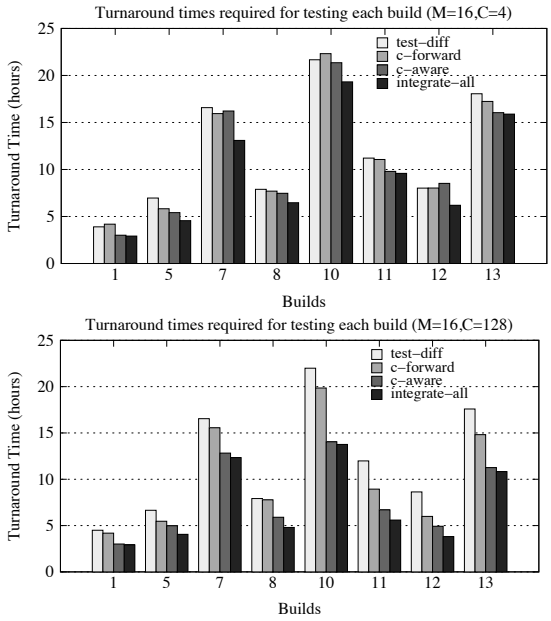


Figure 11: Each optimization technique contributes differently for different cache sizes.

we show results for four cases – *test-diff*, *c-forward*, *c-aware* and *integrate-all*.

In both graphs, we do not see large time decreases from simply forwarding cached configurations across builds (*c-forward*), even for a large cache. This implies that we must utilize cached prefixes intelligently. For the *c-forward* case, whether cached prefixes are reused or not depends on the order in which the *DD-instances* in the test plans for subsequent builds are executed, and the order in which configurations are cached and replaced.

With a smaller cache size, benefits from the optimization techniques are limited because configurations cached from earlier builds often get replaced before they are needed in later builds. However, we still see a small cost reduction by keeping the most valuable configurations in the cache.

In the bottom graph, with cache size 128, we observe that the cache-aware configuration generation technique (*c-aware*) plays a major role in reducing turnaround times. A larger cache can hold more prefixes for reuse, and therefore fewer cache replacements are necessary, and also we can extend cached prefixes with a few additional *DD-instances*. Consequently, it takes less time to execute the resulting test plans. In both graphs, the new cache management policy did not greatly decrease test plan execution time. Since our scheduling policy searches the test plans mostly in depth-first order, in many cases, the least recently used prefixes in the cache were also less valuable for the new policy.

In the simulation with 16 machines, each with 128 cache entries, there was no cache replacement for the entire build sequence. We still observe some additional cost reduction for *integrate-all*, compared to the *c-aware* case. We believe that the benefit is from synergy between the scheduling policy for dispatching prefixes to client machines and the new cache management policy.

5. RELATED WORK

Work on regression testing and test case prioritization [2, 3, 13] attempts to reduce the cost to test software systems

when they are modified, by selecting tests that were effective in prior test sessions and also by producing additional test cases if needed. *Rachet* has similar goals in its attempts to reduce test cost as components in a system evolve over time. In particular, Qu et al. [8] applied regression testing to user configurable systems and showed that a combinatorial interaction testing technique can be used to sample configurations. Robinson et al. [9] presented the idea of testing user configurable systems incrementally, by identifying configurable elements affected by changes in a user configuration and by running test cases relevant to the changes. Although their basic idea is similar to our work, those approaches are applied only to a flat configuration space, not for hierarchically structured component-based systems [8], or they only test modified configurations after a user has changed a deployed configuration [9] without proactively testing the configuration space before releasing the system.

ETICS [6], CruiseControl [1] and Maven [5] are systems that support continuous integration and testing of software systems in diverse configurations, via a uniform build interface. Although these systems can be used to test the component build process, their process is limited to a set of predetermined configurations. *Rachet* instead produces configurations dynamically, considering available component versions and dependencies between components.

Virtual lab manager [11, 12], developed by VMware, can be used to test the build process for a component in various configurations. However, configurations must be manually customized by building each configuration in a virtual machine. Our approach can test compatibilities between components without any intervention from developers after they model the configuration space for their components.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an approach to support incremental compatibility testing for component-based systems that contain multiple components that can independently evolve over time. As part of the approach, we have defined test obligations for testing system builds by capturing the difference across builds and described algorithms for sampling and testing configurations that test only the difference, while reusing test results obtained in prior builds. We also presented methods that make proactive use of configurations cached in prior builds to further reduce the time required for compatibility testing. The results of our empirical study show large performance improvements from incremental testing. In addition, we showed that our cache-aware optimization methods can often reduce test time significantly.

In the future, we plan to investigate methods to automatically extract component dependencies directly from source code and build instructions, thereby enabling developers to more easily create configuration space models that can be used as input to the *Rachet* test process. We are also working to release the Rachet tool to the wider community and are beginning to investigate techniques for compatibility testing for functionality and performance.

Acknowledgments

This research was supported by NSF grants #CCF-0811284, #ATM-0120950, #CNS-0855055, and #CNS-0615072, DOE grant #DEFC0207ER25812, and NASA grant #NNG06GE 75G.

7. REFERENCES

[1] http://cruisecontrol.sourceforge.net.

[2] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, Feb. 2002.

[3] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, Apr. 2001.

[4] J.-Y. Lee and A. Sussman. High-performance communication between parallel programs. In *Proceedings of 2005 Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models*, Apr. 2005.

[5] V. Massol and T. M. O'Brien. *Maven: A Developer's Notebook*. O'Relilly Media, 2005.

[6] A. D. Meglio, M.-E. Bégin, P. Couvares, E. Ronchieri, and E. Takacs. ETICS: the international software engineering service for the Grid. *Journal of Physics: Conference Series*, 119, 2008.

[7] B. O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly Media, first edition, 2009.

[8] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 75–86, Jul. 2008.

[9] B. Robinson and L. White. Testing of user configurable software systems using firewalls. In *Proceedings of the 19th International Symposium on Software Reliability Engineering*, pages 177–186, Nov. 2008.

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

[11] VMware. Accelerating test management through self-service provisioning - whitepaper, 2006.

[12] VMware. Virtual lab automation - whitepaper, 2006.

[13] W. E. Wong, J. R. Horgan, S. London, and H. A. Bellcore. A study of effective regression testing in practice. In *Proceedings of the 8th International Symposium on Software Reliability Engineering*, 1997.

[14] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Direct-dependency-based software compatibility testing. In *Proceedings of the 22th IEEE/ACM International Conference On Automated Software Engineering*, Nov. 2007.

[15] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Effective and scalable software compatibility testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 63–74, Jul. 2008.

[16] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Prioritizing component compatibility tests via user preferences. In *Proceedings of the 25th IEEE International Conference on Software Maintenance*, Sep. 2009.