Information and Software Technology 55 (2013) 445–458

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Testing component compatibility in evolving configurations

Ilchul Yoon [b,*], Alan Sussman [a], Atif Memon [a], Adam Porter [a]

[a] Department of Computer Science, University of Maryland, College Park, MD 20742, USA
[b] Department of Computer Science, State University of New York, Incheon, Republic of Korea

ARTICLE INFO

Article history:
Available online 8 October 2012

Keywords:
Incremental testing
Software component
Compatibility

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—in use can be very large. Moreover, components are constantly being enhanced and new versions are being released. Component developers, therefore, spend considerable time and effort doing compatibility testing—determining whether their components can be built correctly for all deployed configurations–both for existing active component versions and new releases. In previous work we developed *Rachet*, a distributed, cache-aware mechanism to support large-scale compatibility testing of component-based software with a fixed set of component versions.

In this paper, we observe that it is too expensive to perform compatibility testing from scratch each time a new version of a component is released. We thus add a new dimension to *Rachet*: to perform incremental and prioritized 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. Because testers are often interested in focusing test effort on newly released and modified components and their versions, we have developed a prioritization mechanism that enhances compatibility testing by examining the configurations that test new or modified component versions first, while also distributing the work over a cluster of machines. We evaluate our approach using the 5-year evolution history of a scientific middleware component. Our results show that our methods can increase performance significantly over *Rachet*'s previous retest-all approach and also tests important component compatibilities early in the overall testing process, making the process of compatibility testing practical for evolving components.

1. Introduction

Over the last two decades software engineering researchers have studied methods to reduce incompatibilities between software components, and those methods are embodied in technologies such as component models, component interconnection standards and service-oriented architectures [4,6,10,11,13,19]. Despite these efforts, testing modern software components is still a challenging task for many reasons. One particular challenge is that components may be built in many *configurations*. Consider, for example, InterComm [14,24], 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 of the others, and each configuration may need to be rebuilt and retested after each change. Developers may have limited time and resources to perform compatibility testing.

In prior work, we have addressed some of these challenges by creating *Rachet* [30–32], 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* uses a test adequacy criterion and algorithms for generating a set of configurations that satisfy the criterion. Finally, *Rachet* efficiently tests the selected configurations, distributing the *build* effort across a grid of computers, caching and reusing partially built configurations whenever possible. In our study, we used independently compilable and deployable software libraries or tools (e.g., GNU compiler and GMP library) as components in our experiments. Although we believe that our

* Corresponding author. Tel.: +82 32 626 1213.
E-mail addresses: icyoon@sunykorea.ac.kr (I. Yoon), als@cs.umd.edu (A. Sussman), atif@cs.umd.edu (A. Memon), aporter@cs.umd.edu (A. Porter).

0950-5849/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2012.09.010

model used to encode configuration space of a component-based system is domain-neutral and can be used for other systems based on components with different granularity (e.g., Jar files), the current *Rachet* architecture that leverages virtual machines to represent configurations fits better with components that take long time to build and deploy.

Since the original design of *Rachet* did not accommodate component evolution, when new components or new versions of existing components were introduced *Rachet* simply retested all configurations, including the ones that had been covered in previous test sessions. To test component compatibilities incrementally, taking into account component evolution, in [29] we presented methods to rapidly test configurations that contain new or modified components, and evaluated the effectiveness of our approach through simulations with 20 actual builds for the InterComm component, developed over a 5-year period. The experimental results showed that the incremental approach is more efficient than the previous *retest-all* approach, and can produce results for new compatibilities early in the overall test process. More specifically, the paper made 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 decrease testing time.

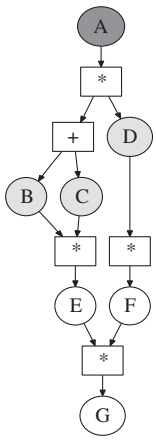
Although testing configurations incrementally can improve the performance of *Rachet*, as we reported in [29], the configurations tested may contain few compatibilities related to new components and their versions, but must contain many compatibilities to build the components required to test the new compatibilities. In this case, incremental testing will take longer to identify compatibilities for new components. In order to alleviate this problem, we have extended our work on incremental compatibility testing and make the following new contributions in this paper:

- A mechanism to prioritize the test order of configurations to discover test results for new compatibilities rapidly.
- An evaluation of the prioritization mechanism.

The next section provides an overview of *Rachet*. Section 3 defines a test adequacy criterion for incremental compatibility testing, and also describes algorithms to generate configurations for incremental and prioritized compatibility testing. Section 4 presents the results of our empirical study. Main extensions from our previous work [29] are described in Sections 3.4 and 4.7. Section 5 describes related work and Section 6 concludes with a brief discussion of future work.

2. Rachet overview

We summarize our prior work on *Rachet* [30–32], which is both a process and an infrastructure to perform compatibility testing for component-based software systems. 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 Fig. 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



Version Annotations

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

Constraints

$(ver(C) == C_2) \longrightarrow (ver(E) \geq E_3)$

Fig. 1. Example configuration model consisting of a CDG and annotations.

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 during 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 Fig. 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.

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 Fig. 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 minimizing 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 the selected *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 generated (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).

As described above, in the process of generating a configuration to cover a *DD-instance* of a component c in a CDG, the **BuildCFG** algorithm backtracks when the configuration is extended by adding a *DD-instance* of another component that violates any constraints. This means that, in the worst case, the algorithm could explore every combination of *DD-instances* of all other components (except the component c), before it returns a *valid* configuration. Therefore, for n components $c_1, c_2, \ldots, c_n$ in a CDG with the *DD-instance* sets $DDI_1, DDI_2, \ldots, DDI_n$, where c_1 is the top component of the CDG and DDI_i is the *DD-instance* set of the component c_i, the worst-case time complexity of the **BuildCFG** algorithm applied to generate a configuration to cover a *DD-instance* in the set DDI_1 is $\prod_{i=2}^{n}|DDI_i|$, where $|DDI_i|$is the number of *DD-instances* in the set DDI_i. However, in practice, constraints may be defined between components with direct dependencies, and such constraints do not make **BuildCFG** backtrack because they are enforced when we compute the set of *DD-instances*.

The **BuildCFG** algorithm is applied first to generate a configuration that covers a *DD-instance* for a component higher in a CDG – the component at depth 0 is the top component, since the algorithm may then choose more *DD-instances* that have not been covered by any other configuration in the recursive process, and as a result, we can minimize the total number of configurations generated. Fig. 2 illustrates the process of generating a configuration to cover a *DD-instance* for component A, (A_1,{B_1,D_1}). Starting from the leftmost sub-graph, the figure shows *DD-instances* selected for the configuration. For the example model in Fig. 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 ith component in the sequence does not depend on any component with an index greater than i. Therefore, *Rachet* can build the 1st component in the sequence, then build the 2nd 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. Fig. 3 depicts a test plan for the running 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 test 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 client machines across the network. In previous work [31], we examined three different test plan execution strategies, where each uses a different method to select the next prefix to be distributed to a client machine that performs the build: a *depth-first* strategy, a *breadth-first* strategy and a *hybrid* strategy. Our results showed that the *hybrid* strategy generally performed best across a wide variety of execution scenarios. In that 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 building the component encoded by the *DD-instance* was successful, (2) tested and building the component failed, or (3) was untestable, meaning that there was no way to produce a configuration to test that *DD-instance*. For example, suppose that in testing for our example, all attempts to build B_2 with E_1, E_2, and E_3 fail. Then, two *DD-instances* of component A, (A_1,{B_2,D_1}) and (A_1,{B_2,D_2}), are untestable because the *DD-instances* require a successfully built B_2 to build A_1.

3. Incremental testing

Our previous work cannot be directly used to efficiently test evolving software systems, because *Rachet* will generate configurations that test *all DD-instances* for the components in a model after any component changes, so the set can include unnecessary configurations that only test *DD-instances* whose results are already known from previous testing sessions. To support

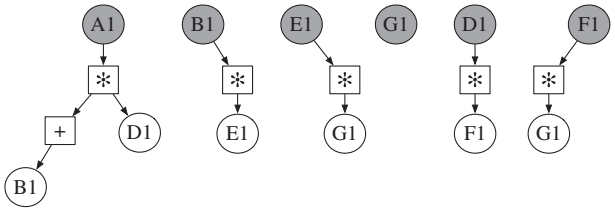


Fig. 2. Applying the **BuildCFG** algorithm to cover a *DD-instance* for component *A*.

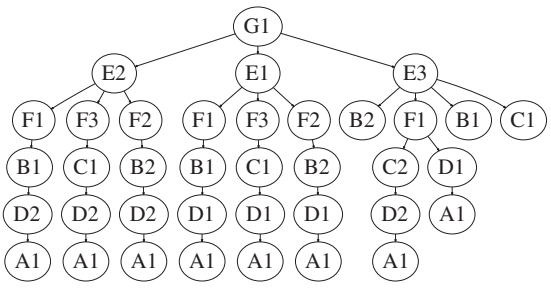


Fig. 3. Test plan with DD-coverage. (Dependency part of each *DD-instance* is omitted.)

incremental compatibility testing, we have extended *Rachet* to first identify the set of *DD-instances* that need to be tested given a set of component changes, compute a minimal set of configurations that cover those *DD-instances*, and order the test executions so that new and modified component versions get tested first. These extensions required us to develop new mechanisms to (1) compute incremental test obligations, (2) reuse cached configurations from previous test sessions, (3) manage cached configurations across test sessions, and (4) order the test executions so that newly modified components and their versions get tested first (we call this *prioritizing* the test order). We now describe each of these mechanisms.

We continue to use our running example from Fig. 1 to explain our algorithms. Suppose that during the previous testing session, $\mathtt{B}_2$ could not be built over any version of component E. As a result, all *DD-instances* in which component A must be built over $\mathtt{B}_2$ have not yet been tested. Now suppose that new versions of components B and D become available, and that the latest version of E, $\mathtt{E}_3$, 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 $\mathtt{B}_3$ and $\mathtt{D}_3$. Next, the modified component is handled by removing the old version, $\mathtt{E}_3$, and then adding a new version, $\mathtt{E}_4$. For this example, the original version of *Rachet* would produce a test plan with 56 component versions to build (Fig. 4a), which is larger than necessary because some configurations involve only *DD-instances* whose tests results are already known.

3.1. Computing incremental test obligations

The types of changes to the configuration models include adding and deleting (1) components, (2) component versions, (3) dependencies and (4) 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. Fig. 5 shows the 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.

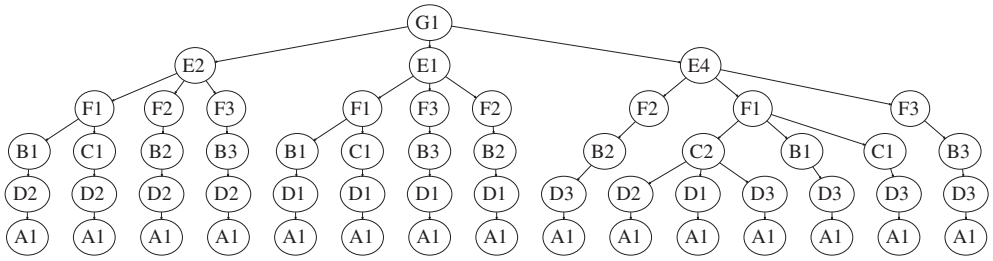
Using this set view, the *DD-instances* that must be tested for $build_i$ are shown as the shaded area in the figure, and are computed as follows:

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

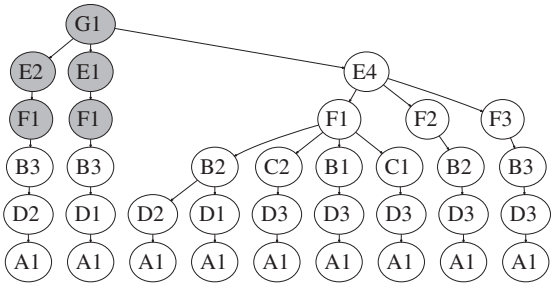
We include previously untestable *DD-instances* in the current testing obligations, since newly introduced component versions might provide new ways to build a given component, thus enabling previously untestable *DD-instances* to be tested.

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 compared to not taking into account prior test results. An outline of this process is as follows:

1. Compute DD^{i}_{test}.
2. Select the *DD-instance* from DD^{i}_{test} that builds 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.



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



(b) An incremental test plan

Fig. 4. Test plans: retest-all (56 components) vs. incremental (35 components). The shaded nodes can also be reused from the previous test session.

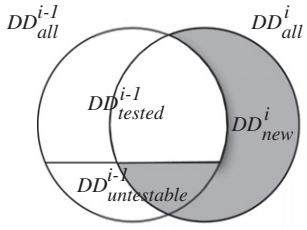


Fig. 5. *DD-instances* for two consecutive builds, $build_{i-1}$ and $build_i$. The *DD-instances* represented by the shaded areas need to be tested in $build_i$.

For the running example, the new algorithm produces nine configurations, reducing the test plan size from 56 (Fig. 4a) to 35 components (Fig. 4b). 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 five components depicted by shaded nodes in Fig. 4b have already been built in partial configurations and the configurations were cached in the previous test session (assuming those partial configurations were not deleted at the end of the test session).

3.2. Cache-aware configuration generation

In our previous work [30–32], we assumed that the cache space in each client test machine is empty at the beginning of a test session. For incremental testing, however, previous tests effort can and should be reused. On the other hand, just preserving the cache between test sessions may not 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 share 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 step 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 Fig. 6, the sub-tree 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 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.

Fig. 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 (Fig. 4b). 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 1.3 components compared to the cache-unaware plan, because almost half of its configurations are extended from cached prefixes. Shorter build sequences can greatly decrease the overall time needed to execute the test plan.

3.3. Managing cached configurations

Because cache space is a limited resource, when the cache is full we must discard a previously cached prefix to add a new one. In previous work [30–32], 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 in a test plan 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 savings from reusing the prefix to

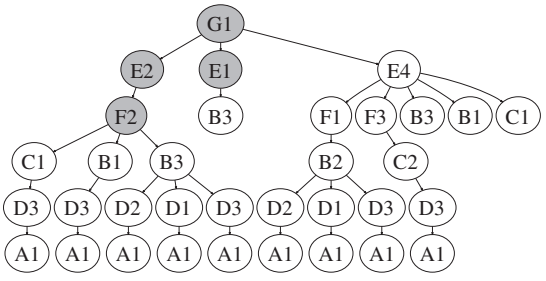


Fig. 6. Test plan produced from configurations selected in a cache-aware manner. Thirty-four component versions must be built. (Shaded area is cached from the previous test session.)

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 Fig. 6, prefixes <$\mathtt{G}_1$, $\mathtt{E}_2$> (call that p_1) and <$\mathtt{G}_1$, $\mathtt{E}_2$, $\mathtt{F}_2$> (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 in the cache after they are no longer needed.

3.4. Prioritizing configuration test order

During compatibility *retesting*, i.e., rerunning the *Rachet* process after one or more components have changed, a test plan for a build can contain *DD-instances* whose results have already been determined from testing prior builds. If a developer wants to obtain compatibility results for new *DD-instances* first, the developer may use a greedy approach that first tests the configurations that contain the most new *DD-instances*. However, because test plan execution in *Rachet* is distributed and cache-aware, this greedy approach does not take advantage of reusing partial configurations cached in multiple client machines. We thus designed an algorithm that dynamically prioritizes configurations based on the *freshness* of the configurations at any point in time during the plan execution. At each client's request, we apply the algorithm to determine the next configuration to be tested.

Algorithm 1. Schedule a configuration, considering both the freshness and the locality of the configuration to be tested.

Algorithm **Prioritized-Incremental-Execution**(*Plan, M*)
1: // *Plan*: the test plan being executed
2: // *M*: the client machine requesting a configuration to test
3: *CandidateCfgs* ← an empty list
4: **for** each configuration *c* in *Plan* **do**
5: $p \leftarrow$ the longest cached prefix of *c*
6: $newddi_c \leftarrow$ the number of new, not yet assigned, and not yet tested *DD-instances* in *c*
7: $ddi_c \leftarrow$ the number of *DD-instances* in *c*, excluding the ones in *p*.
8: $freshness \leftarrow newddi_c/ddi_c$
9: *local* ← 1 if *p* is already cached in *M*, and 0 otherwise.
10: $m \leftarrow$ the number of clients where *p* is cached
11: $CandidateCfgs \leftarrow CandidateCfgs \cup \{\langle c, freshness, local, m\rangle\}$
12: **end for**
13: Sort *CandidateCfgs* by freshness in decreasing order.
14: **return**, the first configuration in *CandidateCfgs*, using *local* and *m* as the first and second tie-breaker, respectively.

Algorithm 1 gives a high-level description of the selection process. When a client requests a new configuration to test, we evaluate the *freshness* of each configuration at that time by computing the ratio of the number of new, not yet assigned, and not yet tested *DD-instances* in the configuration to the number of *DD-instances* in the configuration, excluding the ones contained in the longest cached prefix of the configuration. A configuration (call it c_1) is not considered as a good candidate to be tested by the requesting client, if (1) c_1 contains many *DD-instances* whose results are already determined, or (2) another configuration (call it c_2) that shares a prefix with c_1 is already being tested by other client. The rationale behind this heuristic is that c_1 could be tested later with less test effort by reusing the shared prefix produced by the client testing c_2.

Algorithm 1 first build a list of candidate configurations (*CandidateCfgs*) that can be tested by the requesting client. Each configuration is added into the list, annotated with the freshness and two auxiliary values: (1) *local* is a boolean value that indicates whether the requesting client has the longest prefix of the configuration in its cache. When multiple client machines are employed to execute a test plan, this information can be used to increase the locality of the plan execution, taking advantage of reusing prefixes cached locally in each client. (2) *m* is the number of clients that have the longest prefix of the configuration in their cache. This is used to

reduce redundant work across multiple clients, when there are multiple candidate configurations with identical freshness and *local* values.

Although *local* and *m* are used to increase the locality of the plan execution and also cause each client to test non-overlapping regions of the test plan, a more important concern for developers is to test new *DD-instances* early in the test plan execution. Therefore, we sort the entries in *CandidateCfgs* by the configuration freshness in decreasing order, using *local* and *m* as the first and second tie breakers, respectively, and then dispatch the configuration that is the first entry in the sorted list to the requesting client.

4. Evaluation

Having developed a foundation for incremental and prioritized compatibility testing between evolving components, we now evaluate 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 [14,24]. To provide this functionality, InterComm relies on several 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 documentation that describes build sequences, dependencies and constraints for each component.[2]

Fig. 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, entries in the last eight 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 six 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

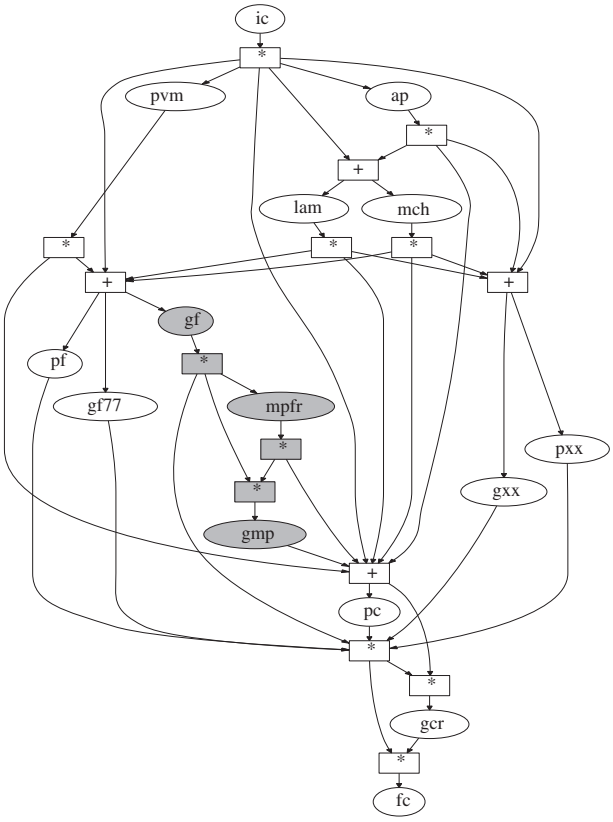


Fig. 7. CDG for InterComm.

Table 1
Components used in the InterComm model.

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

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

[2] We spent about (1) a month to identify dependencies and constraints between components and their versions used in any of the 20 builds and to integrate the information into the models for individual builds; (2) another month to develop tools that download components from different types of source code repositories. However, the efforts could be greatly reduced if the dependency and constraint information are provided clearly in a uniform way.

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

Build	Date	Branches						Released versions							
		ic	gcr gxx	gf77	gf	gmp	mpfr	ic	gcr gxx	gf	gmp	mpfr	lam	pvm	
0	08/25/04		3.4d1	3.4d1				1.1	3.4.0	3.4.0				6.5.9	3.2.6
									3.4.1	3.4.1				7.0.6	3.3.11
1	11/25/04	1.1d1	3.4d2	3.4d2					3.4.2	3.4.2					3.4.5
									3.4.3	3.4.3					
2	02/25/05		3.4d3	3.4d3											
3	05/25/05		3.4d4, 4.0d1	3.4d4	4.0d1				3.4.4	3.4.4	4.0.0	4.1.0, 4.1.1	2.1.0		
									4.0.0			4.1.2, 4.1.3	2.1.1		
												4.1.4			
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	3.4d8	4.0d5, 4.1d1			1.5	3.4.6	3.4.6	4.0.3	4.2.0,4.2.1			
			4.0d5, 4.1d1						4.0.3		4.1.0				
									4.1.0		4.1.1				
									4.1.1						
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			

has changed from its state in the previous build.[3] For example, `l.ld2` in the third column of $build_5$ indicates that there were file changes in the branch `l.ld` 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. For example, at $build_7$, three major versions (`3.4`, `4.0`, `4.1`) of GNU compilers were actively maintained, and four minor revisions were released for the major versions, and there were changes in their development branches. With these changes, InterComm developers had to make sure that the new InterComm version `l.5` released at $build_7$ was build-compatible with the compiler versions available, and also they had to test whether the old InterComm version `l.l` was compatible with new compiler versions.

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 obtain only one version or we considered only one version to limit the time to perform the experiments (`fc`). For this study, we assumed that these versions were available from the first build. Also, we considered development 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 and prioritized testing approach, we first gathered component compatibility results (i.e., success or failure to build the component version encoded by each *DD-instance*) and the time required to build each component version. To collect 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 2.4 GHz CPU with 512 MB memory, running Red Hat Linux) and 32 client machines (Pentium 4 2.8 GHz Dual-CPU machines with 1 GB 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. *Rachet* currently supports the CVS, Subversion, and Mercurial [2,20] source code management systems.

By running the test plan for the integrated model, we obtained compatibility results for the 15,128 *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 a dependency on 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 just described, 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

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

total build times for each simulation. Table 3 shows the number of *DD-instances* corresponding to each region in the diagram in Fig. 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 $\left(DD^i_{test} = DD^i_{all}\right)$, (2) *test-diff*: tests builds incrementally $\left(DD^i_{test} = DD^i_{all} - DD^{i-1}_{tested}\right)$ but without reusing configurations cached in prior builds, (3) *c-forward*: *test-diff* plus reusing cached configurations across builds, but without applying any of our optimization techniques, (4) *c-aware*: *c-forward* plus applying cache aware configuration production (Section 3.2), (5) *integrate-all*: *c-aware* plus applying the improved cache management strategy (Section 3.3), (6) *prioritized*: *integrate-all* but with the prioritized plan execution strategy (Section 3.4). For the different incremental test strategies, we measured the turnaround time for each build in the sequence, and also measured the times at which test results for *DD-instances* are produced while executing the test plan for each build.

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. Fig. 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. The figure shows 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.

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

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

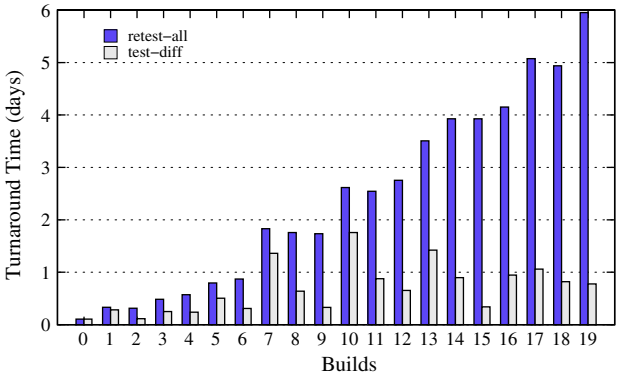


Fig. 8. Turnaround times for testing DD^i_{all} and DD^i_{test} for each build (eight machines and four cache entries per machine).

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 the generated configurations. For example, as seen in Table 3, the sizes of DD^i_{test} $\left(DD^i_{all} - DD^{i-1}_{tested}\right)$ for build 11 and build 15 are comparable (2957 for build 11 and 2941 for build 15), but the testing time required for build 11 is twice as long as that 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

Fig. 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 time. The simulations use 4–32 client machines and the number of cache entries per machine varies from 4 to 128.

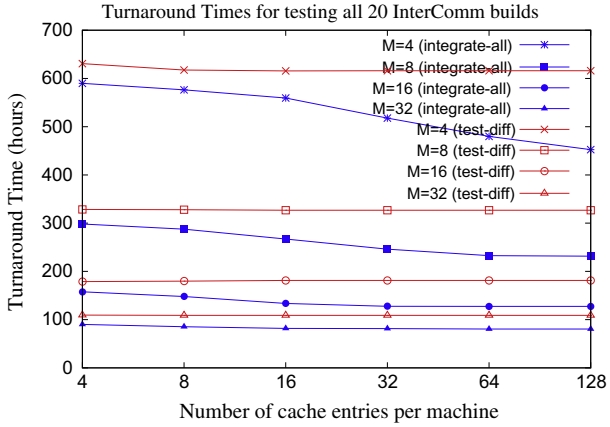


Fig. 9. As the number of cache entries per machine increases, overall test cost decreases up to 24% when optimization techniques are applied, compared to the baseline incremental test.

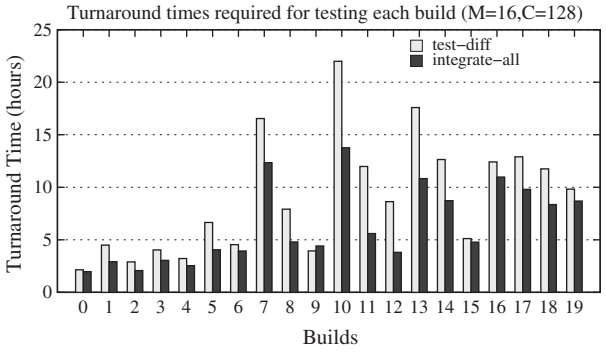


Fig. 10. *Test-diff* vs. *integrate-all*. There are significant cost savings for some builds from the optimization techniques (16 client machines, 128 cache entries per machine).

As the number of cache entries increases, we observe that the optimization techniques reduce turnaround times by up to 24%.[4] That is because a larger cache enables storing more prefixes between builds, so more configurations can be generated by extending cached prefixes, and cached prefixes can also 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 [31], that little benefit was observed beyond a cache size of 8 for InterComm. Also, as reported in that study, turnaround times decreased by almost half as the number of client machines doubles.

We also observed that the benefits from the optimization techniques decrease as more machines are employed. With four 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 four cache entries per machine. Second, communication cost increases 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 may vary, depending on the component changes between two builds. In Fig. 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 nodes in the CDGs.

4.6. Comparing optimization techniques

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

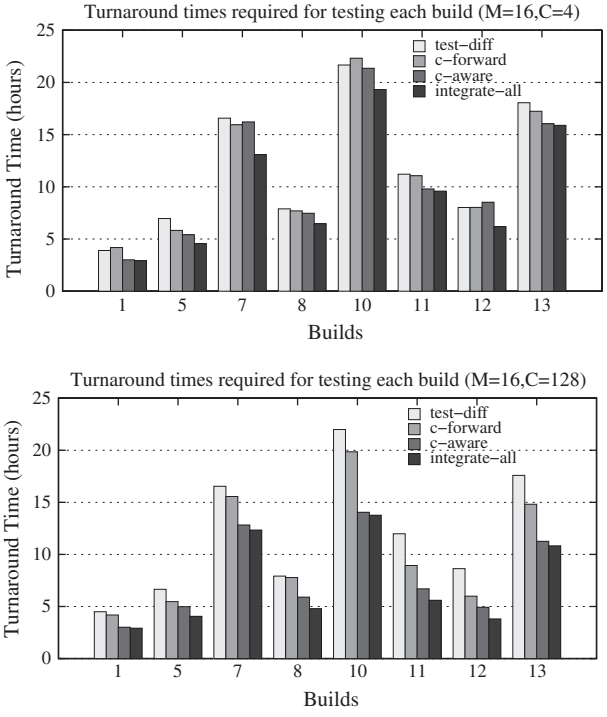


Fig. 11. Each optimization technique contributes differently for different cache sizes.

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

ing 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 *c-aware*. We believe that the benefit is from synergy between the scheduling policy for dispatching prefixes to client machines and the new cache management policy.

4.7. Evaluating the prioritization technique

Recall that our goal for prioritization is to test new *DD-instances* earlier in the test process, and we are interested in answering the question: "To what extent can the prioritization mechanism help to test new *DD-instances* earlier in the overall test process?". To answer this question, we compared our prioritized approach with the cost-based *integrate-all* strategy described in Section 4.3. Fig. 12 depicts how fast the prioritized plan execution (`incprior` in the graphs) tests new *DD-instances* compared to the cost-based plan execution (`Inc.` in the graphs). These graphs show the results for $build_1$ with 4, 8, 16 and 32 client machines, where each client can cache up to eight prefixes. Note that we normalized the *x*- and *y*-axes to allow easier comparison.

In the top-left plot, for four client machines, we see that the prioritized strategy tests more new *DD-instances* earlier, compared to `Inc`. Halfway through the entire test process, the prioritized strategy is able to test almost 85% of *DD-instances* compared to 48% with the cost-based strategy. The difference between the two strategies decreases as we increase the number of machines, because the cost-based strategy can benefit more from the increased parallelism. As described previously, with the prioritized strategy, at any plan execution point *Rachet* dispatches the freshest configuration to a requesting client, and in many cases the best cached prefix to test the configuration has to be transferred over the network before *DD-instances* in the configuration are tested.

We note that the plots contain several spikes, where the percentage of tested *DD-instances* jumps. This happened when *Rachet* failed to build a component version and there was no alternate way to build the component version. If the component version is a prerequisite to build other component versions, a large number of *DD-instances* can be simultaneously classified as *untestable*. For example, in the plot with four machines (top left in Fig. 12), 55 *DD-instances* are classified as untestable at 3.8% of the plan execution time, because *Rachet* failed to build `gf77` version `3.4.3`. This component was required to build other component versions encoded by the 55 *DD-instances*. The *integrate-all* approach tested configurations that contain the *DD-instance* later in the plan execution. That is, the prioritized strategy is effective at quickly identifying results for *DD-instances* that encode new component versions critical in building other components.

We also see that the prioritized strategy tested new *DD-instances* faster for build 7, 10, 13, and 16–19, even with 32 machines. This is because the test plan sizes for the builds are large, compared to other builds, as seen in Table 3. This implies that the prioritized execution may perform better as the plan size grows. We believe that this will be very helpful when a test plan involves many components and component versions that evolve over time.

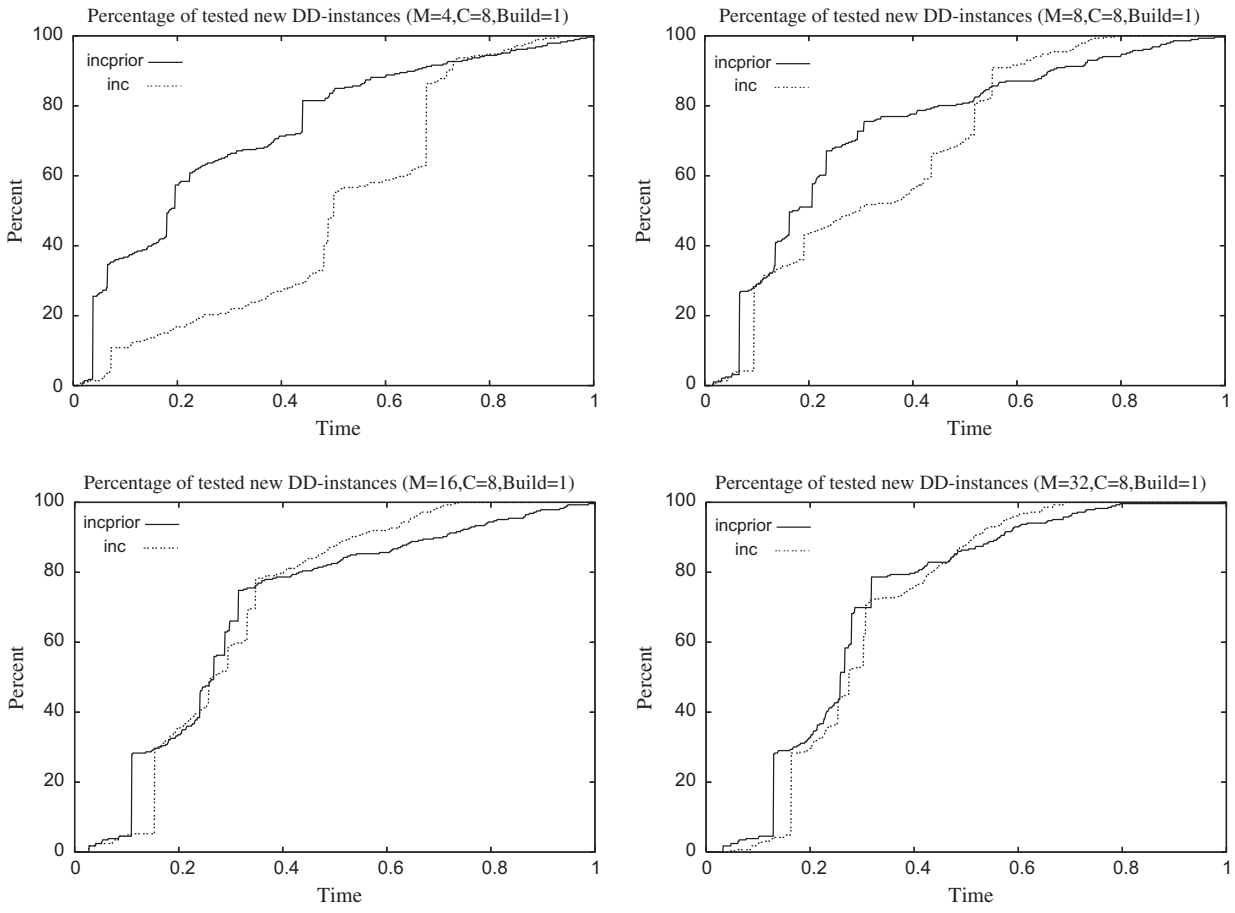


Fig. 12. Prioritized plan execution identifies results for new *DD-instances* quickly, compared to cost-based plan execution.

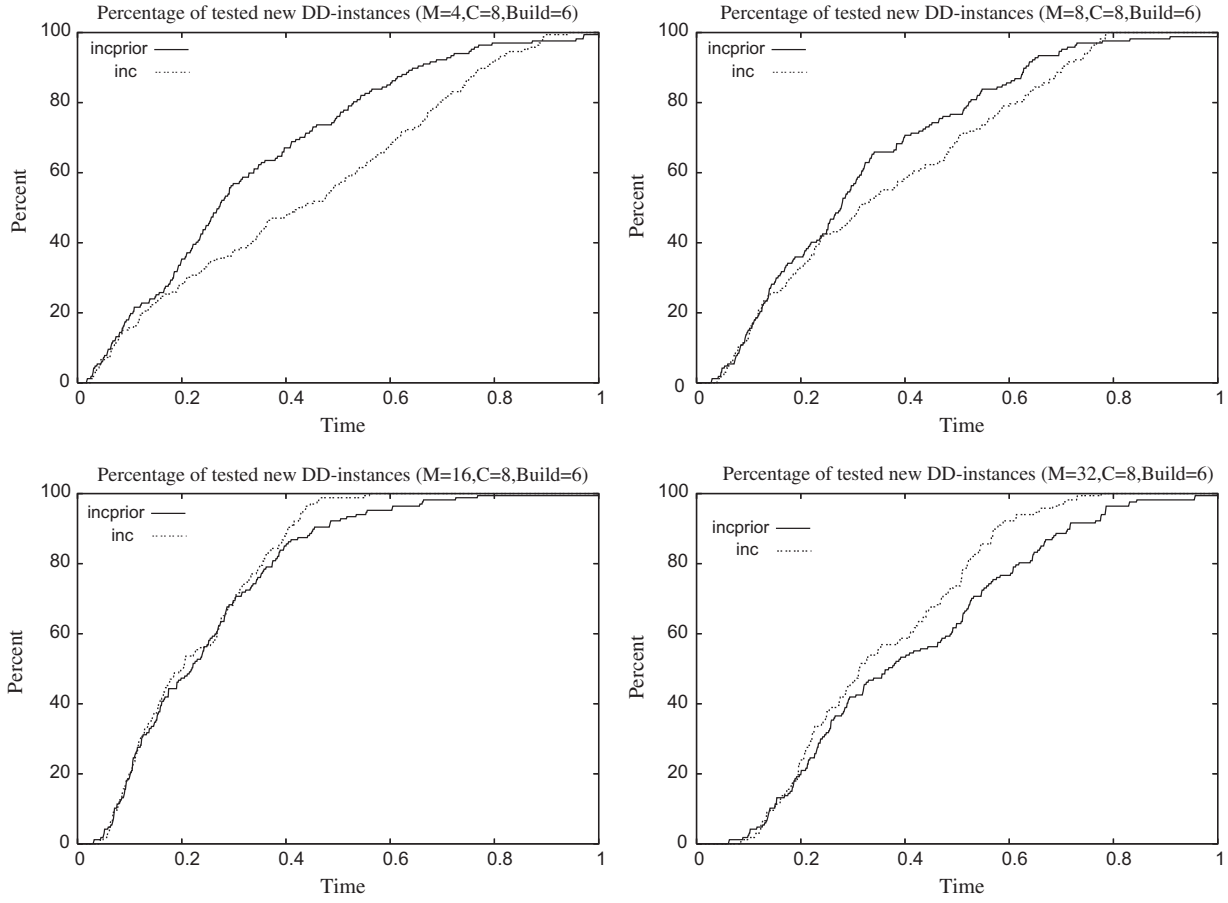


Fig. 13. Percentage of *DD-instances* with prioritized plan execution for $build_6$ (untestable *DD-instances* are excluded).

To see the performance of the prioritized execution without the effects of untestable *DD-instances*, in Fig. 13 we show results for $build_6$ that exclude untestable *DD-instances*. We see that with a small number of machines the prioritized strategy is able to classify more *DD-instances* in DD^6_{test} as successes or failures compared to the *integrate-all* approach. For the subject system, on average, 82% of plan nodes in the initial test plans for 20 InterComm builds represent *DD-instances* related to new or modified components. Therefore, we do not see a large difference between the prioritized and cost-based execution strategies. However, as seen in Fig. 12, the prioritized strategy is more likely to discover build failures early in the plan execution when the failures are related to components required for building many other components.

To quantitatively measure the benefit from the prioritized approach, we use a metric similar to the weighted average of the percentage of faults detected (APFD) [23]. We compute the differences between the computed APFDs of the two strategies, as we vary the number of machines, where each machine can cache up to eight prefixes. We also did not consider *DD-instances* classified as untestable. The area differences are depicted in Fig. 14. When four machines are used, we see that the prioritized strategy has a higher APFD (up to 12% more), but the differences decrease when more machines are employed.

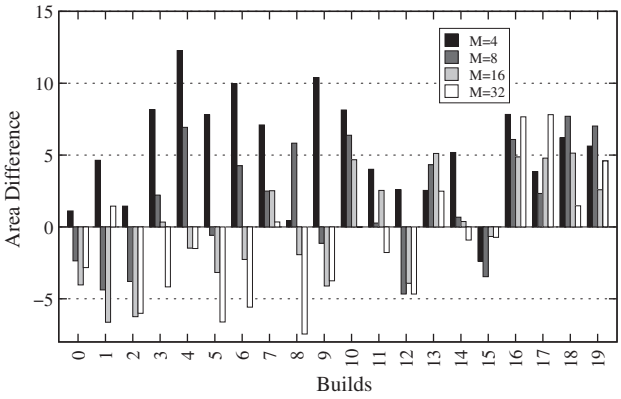


Fig. 14. APFD area difference between the prioritized and cost-based plan execution strategies ($C = 8$).

4.8. Threats to validity

Like other empirical studies, our study also has internal and external validity issues that should be considered when practitioners evaluate the results presented in this paper.

The representativeness of the subject system is a threat to external validity. We used InterComm as our subject system because it is intended to be successfully built on a wide range of configurations that involve many coarse grained components with multiple active versions. All the components can be built separately and their build processes can take a long time. Although we believe that our results would be valid for other systems based on coarse grained components, the results could be further validated with other component-based systems. In addition, there are other types of component-based systems for which we may need to modify our methods to test component compatibilities.

For example, if components in a system are fine grained (e.g., ActiveX components or scientific components conforming to the Common Component Architecture [3]), or the components do not take long to build, the overhead to manage virtual machines in *Rachet* could be too high.

The fixed set of options and their values we used to build components in the experiments can be considered as a threat to internal validity. For example, we considered an MPI component as a prerequisite to build InterComm, and this is true if we build InterComm with the '–with-mpi' option. However, users can disable the option if they do not need MPI support when they build InterComm. As described in [18], the component build process can be affected by enabling or disabling certain build options, and/or by setting different values for the options. Another internal validity issue is that the *CDG* and *Annotations* for the subject system are created by manually inspecting documents provided by component developers. It is possible that components can have undocumented dependencies or constraints, and that human errors occur in the modeling process.

5. Related work

Work on regression testing and test case prioritization [9,12,28] 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. [21] applied regression testing to user configurable systems and showed that a combinatorial interaction testing technique can be used to sample configurations. Robinson et al. [22] 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 [21], or they only test modified configurations after a user has changed a deployed configuration [22], instead of proactively testing the configuration space before releasing the system.

ETICS [17], CruiseControl [1] and Maven [16] 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 generates and tests configurations dynamically, considering available component versions and dependencies between components. *Virtual lab manager* [26,27], developed by VMware, can also 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.

Duarte et al. [7,8] describe a technique to improve confidence in the correctness of applications designed to be executed in heterogeneous environments. To test the correct build and execution of a software system in diverse field configurations, they distribute the software and its unit test suites onto multiple heterogeneous machines accessible in a network of machines. The distributed software is built on the machines and test suites are run to test the behavior of the software. Although their work pursues a similar goal to ours, they do not analyze the configuration space of the evolving system by intelligently sampling configurations to effectively identify component compatibilities, but instead try to determine whether the software under test can be built successfully in a limited set of configurations explicitly selected by the testers.

Syrjänen [25], and Mancinelli et al. [5,15] studied methods to identify inconsistencies (e.g., missing packages or version conflicts) between components contained in GNU/Linux distributions. They used rule-based representations to describe dependencies and constraints between components in the distributions, and formulate the component installability as a satisfiability (SAT) problem. Their work can be used by distribution editors to check the installability of components in Linux distributions, based on a set of dependencies and constraints between components. Our approach is rather to help component developers to determine whether their components can be built in a wide range of possible configurations by testing component compatibilities for all components and their versions required to build a given component.

6. Conclusions and future work

In this paper we have presented an approach to support incremental and prioritized compatibility testing for component-based systems that contain multiple components that can evolve independently 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 from 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, and a method that prioritizes the configuration test order, by testing configurations starting from the one that returns the largest number of new compatibility results when testing time and resource are limited.

The results of our empirical study showed large performance improvements from incremental testing. In addition, we showed that our cache-aware optimization methods can often reduce test time significantly, and that compatibility results can be discovered rapidly with the prioritized plan execution method.

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 performing compatibility testing as part of both functional and performance testing.

Acknowledgments

This research was supported by NSF Grants #CCF-0811284, #ATM-0120950, #CNS-0855055, #CNS-1205501, and #CNS-0615072, DOE Grant #DEFC0207ER25812, and NASA Grant #NNG06GE75G.

References

[1] CruiseControl: an extensible framework to support continuous integration process. <http://cruisecontrol.sourceforge.net>.
[2] Mercurial: a distributed source control management tool. <http://mercurial.selenic.com>.
[3] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S.R. Kohn, L.C. McInnes, S.R. Parker, B.A. Smolinski, Toward a common component architecture for high-performance scientific computing, in: Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing, 1999, pp. 115–124.
[4] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, second ed., Addison-Wesley, 2003.
[5] J. Boender, R.D. Cosmo, J. Vouillon, B. Durak, F. Mancinelli, Improving the quality of GNU/Linux distributions, in: Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, 2008, pp. 1240–1246.

[6] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2001.
[7] A. Duarte, G. Wagner, F. Brasileiro, W. Cirne, Multi-environment software testing on the Grid, in: Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging, 2006, pp. 61–68.
[8] A.N. Duarte, W. Cirne, F. Brasileiro, P. Machado, GridUnit: software testing on the grid, in: Proceedings of the 28th International Conference on Software Engineering, May 2006, pp. 779–782.
[9] S. Elbaum, A. Malishevsky, G. Rothermel, Test case prioritization: a family of empirical studies, IEEE Transactions on Software Engineering 28 (2) (2002) 159–182.
[10] W. Emmerich, An introduction to OMG/CORBA, in: Proceedings of the 19th International Conference on Software Engineering, 1997, pp. 641–642.
[11] T. Erl, Service-Oriented Architecture (SOA): Concepts, Technology, and Design, Prentice Hall, 2005.
[12] T.L. Graves, M.J. Harrold, J.-M. Kim, A. Porter, G. Rothermel, An empirical study of regression test selection techniques, ACM Transactions on Software Engineering and Methodology 10 (2) (2001) 184–208.
[13] M.N. Huhns, M.P. Singh, Service-oriented computing: key concepts and principles, IEEE Internet Computing 9 (1) (2005) 75–81.
[14] J.-Y. Lee, 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, 2005, p. 177b.
[15] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, R. Treinen, Managing the complexity of large free and open source package–based software distributions, in: Proceedings of the 21th IEEE/ACM International Conference On Automated Software Engineering, 2006, pp. 199–208.
[16] V. Massol, T.M. O'Brien, Maven: A Developer's Notebook, O'Relilly Media, 2005.
[17] A.D. Meglio, M.-E. Bégin, P. Couvares, E. Ronchieri, E. Takacs, ETICS: the international software engineering service for the grid, Journal of Physics: Conference Series 119 (4) (2008).
[18] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D.C. Schmidt, B. Natarajan, Skoll: distributed continuous quality assurance, in: Proceedings of the 26th International Conference on Software Engineering, 2004, pp. 459–468.
[19] Object Management Group, Inc., The common object request broker architecture 3.0 specification, 2002.
[20] B. O'Sullivan, Mercurial: The Definitive Guide, O'Reilly Media, first ed., 2009.
[21] X. Qu, M.B. Cohen, G. Rothermel, Configuration-aware regression testing: an empirical study of sampling and prioritization, in: Proceedings of the International Symposium on Software Testing and Analysis, 2008, pp. 75–86.
[22] B. Robinson, L. White, Testing of user configurable software systems using firewalls, in: Proceedings of the 19th International Symposium on Software Reliability Engineering, 2008, pp. 177–186.
[23] G. Rothermel, R. Untch, C. Chu, M. Harrold, Prioritizing test cases for regression testing, IEEE Transactions on Software Engineering 27 (10) (2001) 929–948.
[24] A. Sussman, Building complex coupled physical simulations on the grid with InterComm, Engineering with Computers 22 (3–4) (2006) 311–323.
[25] T. Syrjänen, A rule-based formal model for software configuration. Technical Report A55, Helsinki University of Technology, Laboratory for Theoretical Computer Science, December 1999, pp. 1–74.
[26] VMware, Inc., Accelerating test management through self-service provisioning – whitepaper, 2006, pp. 1–5.
[27] VMware, Inc., Virtual lab automation – whitepaper, 2006, pp. 1–11.
[28] W.E. Wong, J.R. Horgan, S. London, H.A. Bellcore, A study of effective regression testing in practice, in: Proceedings of the 8th International Symposium on Software Reliability Engineering, 1997, pp. 230–238.
[29] I. Yoon, A. Sussman, A. Memon, A. Porter, Towards incremental component compatibility testing, in: Proceedings of the 14th International ACM SIGSOFT Symposium on Component Based Software Engineering, 2011, pp. 119–128.
[30] I.-C. Yoon, A. Sussman, A. Memon, A. Porter, Direct-dependency-based software compatibility testing, in: Proceedings of the 22th IEEE/ACM International Conference On Automated Software Engineering, 2007, pp. 409–412.
[31] I.-C. Yoon, A. Sussman, A. Memon, A. Porter, Effective and scalable software compatibility testing, in: Proceedings of the International Symposium on Software Testing and Analysis, 2008, pp. 63–74.
[32] I.-C. Yoon, A. Sussman, A. Memon, A. Porter, Prioritizing component compatibility tests via user preferences, in: Proceedings of the 25th IEEE International Conference on Software Maintenance, 2009, pp. 29–38.