ABSTRACT

Title of Dissertation: COMPATIBILITY TESTING FOR COMPONENT-BASED SYSTEMS

Ilchul Yoon, Doctor of Philosophy, 2010

Dissertation directed by: Professor Alan Sussman
Department of Computer Science

Many component-based systems are deployed in diverse environments, each with different components and with different component versions. To ensure the system builds correctly for *all* deployable combinations (or, configurations), developers often perform *compatibility testing* by building their systems on various configurations. However, due to the large number of possible configurations, testing all configurations is often infeasible, and in practice, only a handful of popular configurations are tested; as a result, errors can escape to the field. This problem

is compounded when components evolve over time and when test resources are limited.

To address these problems, in this dissertation I introduce a process, algorithms and a tool called Rachet. First, I describe a formal modeling scheme for capturing the system configuration space, and a sampling criterion that determines the portion of the space to test. I describe an algorithm to sample configurations satisfying the sampling criterion and methods to test the sampled configurations.

Second, I present an approach that incrementally tests compatibility between components, so as to accommodate component evolution. I describe methods to compute test obligations, and algorithms to produce configurations that test the obligations, attempting to reuse test artifacts.

Third, I present an approach that prioritizes and tests configurations based on developers' preferences. Configurations are tested, by default starting from the most preferred one as requested by a developer, but cost-related factors are also considered to reduce overall testing time.

The testing approaches presented are applied to two large-scale systems in the high-performance computing domain, and experimental results show that the approaches can (1) identify compatibility between components effectively and efficiently, (2) make the process of compatibility testing more practical under constant

component evolution, and also (3) help developers achieve preferred compatibility results early in the overall testing process when time and resources are limited.

COMPATIBILITY TESTING FOR COMPONENT-BASED SYSTEMS

by

Ilchul Yoon

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2010

Advisory Committee:

Professor Alan Sussman, Chairman/Advisor
Professor Adam Porter
Professor Atif Memon
Professor Ramani Duraiswami
Professor Derek Richardson

DEDICATION

To my wife – Heejong for her endless love and support.

ACKNOWLEDGEMENTS

It is my honor to know so many good people who helped me to complete this dissertation. First and foremost, I deeply thank my advisor, Prof. Alan Sussman. He guided me to set up my research direction and helped me with his deep insight to realize my rough ideas into concrete artifacts. He has always been generous to me and also encouraged me to be confident. I would also like to thank Prof. Adam Porter and Prof. Atif Memon. Throughout discussions with them, I could get many ideas to analyze and to present results effectively. Also, I have always been amazed whenever they touched the core of my monotonous manuscript and transformed it into a shiny piece of work in concise and polished sentences.

Thanks are due to Prof. Derek Richardson and Prof. Ramani Duraiswami for agreeing to serve on my dissertation committee and for

sparing their invaluable time to review my dissertation. They gave me excellent suggestions to improve the quality of the dissertation.

I owe my deepest thanks to my family - my wife, parents, parents-in-law, sister and brothers. I was able to finish this long journey because of their enormous love and support. Especially, it is impossible for me to fully express my gratitude to my wife, Heejong Sung, for her endless patience, advice, encouragement, faith, prayer and love.

During my years at University of Maryland, I had the pleasure of sharing my life with good people around me. I was so lucky to start my study with Jik-Soo Kim, Minkyoung Cho and Youngmin Kim. They always stood by me in joy and sorrow, and spent their time to discuss my problems. I will remember forever the moments we shared together. I am also grateful to my godfather, Prof. Kyu Yong Choi and the members of Darak-bang – Seong Sook Kim, Yeon Seok Kim, Ellen Kim, Won Joon Choi and Myoung Deok Shin and Hyo Jung Kim – for their prayers to God for me.

I would like to thank my lab colleagues, Jaehwan Lee, Sukhyun Song, Gary Jackson, Puneet Sharma and Teng Long. They attended my practice talks and helped me to improve the presentation for the de-

fense. I also would like to acknowledge Ananta Tiwari and my Korean colleagues: Jaeyong Lee, Seungryul Choi, Bongwon Suh, Hyunmo Kang, Joonhyuk Yoo, Jae-Yoon Jung, Beomseok Nam, Ji Sun Shin, Jikhyuk Jung, Hyunyoung Song, Jaehwan Lee, Sukhyun Song, Sungwoo Park, Eunhui Park, Chanhyun Kang, Hyuk Oh, Angela Song-Ie Noh, Tak Yeon Lee, Jaehwa Choi, Sunhee Kim, Sangchul Song and Min-Young Kim. Because of them, my graduate experience at University of Maryland has been one that I will cherish forever.

I also would like to acknowledge the financial support from the Institute of Information Technology in Korea during the early years of my study.

Lastly, I truly thank God for letting me know wonderful people and for all the achievements I have made during my study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Modern software systems are becoming increasingly large and complex, and little software is developed entirely from scratch. Instead, for building systems correctly, a majority of software systems requires (third-party) components such as libraries and tools [15, 17, 70]. One of the top concerns for developers of such software systems is to ensure that their systems can be built without any problem and behave as expected in field environments (or *configurations*) that might be realized in end-users' machines, which may contain different sets of components and their versions required for building the systems. If the systems are released with undetected incompatibilities between components, they can make users spend time for resolving such incompatibilities, and also make it difficult for developers to rationally manage support activities for the systems [16, 39].

To reduce latent incompatibilities between components, software researchers have developed methods and tools such as configuration management systems, interconnection standards, middleware frameworks and product-line and service-oriented architectures [13, 20, 36, 45]. However, despite these advances, it is still challenging to guarantee the compatibility of a system with the expected set of configurations, for several reasons.

First, the number of configurations on which a component-based system may build and execute can be enormous. A system may require multiple components each with multiple versions where each version depends on multiple third-party components, and each of these components in turn has multiple versions and dependencies on other components. In principle, each possible combination of these components is a configuration that some end-user might use, and in many cases, it is infeasible to test all possible configurations and therefore it is necessary to intelligently sample configurations from the vast set of feasible configurations and test them efficiently.

Another challenge is that each component can evolve independently. Component developers may release new versions of their components or modify dependencies to other components without any notice, especially if the components are developed and maintained by independent groups of developers. In the worst

case, a component change can mandate retesting the entire set of configurations. Given the high cost of testing, it is important not to waste time and money for testing compatibilities. However, considering that a large software system involves many components interconnected with complex dependencies, it is difficult for developers to identify configurations that are affected by component changes, and also past test results should be best utilized for saving test effort.

Moreover, the time and resources allowed for the compatibility testing can be limited. In such resource-constrained situations, developers want to see compatibility results they have the most interest early in the test process. Because it is impractical for developers to manually specify an ordering across all the configurations to be tested, and also because there is potentially a large number of such configurations, they need a prioritization mechanism that takes into account developers' preferences over components and their versions.

In practice, to identify configurations with which a component-based system is (in)compatible, developers have performed *compatibility testing* [49] by selecting a set of configurations – each configuration is an ensemble of component versions that respects known dependencies – and by testing whether their system builds and functions properly for each configuration. However, as described above, the large number of possible configurations and the lack of automated testing support

have limited developers to pare down the set to a handful of popular configurations [1, 6] or use only configurations that are already realized on test machines they can access [33, 34]. This implies that the software is released with nearly all of its possible field configurations untested. So costly errors can and do escape to the field. While it might appear that these issues could be avoided by radically restricting the set of supported configurations, in reality that could unacceptably restrict the potential user base.

The goal of this research is to investigate methods for performing compatibility testing of complex and evolving component-based systems in an effective and efficient manner. This involves (1) sampling a small set of configurations that *effectively* test compatibilities between components from the set of all feasible configurations of the systems, and (2) testing the sampled configurations *efficiently* on limited test resources.

To achieve this goal, I present in this dissertation a process, algorithms and a tool called *Rachet*. Developers can identify compatibilities between components by applying Rachet on a formal graph-based model that encodes the configuration space of a software system. Based on the model, Rachet can sample configurations that satisfy a test adequacy criterion, which is to test all *direct dependency relationships* between components in the model. Then, sampled configurations

are tested over a set of test resources by leveraging hardware virtualization technology that enables reusing partially-built configurations and not contaminating the state of test resources. This approach can be applied for performing the compatibility testing of a model that consists of a *fixed* set of components and their versions, and is extended in two directions.

First, I extend the test adequacy criterion for a fixed model to accommodate the evolution of components involved in a model. The extended criterion requires testing only compatibilities that involve components modified since the last test session. This dissertation also presents an algorithm that samples configurations satisfying the criterion and makes use of test results from past test sessions.

Second, I develop a method that prioritizes the order to test configurations taking into account developers' preferences, because different developers can have different interests over components and their versions (e.g., they may want to first see compatibility results related to recently released component versions). The method evaluates the priorities of sampled configurations based on the preferences specified by developers and then the configurations are tested from the most important one, thereby providing developers results of more importance early in the test process.

I evaluate the presented approaches through extensive experiments, and show

that developers can identify (in)compatibility between components effectively with less test effort and can efficiently utilize available test resources.

1.1 Motivating Applications

This research is originally inspired by the following applications.

InterComm InterComm[1] [47, 48, 64, 65] is a middleware library that supports coupled scientific simulations by redistributing data in parallel between data structures managed by multiple parallel programs. For example, a simulation studying the effect of solar weather patterns on cell phone performance in the U.S. might involve multiple simulation modeling applications: solar activity on the sun's surface, radiation propagation in the region between the sun and the earth, the effects of the solar wind on earth's ionosphere, etc. InterComm couples the applications, which may be written in different languages and run in parallel on diverse operating systems, and enables data to be transferred between them at appropriate times and at appropriate simulation scales.

To support that, InterComm relies on several *system components* including multiple C, C++ and Fortran compilers, parallel data communication libraries, a

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

process management library and a structured data management library. Each component has multiple versions and there are complex dependencies and constraints between the components and their versions.

PETSc PETSc (Portable, Extensible Toolkit for Scientific computation)[2] [12, 43] is a collection of data structures and interfaces used to develop scalable high-end scientific applications. Similar to InterComm, PETSc is designed to work on many Unix-based operating systems and Windows. It provides interfaces and implementations for serial and parallel applications and depends on multiple compilers and parallel data communication libraries to provide the functionality. Also, to enhance the performance of application developed using PETSc, PETSc also relies on third-party numerical libraries such as BLAS [32] and LAPACK [9], and uses Python as the deployment driver.

Although these systems have been used for building many scientific applications and have been improved to correct bugs and also to provide more functionality, there has been no systematic effort to test the compatibility of the systems on diverse field configurations. InterComm has been extensively tested in *only* three different configurations, where each is realized on developers' and users'

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

machines running different operating systems. Likewise, there is no detailed information on configurations on which PETSc is compatible (or incompatible), although PETSc developers have documented brief discussions on building PETSc on multiple configurations with different operating systems.

Sugar interface (OLPC) The One Laptop Per Child (OLPC) project[3] is an international effort to provide educational opportunities for the world's poorest children by empowering each child with a low-cost, connected laptop with free content and software. All software tools running on their laptops are free and there are groups of volunteer developers for creating software that runs on the laptops. In order to develop applications, developers have to use a development environment called *Sugar*. There are dependency chains between components required for building Sugar on an operating system. For example, Sugar depends on *Telepathy*, a framework for real-time conversations, including instant messaging and voice/video calls, and Telepathy in turn depends on *Avahi*, a network service that enables programs to publish and discover services and hosts running on a local network without any specific configuration.

[3] *http://laptop.org*

The Sugar environment is an example system that requires compatibility testing in a non-scientific domain. Currently, to develop applications that use the latest Sugar features, developers have to use an operating system that contains Sugar already in the operating system distribution, or they have to use a limited set of operating systems and versions for building the Sugar from the source code. In other words, developers have to change or upgrade their operating systems. If Sugar developers perform compatibility testing and identify incompatibilities between components required for building the Sugar environment, it would be possible to increase or at least to figure out the range of platforms that developers can use.

1.2 Thesis and Contributions

In this dissertation, I support the following thesis: *Direct-dependency-based configuration sampling techniques can be effectively employed for testing build-compatibility of component-based systems*. To support this thesis, a set of algorithms and tools have been developed and evaluated by performing compatibility testing for two real-world systems. More specifically, I make the following novel contributions not addressed in previous related research:

1. I present the first approach for systematically supporting compatibility test-

ing by examining a small set of configurations sampled from the configuration space of component-based systems. The approach proposed in this dissertation can help developers to rapidly and effectively identify compatibilities between components required for building their systems.

2. I present and evaluate an approach for incremental compatibility testing . When components in a system model evolve constantly over time, that approach can decrease the time required for testing the compatibility of the modified system by a large amount by sampling and testing only configurations that test compatibilities related to modified components, while utilizing previous test results for the sampling and testing process.

3. I present and evaluate a prioritization mechanism that schedules the test order of sampled configurations. The mechanism makes use of developers' preferences over components and their versions for computing priorities of configurations, and can provide developers with compatibility results for highly-preferred configurations early in the test process.

The usefulness of this research is demonstrated by performing experiments and simulations on the InterComm and PETSc. In this dissertation, the application of compatibility testing is restricted to the *build process* (i.e. compilation

and deployment) of a component with other components required for building the component.

1.3 Structure of the Dissertation

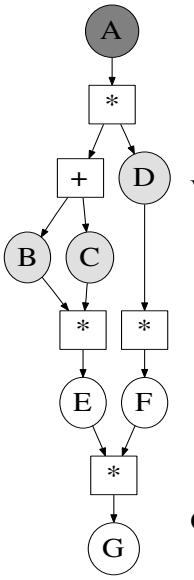
The rest of this dissertation is organized as follows. The next chapter presents studies related to this research. Chapter 2 describes a high-level overview of the steps needed to perform compatibility testing for component-based systems. Chapter 4 formally defines a model for capturing the configuration space of a component-based system and algorithms for sampling and testing configurations. Chapter 5 presents an approach for incremental compatibility testing under the circumstances that components in a system model evolve over time. In Chapter 6, a mechanism for prioritizing the test order of configurations respecting developers' preferences is presented. Experiments and simulation results obtained from empirical studies are presented in each of Chapters 4, 5, and 6. Finally, Chapter 7 concludes this dissertation with a brief discussion and further work.

Chapter 2

The Compatibility Test Process Using Rachet

I describe in this chapter a high-level overview of the steps needed to perform compatibility testing for a given software system under test (SUT) using Rachet.

1. Model system configuration space: To define the configuration space, i.e., the ways in which the SUT may be legitimately configured, developers first need to identify the components required for building the SUT. This information can often be obtained, at least in part, from the component providers. Dependency relationships between components are then encoded as a directed acyclic graph called a *Component Dependency Graph (CDG)*. The example CDG depicted in Figure 2.1 shows dependencies for a SUT component called `A`. As captured in the figure via an `AND` node (represented by `*`), `A` requires component `D` and either one of `B` or `C` (captured via an `XOR` node represented by `+`). Components `B` and



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

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

Figure 2.1: An Example System Model

C require E; D requires F; and E and F require G, the bottom node that does not depend on any other component. *Annotations* include version identifiers for components and constraints between components and/or over full configurations, written in first-order logic. For example, in Figure 2.1, component C has two version identifiers and component C's version C_2 may only be built with E's versions E_3 and higher. Together, the CDG and Annotations form the model called an *Annotated Component Dependency Model (ACDM)*. We formally describe our model in Section 4.1.

2. Determine test coverage criterion: The model encodes the configuration space for the SUT, representing all the different ways in which the SUT may be legally configured in end users' machines. For non-trivial software, this space can be quite large and developers must determine which part of the space will be tested for compatibility testing. For example, they may decide to test configurations exhaustively, which is often infeasible because there are a large number of configurations to be tested and also building components contained in configurations can take a long time. Instead, they may choose more practical criteria that *systematically sample* the space. One such sampling strategy is based on the observation that the ability to successfully build a component c is strongly influenced by the components on which c *directly depends*; the definition and rationale behind this criterion is further described in Section 4.2.

3. Sample configurations and create test plan: Given the model and the coverage criterion, Rachet samples configurations that satisfy the coverage criterion where each configuration describes a set of component versions to build and dependency information used for the build process. Then, a test plan is created from the sampled configurations. The test plan specifies a schedule to test successful builds of the components contained in the configurations.

4. Execute test plan: Finally, Rachet executes the test plan by distributing configurations to multiple machines and collecting results. There can be various ways to schedule the configuration test order. We describe in section 4.6 three scheduling strategies we have developed.

In this dissertation, testing a configuration checks whether component versions contained in the configuration can be built without any error when Rachet builds each component on top of other components in the configuration. Therefore, a sequence of instructions for building component versions is executed during the execution of a test plan.

When a component build fails, Rachet dynamically modifies a test plan so as not to lose test coverage. Such build failure may prevent testing other components in the configuration. In this case, Rachet creates additional configurations that try to build those component versions in alternate ways, as described in Section 4.7.

Chapter 3

Related Work

This chapter introduces efforts by software researchers over the last decade for ensuring compatibilities of a system with various field configurations.

Compatibility Testing on Multiple Configurations

Duarte et al. [33, 34] describe a technique that tests the behavior of a software system on diverse field configurations. They distribute JUnit test suites of a system onto multiple heterogeneous machines accessible in a network of machines. The distributed software is built on the machines and test suites are run for testing the behavior of the software. The configuration provisioning and actual build of the software are handled by a system called SmartFrog [61], which uses a model-based approach for describing configurations. Although their work pursues a similar goal to the approaches presented in this dissertation, they do not

analyze the configuration space of the system for sampling configurations that can effectively identify compatibilities between components, but instead they simply run given test suites only on a limited set of configurations realized on available machines at the time tests are performed.

VMware has developed approaches called *Test lab automation* and *Virtual lab manager* [6, 7, 8] to support compatibility testing on top of various configurations. Although this approach can provide developers with automated support for testing software systems in various configurations realized as virtual machines, developers have to manually customize configurations. Our approach can achieve this without any intervention from developers after they model the configuration space of their software system.

Combinatorial Interaction Testing

The Skoll system [56, 57, 72, 73] is designed to ensure correct build and execution of a software system across a large configuration space, utilizing heterogeneous and distributed resources. Skoll is different from our work in that Rachet addresses a configuration space defined by architectural concerns. Skoll is more focused on the configuration space as defined by traditional compile- and run-time options. Techniques to test highly configurable systems have been extended

to testing of software product-lines. Cohen et al. [26] apply combinatorial interaction testing methods to define test configurations that achieve a desired level of coverage, and identify challenges to scaling such methods to large, complex software product lines. Although not directly related to our idea of sampling configuration spaces via testing DD-instances, they too illustrate how software product line modeling notations can be mapped onto an underlying relational model that captures variability in the feasible product-line instances. They use the relational model as the basis for defining a family of coverage criteria for product-line testing.

Software Regression Testing

There have been studies on software regression testing that select test cases for testing modified systems, since running all test cases can be very expensive and also there are test cases not related to modified parts of the systems [42, 55, 58, 59]. Qu et al. [58] showed that a combinatorial interaction testing technique can be used to select test cases for user configurable systems. Although the basic idea of incremental compatibility testing in Chapter 5 is similar to their work, their approach is applied only to a flat configuration space, which is not for hierarchically arranged systems. Robinson et al. [59] presented an idea for testing user

configurable systems incrementally by identifying configurable elements that are affected by changes on a user configuration and by running test cases called *firewalls* for testing the identified elements. However, they do not proactively test the configuration space and instead postpone testing until a user has changed a deployed system configuration.

Continuous System Integration

As an effort to reduce integration problems during the software development process, there is an industry practice called *continuous integration* [14, 38]. It is an effort to ensure the compatibility of a system through the lifespan of the system by integrating source code changes frequently into the complete software system and by inspecting whether those changes cause problems on top of various machine configurations. As reported in [60], it has been strongly advocated because it can be applied to many software projects with relatively low effort and also because problems originating from the difference between development and field configurations can be detected earlier in the software development process. There are several practical tools supporting continuous integration on top of diverse configurations through a uniform build interface. Such tools include ETICS [54], CruiseControl [2] and Maven [51]. Although these systems perform build tests

for components, their build process is limited to a set of predetermined configurations. Rachet rather produces plausible configurations automatically considering available information on component versions and inter-component dependencies.

Component Installation Management

Our work is broadly related to component installation managers that deal with dependencies between components. Opium [68] and EDOS [50] are two example projects. Opium makes sure a component can be installed on a client machine. The problem of determining whether a component can be installed on a client machine is modeled as a satisfiability problem and is solved using a SAT solver for finding an optimal satisfiable configuration for installing the component. The EDOS project checks for conflicting component requirements at the distribution server. It provides a set of tools for managing the compatibility of components contained in a distribution server. Both projects assume that component dependencies and constraints are correctly specified by component providers and that there is no compatibility problem if the dependencies and constraints are satisfied, Our approach differs in that we validate component compatibilities by testing a set of configurations in which the components may be installed.

Chapter 4

Sampling and Testing Configurations

In this chapter, I describe in detail each step in the process described in Chapter 2. I first define a formal graph-based model that encodes all deployable configurations and a test adequacy criterion for compatibility testing. Then, I present algorithms that generate configurations satisfying the criterion. Then, I describe a method to test sampled configurations efficiently utilizing multiple machines and present experiment and simulation results obtained by applying the Rachet process to two real-world systems [74, 75].

4.1 Configuration Space Model

Components, their versions, inter-component dependencies and constraints define the configuration space of a system under test (SUT) where the SUT is sup-

posed to be deployed successfully. The configuration space of the SUT is captured into a formal model called a *Annotated Component Dependency Model (ACDM)*. An ACDM consists of a *Component Dependency Graph (CDG)* and *Annotations (Ann)*.

A CDG has two types of nodes – component nodes and relation nodes; directed edges represent dependencies between components encoded by the nodes. For example, Figure 2.1 depicts an SUT A that requires components (B and D) or (C and D), each of which depends in turn on other components. As shown in the figure, inter-component dependencies are captured by relation nodes labeled either "$*$" or "+", which are interpreted respectively as applying a logical AND or XOR over the relation node's outgoing edges.

Annotations provide additional information about components in a CDG. The first set of annotations for this example system is an ordered list of version identifiers for each component. Each identifier represents a unique version of the corresponding component. In Figure 2.1, component B has two version identifiers: B_1 and B_2.

Version-specific constraints are common between various components in a model. For example, in Figure 2.1 component C has two versions and depends on component E, which has 3 versions. Suppose that component C's version

$\texttt{C}_2$ may only be compiled using E's versions $\texttt{E}_3$ and higher. This "constraint" is written in first order logic and appears as $(\texttt{ver(C)} == \texttt{C}_2) \rightarrow (\texttt{ver(E)} \geq \texttt{E}_3)$. Global constraints may also be defined over entire configurations. For instance, for our case studies in this dissertation, we require all components depending on a C++ compiler version to use the identical C++ compiler version in any single configuration.

We now formally define the ACDM:

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

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

Furthermore, valid CDGs must obey the following properties: (i) There is a single distinguished component node with no incoming edges, called ***top****. Typically* ***top*** *represents the SUT. (ii) There is a single distinguished component node with no outgoing edges, called* ***bottom****. This component is not dependent on any other component. (The bottom node may represent an operating system, but that is not required.) Dependencies between components may also be encoded using other formalisms such as feature-based [29, 30] or rule-based models [66].*[1] *(iii) All other component nodes,* $v \in \{C/\{$***top,bottom***$\}\}$*, have exactly one incoming edge and one outgoing edge.*

Definition 3 *The annotation set,* Ann *used in an ACDM contains two parts: (i) For each component* $c \in C$*, a set of component properties. One example property is the range of elements (versions) over which* c *may be instantiated, which must be specified for each component. (ii) A set of constraints between components and over configurations. The constraints are specified in a set of expressions that use boolean operators (*$\vee$*,* $\wedge$*,* $\rightarrow$*,* $\neg$*) and relational operators (*$\leq$*,* $\geq$*,* $==$*,* $<$*,* $>$*); component properties are used to evaluate the expressions.*

[1] To capture dependencies, graphical notations similar to the CDG are used for feature-based models, and textual descriptions are used for rule-based models.

4.2 Direct Dependency between Components

Except for the component encoded by the bottom node, each component in a CDG depends on all components on any path from the node encoding the component to the bottom node. However, the correct build of the component depends on a set of components that are directly used by the component.[2] Definition 5 defines a set of components on which a component *directly depends*.

Definition 4 *In A CDG, a component c directly depends on a set of components, DD, such that for every component, $DD_i \in DD$, there exists at least one path in the CDG, not containing any other component node, from the node encoding c to the node encoding DD_i.*

For example, the component B in the example from Figure 2.1 directly depends on the component E, although it also uses functionalities provided by the component G through the component *E*.

From these direct dependencies defined between components, Rachet computes *DD-instances*, which are the concrete realizations of direct dependencies,

[2]Hence, in practice, for building a component, many component build tools such as GNU Autoconf/Automake [31, 69] and Ant [44] check only for the existence of other components on which the component to build directly depends, and check for basic functionalities provided by the components, by generating and testing a simple program during the build process.

specifying component versions. A *DD-instance* is a tuple, (c_v, d), where c_v is a version v of component c, and d is the set of component versions on which c directly depends. For each component in a model, Rachet computes a set of DD-instances. When multiple relation nodes lie on a path between a component and other components on which it directly depends, Rachet applies the following set operations recursively for the relation nodes: Union for XOR nodes and Cartesian product for AND nodes. For example, (A_1,$\{\mathsf{B}_1$,$\mathsf{D}_1\}$) is one of 8 DD-instances for the component A in the example from Figure 2.1.

Since the application context for the compatibility testing in this dissertation is restricted to checking error-free build of components[3], testing a DD-instance (c_v, d) means checking whether c_v can be built without any build error with the component versions contained in d. If c_v can be built without any error, we say that c_v is *build-compatible* with d.

Note that, for testing a DD-instance (i.e., checking the build-compatibility of c_v with d), it is necessary to build in advance the component versions contained in d. Hence, in addition to the DD-instance to be tested, a *configuration* must contain all DD-instances to build component versions contained in the dependency part of

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

any DD-instance in the configuration. In a CDG, a configuration to test a DD-instance of a component can be formally defined as follows:

Definition 5 *In a CDG $G = (N, E)$, a configuration to test a DD-instance of a component represented by a component node $n \in N$, is an annotated subgraph of G, $G' = (N', E')$, such that (1) $n \in N'$, (2) for every component node $n' \in N'$, n' is annotated with a DD-instance for n', (3) for a component node $n' \in N'$ and another component node n'', $n'' \in N'$ if and only if a version of the component represented by n'' is contained in the dependency part of the DD-instance for n', (4) for an AND relation node $n' \in N$, $n' \in N'$ if and only if all nodes connected by n''s outgoing edges are contained in N', (5) for an XOR node $n' \in N$, $n' \in N'$ if and only if at least one node connected by n''s outgoing edge is contained in N', (6) for every edge $e = (v, w) \in E$, $e \in E'$ if and only if $v \in N'$ and $w \in N'$.*

From this definition, for the example system from Figure 2.1, a configuration to test the DD-instance (A_1,$\{\mathsf{B}_1,\mathsf{D}_1\}$) is the subgraph of the CDG that contains component nodes A, B, D, E, F and G, where each component node is annotated with DD-instances: (A_1,$\{\mathsf{B}_1,\mathsf{D}_1\}$), ($\mathsf{B}_1$,$\{\mathsf{E}_3\}$), ($\mathsf{D}_1$,$\{\mathsf{F}_2\}$), ($\mathsf{E}_3$,$\{\mathsf{G}_1\}$), ($\mathsf{F}_2$,$\{\mathsf{G}_1\}$) and ($\mathsf{G}_1$,$\emptyset$), respectively. All relational nodes connecting the component nodes must also be contained in the subgraph.

To test each configuration, component versions encoded by DD-instances for component nodes in a subgraph have to be built, enforcing the dependency to build each component version, starting from the component for the bottom node of the subgraph. Therefore, for the rest of this dissertation, a configuration is considered as an ordered list of DD-instances, where the list starts with a DD-instance of the component with no dependency and other DD-instances are ordered respecting dependencies for building component versions encoded by the DD-instances.

4.3 Configurations with Exhaustive Coverage

The most straightforward way to identify the range of configurations in which the SUT is build-compatible is to build the exhaustive set of possible configurations. To compute an exhaustive configuration set, we start from the *bottom node* of the CDG, and for each node type, do the following:

• *Component node:* compute new configuration set by extending each (partial) configuration in the configuration set of its child node (a relation node) with each DD-instance of the component, only when adding the DD-instance to the configuration does not violate any constraints. For each DD-instance (c_v, d) of the component, we first identify configurations in the configuration set of the child node

where each configuration has all DD-instances for building component versions contained in d. Then, extend each configuration with the DD-instance (c_v, d), if adding the DD-instance to the configuration does not violate any constraints. For the *bottom node*, simply return the set of configurations where each contains a DD-instance of the component represented by the bottom node, as the component has no dependency.

• *AND node:* compute the Cartesian product of configurations taken from each configuration set computed for the child nodes of the AND node, then merge the configurations in each combination in the product by creating a new configuration that contains all DD-instances from the configurations. We enforce two rules implicitly in merging configurations. First, only one version for each component is allowed in a combined configuration. That is, we do not allow combining configurations when each contains a DD-instance for the same component, but with a different version. Second, we require that a combined configuration contains a single DD-instance for each component version.

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

Even for the simple CDG in Figure 2.1, the number of configurations in the exhaustive set for building component A is 60. Since a CDG for a real applica-

tion can be very complex and contain many components, as shown in Figure 4.5, the number of configurations for exhaustive coverage may be large. Considering the potentially long time required for building each complete configuration, for many CDGs it may be infeasible to test all possible configurations. This means that we need a method that samples configurations intelligently so that the results from testing the sampled configurations are sufficient for determining whether a configuration realized in a user's machine can be used for building a system. Testing randomly sampled configurations may not provide complete information for making the decision.

4.4 Configurations with DD-Coverage

The default sampling strategy of Rachet is called *DD-coverage* and is based on testing the DD-instances for components in a model. The motivation for this strategy is that the correct build of a component mostly depends on a set of components on which the component directly depends.

Once *DD-instances* for all components have been computed, Rachet computes a set of configurations in which each DD-instance appears at least once. This is achieved by applying the algorithm **DD-Coverage** shown in Figure 4.1 to all

Algorithm DD-Coverage ($CDG\ G$**)**

```
1:  ConfigSet ← ∅ // configuration set
2:  for each component node n ∈ G in topological order from the top node do
3:     Let c the component represented by the node n
4:     Let DD_c the set of DD-instances for the component c
5:     for each uncovered ddi = (c_v, d) ∈ DD_c do
6:        C ← BuildCFG({ddi}, d)
7:        if C ≠ ∅ then
8:           ConfigSet ← ConfigSet ∪ {C};
9:           set ddi covered;
10:       end if
11:    end for
12: end for
13: return ConfigSet
```

Figure 4.1: DD-Coverage apply BuildCFG for generating each DD-instance for components in a CDG

DD-instances for components in a CDG. The algorithm takes each component in a CDG in a topological order, starting from the top node and then attempts to generate a configuration that tests each DD-instance for the component by running the algorithm **BuildCFG** shown in Figure 4.2, and we say that the DD-instance is *covered* by the generated configuration.

The algorithm **BuildCFG** takes two parameters: (1) a list of DD-instances already selected for the configuration under construction, and (2) a set of component versions whose DD-instances must still be added to the current configuration. To generate a configuration for 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. The algorithm then selects a DD-instance for some component version in the second parameter (line 4). 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 (line 5). Then, the algorithm checks whether the extended configuration violates any constraints. If the configuration does not violate constraints, **BuildCFG** is called recursively with the updated parameters (line 11). If there has been a constraint violation, the algorithm backtracks to the state before the DD-instance was selected and tries another DD-instance, if one exists. The algorithm 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 necessary for a configuration that covers ddi_1 (and all other DD-instances selected for making the configuration complete).

The algorithms for generating configurations work greedily. Each configuration it generates covers as many previously uncovered DD-instances as possible, with the goal of minimizing the total number of configurations necessary to

Algorithm BuildCFG(Cfg, U)

1: // Cfg: configuration under construction
2: // U: component versions whose DD-instances need to be picked
3: $c'_v \leftarrow$ a comp. version from U
4: **for** $ddi' = (c'_v, d') \in DD_{c'}$ **do**
5: $(Cfg', U') \leftarrow$ (append(Cfg, ddi'), $d' \cup U - \{c'_v\}$)
6: // $DD_{c'}$ is the DD-instance set of c'_v
7: **if** Cfg' does not violate any constraints **then**
8: **if** $U' == \emptyset$ **then**
9: set ddi' covered; **return** Cfg'
10: **else**
11: $Cfg'' \leftarrow$ **BuildCFG**(Cfg', U')
12: **if** $Cfg'' \neq \emptyset$ **then**
13: set ddi' covered; **return** Cfg''
14: **end if**
15: **end if**
16: **end if**
17: **end for**
18: **return** $\emptyset$ // there is no legal way to build c'_v with the DD-instances in Cfg

Figure 4.2: Algorithm to generate a configuration to cover a DD-instance

cover all DD-instances for components. This is achieved in the algorithm, by (1) selecting an *uncovered DD-instance first* in the selection process (line 4 in the **BuildCFG** algorithm), and by (2) applying the algorithm for the DD-instances of components in topological order, starting from the *top* component in a CDG (line 2 in the **DD-Coverage** algorithm), since more DD-instances may be covered from multiple DD-instance sets when applied to a DD-instance of components close to the *top* component in the CDG, compared to those farther from *top*. For the example system, **DD-Coverage** produced 11 configurations that cover all 31 DD-instances for the components involved in the model.

4.5 Test Plan Synthesis

In order to test generated configurations, we take each of the configurations and topologically sort the DD-instances contained in the configuration to produce an ordered sequence of components. That is, the i^{th} component in this sequence does not depend on any component with an index greater than i. Therefore it is legal to build the configuration by first building the 1^{st} component in the sequence, then building the 2^{nd} component, etc.

The configurations may be tested one at a time by building each component

in each configuration on a machine according to the sequence order. However, the total number of component versions that must be built can be reduced by utilizing the fact that multiple configurations may contain identical sequences of DD-instances. For example, all configurations contain the same first DD-instance, that builds an operating system, if only one version of the operating system is used in the model.

To reduce the number of components to build, Rachet then combines the build sequences for all configurations 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 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 partial configurations across multiple configurations. The prefix tree then acts as a *test plan*, showing all opportunities to share common build effort. Figure 4.3 depicts two test plans, one from the 60 configurations produced exhaustively (EX-plan) and the other from the 11 configurations with DD-coverage (DD-plan) for the example system. An example configuration contained in both plans is shaded in the figure. The DD-plan for the example system contains 11 configurations, with 37 nodes (component ver-

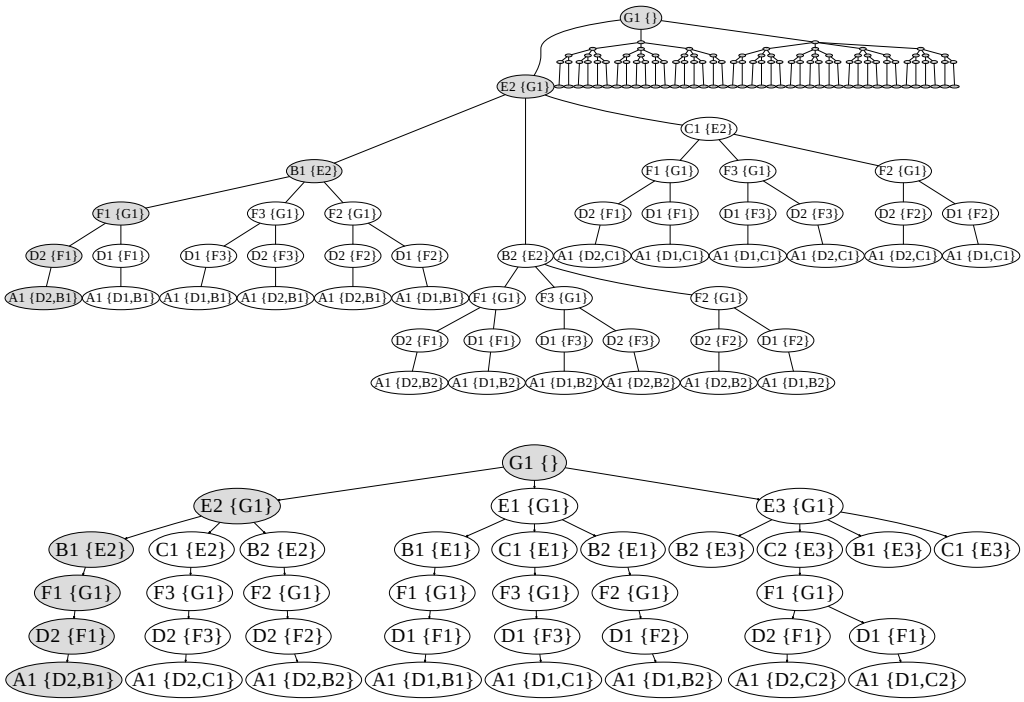


Figure 4.3: EX-plan (top) and DD-plan (bottom) for example model

sions to be built), reduced from 56, the number of DD-instances contained in the configurations generated by applying the **BuildCFG** algorithm.

4.6 Test Plan Execution Strategies

A test plan created by the process described in the previous section can be executed in several ways. Test plan execution visits all nodes in a plan, and when a plan node is visited, Rachet tests the *build-compatibility* of the DD-instance (c_v, d) represented by the node. That is, Rachet builds c_v with the component versions

in d and records the result. However, to do that, Rachet first needs to build all component versions represented by the nodes in the path from the root node to the parent of the plan node, and it can be time consuming if Rachet builds all the component versions from scratch on an empty configuration, whenever Rachet visits a plan node.

Therefore, Rachet uses a *virtual machine* (VM) for a *partial configuration* (i.e., a prefix of the test plan under execution). A benefit from building components inside a VM is that we can avoid contaminating the persistent state of a physical test resource (machine). In addition, if the components in a prefix are built successfully without any error, the modified machine state has the correct state for the prefix and the VM may be *reused* to test DD-instances represented by descendant nodes in a subtree rooted at the last node of the prefix, since a VM can be represented as a file and can be cloned by copying the VM. When we test the DD-instances, we need only to build additional components by reusing the VM state.

In this section, we describe three plan execution strategies and also describe the mechanism for dynamically handling component build failures. Although not required, for executing a test plan, we assume that a single server controls the plan execution and dispatches prefixes to multiple clients. We also assume that each

client has disk space available to cache VMs (completed prefixes) for reuse.

Rachet's final output after the execution of a test plan are test results indicating whether each DD-instance was (1) *tested and built successfully*, (2) *tested and failed to build*, or was (3) *untestable*, meaning that there was no way to produce a configuration to test that DD-instance. For example, suppose that in testing the system in Figure 2.1 all attempts to build B_2 with E_1 through E_3 fail. Then all DD-instances that require a version of A to be built over B_2 are untestable.

4.6.1 Parallel Depth-First Strategy

The parallel depth-first strategy is designed to maximize the reuse of locally cached prefixes at each client during the plan execution. When a test client completes testing a prefix from the plan root to a node n and subsequently requests a new prefix, the test server assigns a prefix according to the following rules, attempting to maximize the reuse of cached prefixes.

First, if the node n is a non-leaf node in the plan, the prefix for one of n's unassigned child nodes is chosen as the next prefix for the client. The client will then reuse the VM state from its previously tested prefix, so only has to build one additional component (the one represented by the last node of the new prefix). This is typically the least expensive way to test a new prefix, because the cost to

test the prefix is only the time to boot up the VM and to build one component on top of the VM.

Second, if the node n is a leaf node, prefixes already stored in the cache space of the client are utilized to assign a new prefix. Starting from the node for the most recently cached prefix, the algorithm searches for an unassigned descendant node in the plan in depth-first order. Nodes whose DD-instance is currently under test by other clients, and their subtrees in the plan, are not considered by the search. In this case, the test client must build the *difference* between the assigned prefix and the reused prefix.

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

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

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

4.6.2 Parallel Breadth-First Strategy

The parallel breadth-first strategy focuses on increasing the number of prefixes being tested simultaneously, and secondarily tries to maximize the reuse of locally cached prefixes. To dispatch prefixes in breadth-first order, the server maintains a

priority queue of plan nodes ordered according to their depth in the plan.

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

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

Unlike the depth-first strategy, for the breadth-first strategy, a completed prefix is always cached if its corresponding node in the plan is a non-leaf node. The rationale behind this choice is that in many cases the prefixes for the child nodes will not be assigned to the same client. This strategy will keep all clients busy as long as there are unassigned nodes in the queue throughout the plan execution.

Therefore, we expect a high level of parallelism. However, we also expect less local cache reuse and increased network cost compared to the depth-first strategy, because of transferring many cached prefixes across the network.

4.6.3 Hybrid Strategy

We have described costs and benefits of the depth-first and breadth-first strategy. Although the depth-first strategy tries to maximize the locality of reused prefixes, during the early stage of plan execution it may not fully maximize the parallelism that could be obtained by testing prefixes on all available clients. On the other hand, the breadth-first strategy may achieve a high level of parallelism, but may also increase the network cost to test prefixes.

The hybrid strategy is designed to balance both the locality of reused prefixes and the parallelism throughout plan execution, by combining the features of both strategies. As in the breadth-first strategy, a priority queue of plan nodes is created by traversing the test plan, and is used to increase the parallelism during the early execution stages of the plan. That is, for the initial requests from test clients, prefixes for nodes in the queue are assigned to *all* available clients immediately at the beginning of plan execution.

To maximize locality for reused prefixes, the first and second rules for the

depth-first strategy are subsequently applied to assign prefixes to requesting clients. If both rules fail to find an unassigned node, the test plan is traversed in breadth-first order from the root node to find an unassigned node. This is based on the heuristic that a node closer to the root node will likely have a larger subtree beneath it than nodes deeper in the tree, which will lead to more work being made available for a client reusing locally cached prefixes.

4.7 Dynamic Failure Handling

For a DD-instance (c_v, d), if test plan execution failed to build c_v on top of the component versions in d, c_v is *build-incompatible* with the component versions contained in d, and Rachet uses this failure information to guide further plan execution. Since the DD-instance is encoded by a node of a prefix in the plan, the build failure prevents testing of all DD-instances represented by the descendant nodes of the node in a test plan. This is because we need a VM on which all DD-instances of the ancestor nodes have been built to test the DD-instances. However, the failure does not imply failure for *all* the DD-instances affected by the failure. Instead of simply regarding the DD-instances as *untestable*, we dynamically adjust the test plan under execution for testing the DD-instances in alternate ways,

if possible. That is achieved by producing additional configurations that cover the DD-instances, and merging the new configurations into the test plan. This strategy attempts to maximize the number of DD-instances tested during the plan execution.

Since one DD-instance in the test plan can participate in multiple configurations, it can appear as multiple nodes in different branches of a test plan. If one of the nodes encoding an identical DD-instance fails, we expect the others to also fail from Definition 5. Thus, the test plan nodes affected by a build failure are not confined to the descendant nodes in the subtree of the failed node, but also include all descendant nodes in the subtrees of the nodes encoding the same DD-instance. Therefore we must produce new configurations to cover all the affected DD-instances.

To reduce the number of newly produced configurations we apply the **Build-CFG** algorithm in Section 4.4 for the affected DD-instances represented by the nodes under the subtrees in a post-order tree traversal. As we generate configurations for covering DD-instances of components in the CDG in topological order, we expect that a DD-instance represented by a node will be covered while generating configurations for the DD-instances represented by its descendant nodes.

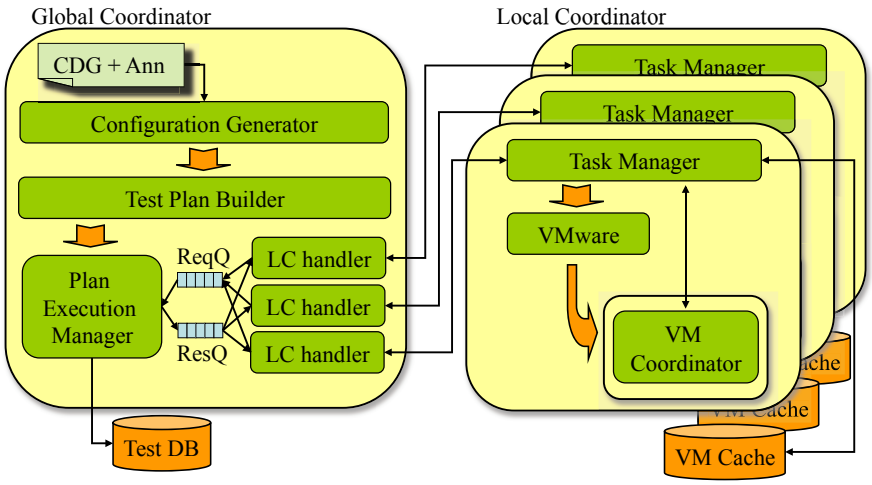


Figure 4.4: Rachet Software Architecture

4.8 Rachet Architecture

We have developed an automated test infrastructure that supports the Rachet process. The Rachet infrastructure is designed in a client/server architecture, as illustrated in Figure 4.4, utilizing hardware virtualization technology. We call the Rachet server the *Global Coordinator (GC)* and the client the *Local Coordinator (LC)*.

Global Coordinator (GC)

The GC is the centralized test manager that directs overall test progress, interacting with multiple clients. It first generates configurations that satisfy the desired coverage criteria (e.g., DD-coverage) and also produces a test plan from the

configurations, using the algorithms described in Section 4.5. Then, the GC dynamically controls test plan execution by dispatching prefixes and the ancillary information necessary to test the prefix to multiple clients, according to one of the test plan execution strategies described in Section 4.6.

Specifically, the GC has a *testmanager* and a set of *lchandlers*, where each *lchandler* is dedicated to a client machine. The *testmanager* is responsible for creating configurations and a test plan. During the test execution, the *testmanager* satisfies requests from clients and updates test results in a database. When a client first requests a prefix to test, the GC creates an *lchandler* for the client and that *lchandler* is responsible for all further communication with the client. Each *lchandler* enqueues requests from the LC into a *shared request queue*. Then, the *testmanager* handles the requests in first-in-first-out (FIFO) order and enqueues matched responses into a *shared response queue*, which enables each *lchandler* to send responses to the LC with which it communicates.

Local Coordinator (LC)

The LC controls testing prefixes in a client. One LC runs on each test machine and interacts with the GC to receive information on prefixes to test and also to report execution results.

As described previously, testing a prefix in the current Rachet design and implementation means *building* the component versions represented by the DD-instances contained in the prefix, taking into account the dependency information used for building the component versions. To do the builds, the LC employs hardware virtualization technology. The components are built within a *virtual machine (VM)*, which provides a virtualized hardware layer. This design is advantageous since the persistent state of the test machine is never changed, so a large number of prefixes can be executed on a limited number of physical test machines. The Rachet implementation currently uses *VMware Server* as its virtualization technology, since it handles virtual machines reliably and also provides a set of well-defined APIs to control the VM. A key feature of VMware Server is that the complete state of a VM is stored as files on the disk of the test machine, so can be saved and later reused (i.e. the VM can be stopped and copied, and the original VM and the copy can be restarted independently). We assume that each client has disk space for storing (*caching*) VMs.

Virtual Machine Coordinator (VMC)

The VMC is responsible for the actual component build process in a VM. When a VM is started by the LC, the VMC is automatically deployed in the VM and started by the LC. The VMC then interacts with the LC to receive commands for building component versions contained in the prefix assigned to the LC and also to send command execution results back to the LC. The instructions to build each component are translated into appropriate *system* commands by the VMC and executed in the VM for actual component builds.

Interactions among GC, LC and VMC

Three coordinators in the Rachet system interact with each other to execute a test plan as follows:

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

2. *Assign a prefix*: When a LC requests a new prefix, the GC looks for a prefix from the plan based on a desired plan execution strategy, and dispatches the prefix with ancillary information. For example, the LC needs to know

which VM should be reused for testing the prefix.

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

4. *Deploy and Launch the VMC*: The LC starts the provisioned VM, and deploys and launches the VMC in the VM. The VMC automatically connects to the LC and establishes a communication channel.

5. *Build components*: The LC sends instructions to the VMC to build the components contained in the prefix, and the VMC translates the instructions into a series of commands (e.g., for Unix, commands for executing programs from a *shell*) and then executes them on the VM.

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

4.9 Evaluation

In this section, we present experimental and simulation results obtained by performing empirical studies with the two scientific middleware systems described in Chapter 1. In particular, we focus on examining the cost and benefit of testing only configurations with DD-coverage compared to testing all feasible configurations, and on investigating the performance behavior of plan execution strategies.

4.9.1 Modeling the Subject Systems

To perform compatibility testing for the subject systems, we first modeled component dependencies, working directly with the InterComm developers and carefully inspecting documentation provided by the PETSc developers. In Figure 4.5, we show the component dependencies captured for InterComm and PETSc in a single CDG. The nodes specific to PETSc are shaded in the figure. Version annotations and brief description on the components used in the CDG are depicted in Table 4.1.

In addition to component versions, the following constraints are specified as part of the annotations and must be satisfied by each configuration. First, if the same vendor compilers (i.e., *gcr, gxx, gf, gf77* or *pc, pxx, pf*) are used in a con-

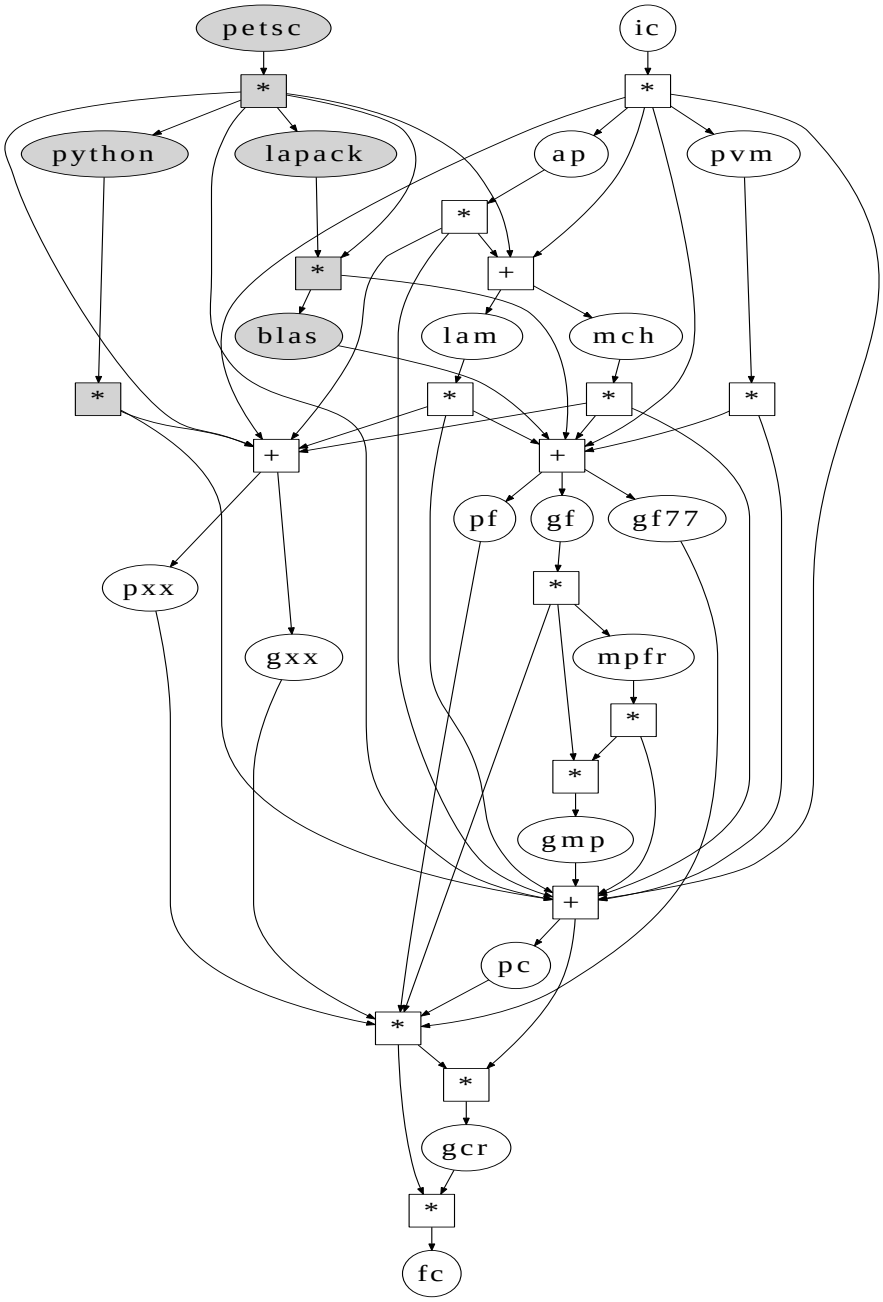


Figure 4.5: A Combined CDG for InterComm and PETSc. (Shaded nodes are specific for PETSc.)

Component	Version Identifiers	Description
petsc	2.2.0	PETSc, the SUT
ic	1.5	InterComm, the SUT
python	2.3.6, 2.5.1	Dynamic OOP language
blas	1.0	Basic linear algebra subprograms
lapack	2.0, 3.1.1	A library for linear algebra operations
ap	0.7.9	High-level array management library
pvm	3.2.6, 3.3.11, 3.4.5	Parallel data communication component
lam	6.5.9, 7.0.6, 7.1.3	A library for MPI (Message Passing Interface) standard
mch	1.2.7	A library for MPI (Message Passing Interface) standard
gf	4.0.3, 4.1.1	GNU Fortran 95 compiler
gf77	3.3.6, 3.4.6	GNU Fortran 77 compiler
pf	6.2	PGI Fortran compiler
gxx	3.3.6, 3.4.6, 4.0.3, 4.1.1	GNU C++ compiler
pxx	6.2	PGI C++ compiler
mpfr	2.2.0	A C library for multiple-precision floating-point number computations
gmp	4.2.1	A library for arbitrary precision arithmetic computation
pc	6.2	PGI C compiler
gcr	3.3.6, 3.4.6, 4.0.3, 4.1.1	GNU C compiler
fc	4.0	Fedora Core Linux operating system

Table 4.1: Component Version Annotations for InterComm and PETSc

figuration, they must have the same version identifier. Second, only a single MPI component (i.e., *lam* or *mch*) can be used in a configuration. Third, only one C++ compiler, and only one of its versions (gxx version X or pxx version Y) can be used in a configuration. Fourth, if both a C and a C++ compiler are used in a configuration, they must be developed by the same vendor (e.g., GNU Project or PGI). For PETSc, we applied one additional constraint: compilers from the *same* vendor must be used to build the PETSc or MPI component. With these constraints, there are 302 and 160 DD-instances for the components contained in the InterComm and PETSc models, respectively.

System	Coverage	# of cfgs	# of Comp$_{cfgs}$	# of Comp$_{plan}$
InterComm	EX-Coverage	3552	39840	9919
InterComm	DD-Coverage	158	1642	677
PETSc	EX-Coverage	1184	14336	3493
PETSc	DD-Coverage	90	913	309

Table 4.2: Test Plan Statistics for InterComm and PETSc

4.9.2 Experiment Setup

For each subject system, we generated two test plans. Table 4.2 summarizes the number of produced configurations, and the number of components contained in those configurations and in the test plan. The first test plan, called EX-plan, was generated using the exhaustive coverage criteria, and the other test plan, called DD-plan, only covers all DD-instances identified for the components in a model. For example, the PETSc EX-plan has 1184 configurations, containing a total of 14336 components to be built. However, the number of components in the final test plan is only 3493, since configurations are merged to produce the test plan.

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

the DD-plans for both subject systems using the parallel breadth-first and hybrid strategies with the same numbers of client machines.

For actual experiments, we ran the GC on a machine with a Pentium 4 2.4GHz CPU and 512MB memory, running Red Hat Linux 2.4.21-53.EL, and we ran LCs on up to 32 machines, all with Pentium 4 2.8GHz Dual-CPU and 1GB memory, running Red Hat Enterprise Linux version 2.6.9-11. All machines were connected via Fast Ethernet. One LC runs on each machine, and each LC runs one VM at a time for testing a prefix. The number of entries in the VM cache for each LC is set to 8, because we observed little benefit from more cache entries for the InterComm example in another experiment, and also because test plans for PETSc are smaller than test plans for InterComm in this scenario.

In addition to these experiments on the real system, we ran simulations using our event-based simulator that mimics the behavior of the key Rachet components, described in Section 4.8, to better understand the performance characteristics of the Rachet on larger sets of resources than we were able to use for the real experiment (both because of limited resource availability and the time required for performing experiments). For the simulations, we used the information obtained from running actual experiments. Such information includes the test results for DD-instances and average times required for building component versions.

4.9.3 Cost/Benefit Assessment of DD-Coverage

As shown in Table 4.2, the EX-plans for both systems have a large number of configurations compared to the DD-plans. Since it takes up to 3 hours to build a configuration for either InterComm or PETSc, it requires about 10600 and 470 CPU hours to execute the InterComm EX-plan or DD-plan, respectively, and 3500 or 270 CPU hours for the corresponding PETSc plans.

With a naive plan execution strategy where each configuration is always built from scratch, with 8 machines it would still take 1325 or 438 hours, respectively for the EX-plans with perfect speedup, and 59 or 34 hours for the DD-plans. However, since our plan execution strategies reuse build effort across configurations, the plan execution times for both plans are expected to be much smaller than times with the naive execution strategy. In our experiments, execution times were further shortened due to many build failures during the plan execution.

The cost savings obtained by executing the DD-plans are shown in Figure 4.6. With 8 machines, the InterComm EX-plan took about 29 hours with the parallel depth-first strategy, during which 461 component builds were successful and 687 failed. All other builds could not be tested due to build failures of other components required for the builds. For the PETSc EX-plan, about 29 hours were needed, during which we observed 724 build successes and 407 build failures,

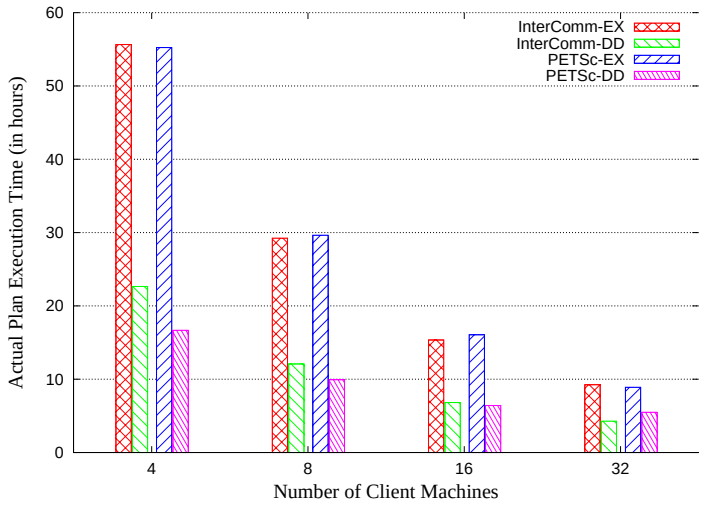


Figure 4.6: Actual turnaround time for executing InterComm and PETSc EX-plans and DD-plans using depth-first strategy.

with the rest not able to be tested. Compared to the EX-plans, the InterComm DD-plan took 12 hours with 275 successful component builds, and the PETSc DD-plan took 10 hours with 216 successful builds.

In our experiments, the execution times for the EX-plans took only 2.5 – 3 times more than those for the DD-plans, because many build failures occurred during plan execution, especially for the components close to the bottom node in the CDG. Note that the difference in execution times between the EX-plans and DD-plans decreases as more clients are used, since the Rachet system always tries to best utilize the machines for plan execution and therefore a larger plan can benefit more when many clients are available.

The results show that Rachet was able to achieve large performance benefits by testing only configurations covering DD-instances, and also was able to execute the test plans efficiently using the depth-first execution strategy. However, we also need to know the potential loss of test effectiveness from using the DD-plan, which only samples a subset of the configurations that are tested by the EX-plan. To do that, we identified the successes and failures for all DD-instances of components in the InterComm and PETSc model by executing EX-plans, respectively. Then, we checked whether building a component version encoded by each DD-instance succeeds or fails in the DD-plan.

We found that each failed DD-instance from the InterComm EX-plan execution exactly maps to a corresponding DD-instance failure in the DD-Plan. However, for the PETSc EX-plan, we observed 8 DD-instances where the PETSc component build success or failure depended on the compilers used for building components on which the PETSc component depends (e.g., when Rachet tries to build a version of the PETSc component with the GNU compilers on a VM, the MPICH component might have been previously built on the machine with the GNU compilers *or* with the PGI compilers.) Unfortunately, all those instances were reported as successful builds during the DD-plan execution. We observed that this happened because there were missing constraints in the model. For these

instances, the missing constraint was that compilers from the same vendor must be used to build all the components on which the PETSc component directly depends. PETSc developers might have simply assumed this constraint. However, users do not always have complete information on the compilers used to build those components on their system, especially if the system is managed by a separate system administrator. Another observation for the PETSc component is that it was never able to be built successfully using the LAM/MPI component. It seems that some undocumented method is required to build PETSc using that MPI implementation.

For InterComm, due to many build failures of the components in the model (mainly, because of not being able to build older versions of the PVM component), we were therefore only able to test build compatibility for 7 DD-instances out of the 156 total DD-instances for the InterComm component. However, they were not the ones on which InterComm had been tested previously. The results show that InterComm can be successfully built with the combinations of PGI C/C++ compiler version 6.2, all versions of the GNU Fortran77 or GNU Fortran90 compilers specified in the model, and MPICH version 1.2.7. This is a larger set of components than what the InterComm developers had previously tested, as doc-

umented on the InterComm distribution web page[4]. The DD-instance with GNU C/C++ compiler version 3.3.6 and with the PGI Fortran compiler version 6.2 failed to build. The failure occurred because the InterComm *configure* process reported a problem in linking to Fortran libraries from C code. This result is interesting since the InterComm developers were able to build successfully with GNU C/C++ version 3.2.3 and PGI Fortran version 6.0. InterComm developers investigated the reason for the failure, and it turned out that the failure was due to a missing environment variable required for the *configure* step. This was not documented in the InterComm manual, and InterComm developers updated the manual accordingly to notify users that the environment variable (`LDFLAGS`) must be set correctly if a PGI Fortran compiler is used for the InterComm build.

4.9.4 Comparing Plan Execution Strategies

As seen in Figure 4.6, Rachet scales very well as the number of machines used for running Rachet clients increases from 4 to 32. When we double the number of available machines, the execution time decreases by almost half, up to 16 machines. This means that Rachet can fully utilize additional resources to maximize the number of prefixes tested in parallel. However, Figure 4.6 only shows results

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

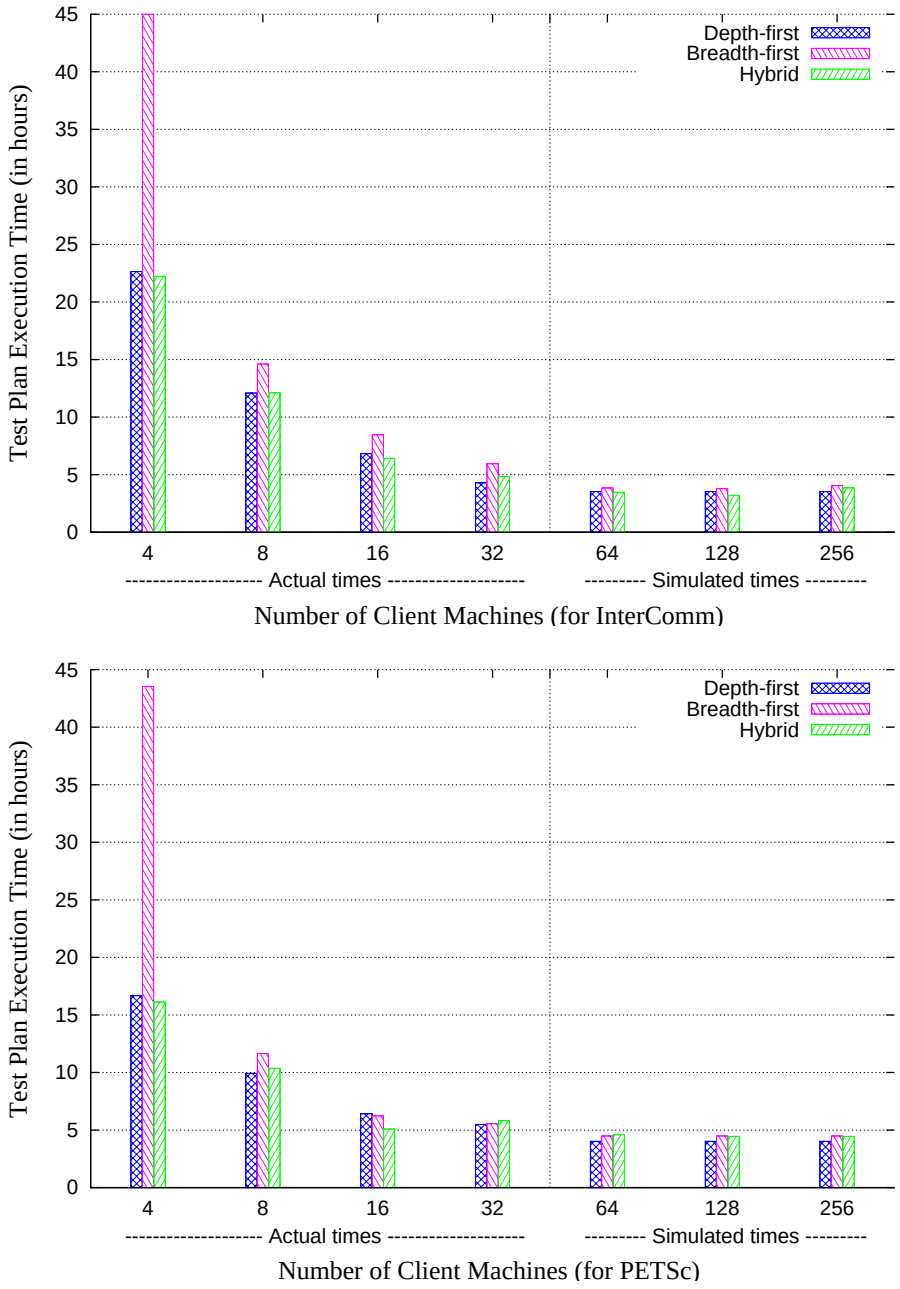


Figure 4.7: Turnaround times for executing the InterComm and PETSc DD-plan with different plan execution strategies.

obtained by executing the DD and EX plans for the subject systems using the parallel depth-first strategy. To analyze the performance behavior of the different plan execution strategies, we also executed the DD-plans for InterComm and PETSc using the other strategies.

Figure 4.7 shows the combined results from both actual and simulated test plan executions with different strategies. For both systems, we ran actual experiments with 4, 8, 16 and 32 clients. For larger numbers of clients, we ran simulations to compute expected plan execution times. The data used for the simulations, including the component build successes/failures, the times for managing VMs (e.g., VM start-up time) and the times for building components, were all obtained from real experiments. The simulated execution times were, on average, about 18% less than the real execution times for up to 32 clients.

We found that the breadth-first strategy performed worst for most runs. As described before, with the breadth-first strategy, Rachet tries to utilize as many machines as possible throughout the plan execution, and always reuses the best cached VM for testing each prefix. However, the time to transfer the VMs across the network was a performance bottleneck, even though the clients were connected via Fast Ethernet. Breadth-first performed especially poorly with 4 machines compared to the other strategies, because in many instances the cached

prefix that requires the minimum additional component builds for testing a newly dispatched prefix had already been replaced in the VM cache before it was needed, and as a result all components contained in the prefix had to be rebuilt.[5] For our experiments and simulations, we used a Least-Recently-Used (LRU) cache replacement policy to manage the VM cache on each machine. We also experimented with a First-In-First-Out (FIFO) cache replacement policy, but did not see a significant performance difference compared to LRU.

Many build failures that occurred during the plan executions are responsible for the similar performance between the hybrid and depth-first strategy in Figure 4.7. With a small number of clients, the depth-first strategy can maximize the number of prefixes that are tested in parallel *shortly* after starting the plan execution. However, with many clients, build failures negate the benefits of the hybrid strategy that are achieved by maximizing the number of prefixes dispatched early in the plan execution.

We also observe that little benefit is achieved with more than 32 machines for all strategies, because many machines remained idle waiting for prefixes to be dispatched, after all available prefixes are dispatched to other machines. More-

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

over, the execution times may even increase slightly with a large number of machines, because the local cache hit rate drops when prefixes are spread across the machines, and also because additional time is needed to transfer cached prefixes across the network, negating the benefit of greater parallel component builds. Despite these overheads, we expect that the hybrid strategy will achieve the best performance as we increase the number of machines, if a test plan has fewer failures.

Therefore, for our final experiment, we evaluated how Rachet behaves as the number of successful component builds grows. As previously noted, many DD-instances were classified as untestable in the actual experiments, because at least one component in the dependency part of the DD-instances could not be built successfully in all possible ways. If developers were to fix some of these problems, many more DD-instances would be testable, greatly increasing the effective size of the test plan.

We ran simulations for this scenario and measured the benefit of testing only DD-instances, under the assumption that no build failures occurred during plan execution. Figure 4.8 shows expected plan execution times for the subject systems. Both the EX-plan and the DD-plan are executed with the hybrid strategy, and we also applied the other strategies for the DD-plan. We observe that the hy-

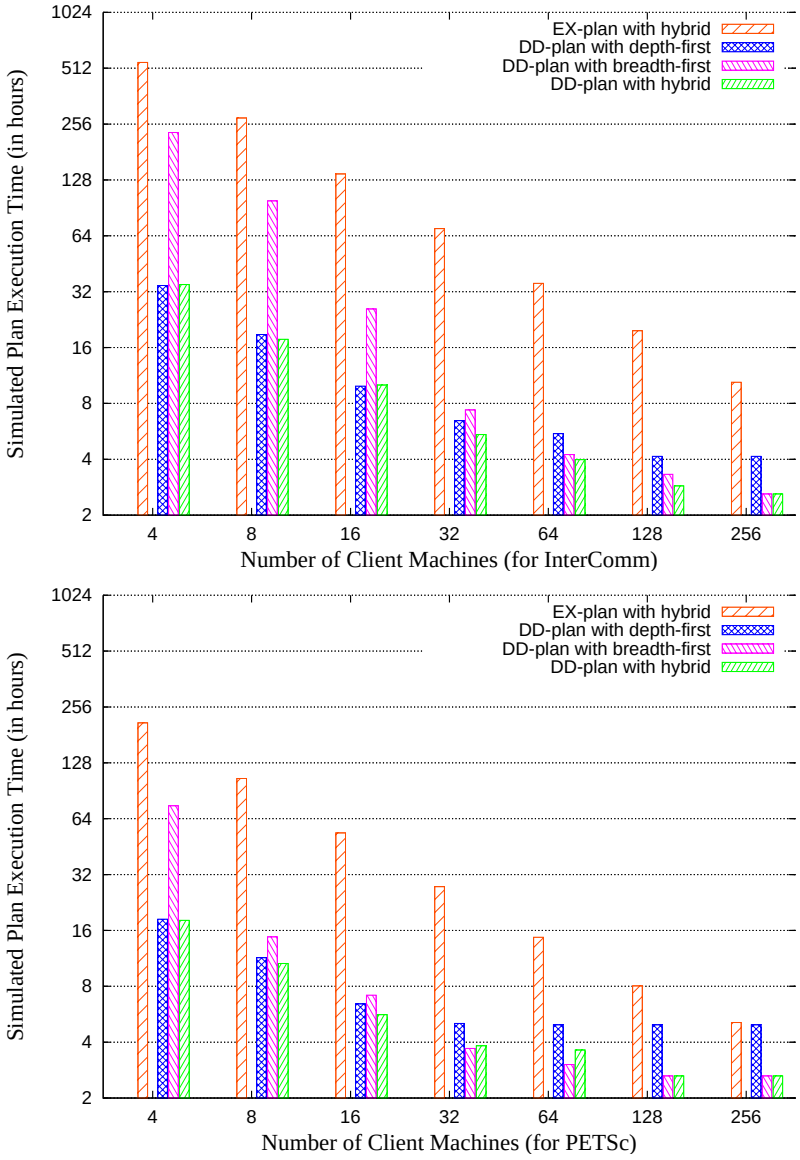


Figure 4.8: Simulated time to execute InterComm and PETSc EX-plan with hybrid execution strategy, and DD-plan with all strategies, assuming no build failure.

brid strategy balances well both the prefix reuse locality and parallel component builds across all numbers of clients. The hybrid strategy is competitive with the depth-first strategy for small numbers of clients, because it tries to maximize the reuse of locally cached prefixes. And the hybrid strategy also achieves good performance for a large number of clients, since the extra costs for caching prefixes and reusing VMs during the early stage of test plan execution are avoided, compared to the depth-first strategy. Although the breadth-first strategy shows good performance with 32 or more machines, such performance relies on the availability of a fast network connecting *all* client machines and small plan sizes. In other simulations that execute a larger test plan with 52618 plan nodes, by employing 64 clients, each with 8 cache entries, we observed that the breadth-first strategy took much longer than the hybrid strategy and also involved many more cache replacements.[6] The simulation results show that the breadth-first strategy took much longer than the hybrid strategy and also involved a large number of cache replacements.

With 256 machines, the time required to execute the InterComm and PETSc DD-plans must be close to the optimal execution time, since all configurations involved in each plan are dispatched to available machines at once and tested in

[6]The test plan was for the monolithic InterComm model in Chapter 5.

parallel. For this case, the overall plan execution time is the time required for building the configuration that takes the longest.

4.10 Summary

In this chapter, I have presented an effective and scalable method for performing compatibility testing of component-based systems.

First of all, to encode the entire set of configurations where a component-based system can be deployed, I developed a formal graph-based representation annotated with component versions and constraints between components and/or over configurations. Because there are large number of configurations for a system and also because available resources are limited, it is infeasible in many cases to test all feasible configurations.

To address this problem I focused on the observation that the successful build of a component mostly depends on other components that are directly used by the component to build. Based on the observation, I defined a test adequacy criterion called *DD-coverage*, which tests all version relationships (*DD-instances*) between a component and other components on which the component *directly depends*, and I also developed an algorithm for generating a set of configurations that satisfy

the coverage. The results from empirical studies on two large software systems demonstrated that compatibilities between components can be identified rapidly and effectively by testing configurations with DD-coverage.

Since there are many configurations that share common prefixes, I developed a method to further reduce efforts required for testing configurations. I first combined all configurations into a single prefix tree called a *test plan* and reused common efforts required for testing prefixes shared across configurations. This is accomplished by utilizing hardware virtualization technology. I used virtual machines (VMs) for building components and saving partial configurations into files. This approach was advantageous because I could avoid contaminating states of test resources and also because VM states could be stored into files on test resources for further reuse and could be transferred between test resources.

To execute a test plan efficiently utilizing multiple test resources, I developed three plan execution strategies (*parallel depth-first, parallel breadth-first and hybrid*) where each strategy is designed for increasing the reuse of partial configurations locally stored on each test machine and/or for decreasing the number of idle machines throughout the plan execution. For all strategies, I employed a contingency planning mechanism for improving the test coverage compared to static approaches when an attempt to build a component fails. I analyzed the tradeoffs

between plan execution strategies when different numbers of machines are employed, by running both actual experiments and simulations. The results suggest that the hybrid strategy can achieve the best performance by attaining both high locality to optimize prefix reuse and high parallelism, for both small and large numbers of client machines.

Chapter 5

Embracing Component Evolution

The previous chapter describes an approach for testing compatibilities between components for a system that involves a *fixed* set of components and their versions. In the approach, I developed methods that sample and test a reduced set of configurations that test all DD-instances of the components in a model. While effective, that approach is impractical for testing evolving systems, because no matter how much or how little a system changes, that approach will generate configurations that satisfy the DD-coverage for a modified system and will retest all the generated configurations. This is unnecessary because many of the generated configurations are to test DD-instances whose results were already known from previous testing sessions. Clearly, such unnecessary work should be avoided.

To remedy this limitation, I have improved the approach in the previous sec-

tion to support *incremental build testing*. As part of the approach, I present in this chapter (1) a new adequacy criterion for *incremental build testing*, (2) an algorithm for computing incremental testing obligations, given the test adequacy criterion and the changes to the system configuration space, (3) an algorithm for selecting small sets of configurations that efficiently fulfill the incremental testing obligations, and finally (4) optimization techniques that use artifacts and test results from previous test sessions to improve the configuration selection and test process.

5.1 Test Adequacy Criterion

To support incremental build testing I have extended the approach in the previous chapter to (1) identify a set of DD-instances that need to be tested given a set of changes to a system, and (2) compute a set of configurations that test those DD-instances.

Consider the running example from Figure 2.1. Suppose that during the last testing session 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 B_2 have been *untestable*. Now suppose that new versions of components B and D become avail-

able, and that the latest version of E, 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 versions B_3 and D_3. Next, the modified component is handled by removing the old version, E_3, and then adding a new version, E_4. For this example, the previous approach would produce a test plan with 56 component versions to build (Figure 5.2(a)). This is larger than necessary. Some DD-instances involving new or previously untested components (and their versions) need to be tested, but other unchanged DD-instances do not.

The types of changes that are relevant to build testing include adding or deleting components, component versions, dependencies or constraints. To deal with all such changes in a uniform way, I compute the set of DD-instances for both the old and new configuration models and then use a set differencing operation to compute the DD-instances to be tested. Assuming that individual component names and version identifiers always refer to the same underlying software components, the relationship between the DD-instances for two successive models for a system is easily described using a Venn diagram. Figure 5.1 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}

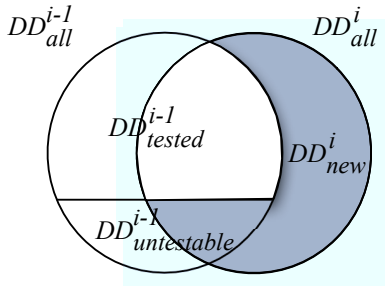


Figure 5.1: The DD-instances for two consecutive builds, $build_{i-1}$ and $build_i$. The DD-instances represented by the shaded areas need to be tested in $build_i$.

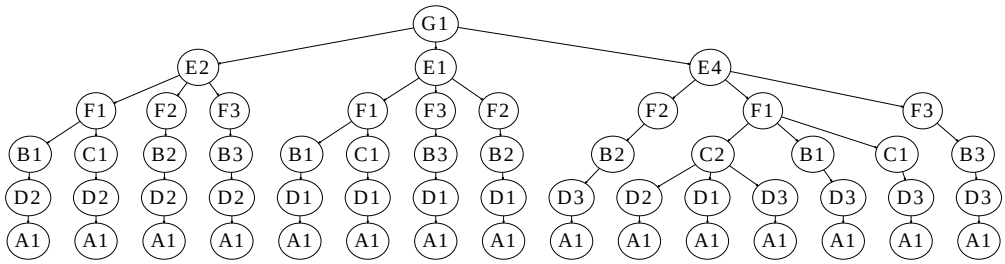
whose build status (success or failure) was determined in testing $build_{i-1}$ and $DD^{i-1}_{untestable}$ is the subset of DD^{i-1}_{all} whose build status is unknown – each of those DD-instances could not be tested because at least one of the component versions in the dependency part of the DD-instance could not be built 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:

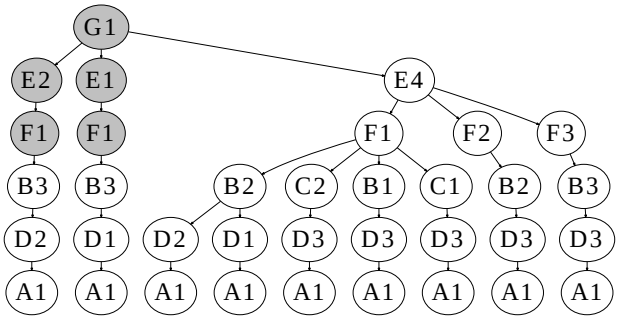
$$DD^{i}_{test} = DD^{i}_{all} - DD^{i-1}_{tested}$$

Previously untestable DD-instances are included in the current testing obligations, as newly introduced component versions might provide new ways to build a given component, thus enabling previously untestable components to be tested.

As just described, the set DD^{i}_{test} contains all DD-instances that must be tested for $build_i$. The next step applies the **BuildCFG** algorithm as many times as necessary to generate a set of configurations that cover all the DD-instances in DD^{i}_{test}.



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



(b) An incremental test plan

Figure 5.2: Test plans: Retest-All (56 components) vs. Incremental (35 components). The shaded nodes can also be reused from the previous test session.

The algorithm is applied to generate configurations *only* for DD-instances that have not yet been covered by any generated configuration, and also applied starting from DD-instances that build components closest to the top node in a CDG. When finished the configurations are merged into a test plan and then executed. An outline of this process is as follows:

1. Compute DD^i_{test}.
2. Select the DD-instance from DD^i_{test} that is 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 5.2(a)) to 35 components (Figure 5.2(b)). As the test plan executes, Rachet caches *partially-built configurations* (prefixes in the test plan) on the client machines when a prefix can be reused later in the test process, to speed up testing longer prefixes that share the prefix. As a result, for the running example, the total number of components to build is only 30, because the 5 components depicted by shaded nodes in Figure 5.2(b) have already been built in the previous test session and those partial configurations were cached (assuming that those partial configurations were not deleted at the end of testing).

In the following sections, I will explore performance benefits that can be achieved by better using the partial configurations that have been cached on the client machines.

5.2 Cache-Aware Configuration Generation

The approach in Chapter 4 assumed that the cache space of each client machine is empty at the beginning of each test session. 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 prefixes in the cache are shared by at least one configuration generated for the new test session. This section describes a method that uses information about cached prefixes from previous test sessions 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 5.1 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, the algorithm first picks the *best* prefix, which is the one that requires the minimum number of additional DD-instances to turn the prefix into a full configuration that tests the DD-instance. Then in the 2.1.2, the **BuildCFG** algorithm is used to extend the prefix by adding DD-instances. It is possible that **BuildCFG** fails to generate a configuration by extending the best prefix, due to constraint violations. In that case, the algorithm repeats from 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. This process can be very costly, because the algorithm must check for constraint violations whenever a DD-instance is added to the configuration under construction. Therefore, in order to pick a prefix that will be extended into a full configuration for testing a DD-instance, we instead employ a heuristic. We first compute a sub-graph of the CDG for the system under test, starting at the node that represents the component for which the DD-instance is computed. Then, the best prefix is the one that contains the maximum number of DD-instances that are needed to build the components in the sub-graph. The rationale behind this heuristic is that fewer DD-instances may be needed, when we construct a configuration by extending a cached prefix that

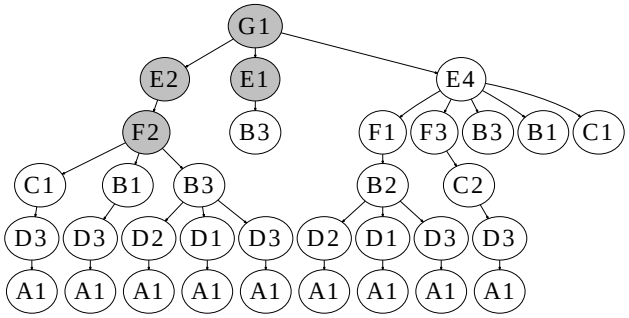


Figure 5.3: Test plan produced from configurations selected in a cache-aware manner. 34 component versions must be built. (Shaded nodes are cached, from the previous test session.)

already contains components required to test the DD-instance. Prefixes that contain DD-instances for components outside the sub-graph are not considered for the extension.

As previously discussed, running the **BuildCFG** algorithm with a prefix in the cache as an extension point may fail to generate a configuration, and it would improve performance if there is an efficient method to determine whether a configuration that covers a DD-instance can be generated by extending a given cached prefix, especially if a large number of cached prefixes is available. Although that decision cannot be made until the algorithm **BuildCFG** is applied, it is at least possible to check whether any constraint is violated when the DD-instance is added to the prefix. This is efficiently achieved by maintaining an auxiliary data structure called a *cache plan*, which is a prefix tree that combines prefixes in the

cache. (In Figure 5.3, 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 tested, 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, all prefixes reaching any node in the subtree starting at the node are filtered out.

Figure 5.3 shows a test plan created by merging the configurations generated by applying the cache-aware algorithm to the example system. The test plan has 34 nodes, 1 fewer than the test plan that does not consider cached prefixes (Figure 5.2(b)). The number of components that actually need to be built is 30 in both cases because prefixes in the cache can be reused. Note however, the average build sequence length decreases in the cache-aware plan by more than 1 component, because almost half the configurations are extended from cached prefixes.[1] This factor significantly decreases the turnaround time needed to complete the test plan.

[1]We average the number of components that must be built additionally when we reuse the best cached prefix for testing each configuration.

5.3 Managing Cached Configurations

If it were possible to cache all prefixes built during test plan execution for later use, the overall time required for executing each test plan would decrease, since the best prefix in the cache can always be reused. However, in practice, cache space is a limited resource, so when the cache is full a previously cached prefix must be discarded to add a new one. The approach in the previous chapter employs 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. However, this strategy does not take into account reuse across multiple test sessions.

In order to keep prefixes with more reuse potential longer in the cache throughout multiple test sessions, I have designed a heuristic that assesses the reuse potential of prefixes in the cache. The reuse potential consists of (1) the expected time that can be saved by reusing the prefix for executing the remaining portion of the

current test plan, and (2) the average change frequency of components contained in the prefix across previous test sessions. When a prefix in the cache needs to be replaced, the reuse potential has to be first computed for each prefix in the cache.

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 in the cache, we first identify, for each test 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. For our running example, in Figure 5.3, 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 prefix p_1 is 0, since prefix p_2 is the best prefix for testing all plan nodes in the subtree starting from p_1. Therefore, if a prefix in the cache has to be deleted, p_1 can be removed without increasing plan execution time, since a more useful one is available in the cache.

We also estimate how likely a prefix cached during the execution of a test plan is to be helpful for executing test plans in 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 the 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 for 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 for computing 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 Chapter 4, 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.

5.4 Evaluation

Having developed a foundation for incremental build testing of evolving component-based systems, this section describes evaluation results obtained by applying the presented approach on the subject systems described in Chapter 1.

For this study, I tested InterComm, PETSc and other components required for building InterComm and PETSc; for each I used the change history over a 5 year period. To limit the scope of the study, this 5 year period is divided into 20 epochs, each lasting approximately 3 months. I took a snapshot of the entire system at the end of each epoch, producing a sequence of 20 snapshots. In the remainder of the dissertation, these snapshots are referred to *builds* and the sequence of models for the builds are used as the model for testing the evolving InterComm and PETSc systems.

5.4.1 Modeling the Subject Systems

We first modeled the configuration space of InterComm and PETSc. This involved creating the CDGs, and specifying version annotations and constraints. Two types of version identifiers are considered – one is for identifying versions officially released by component developers, and the other is for the change his-

tory of branches (or tags) in source code repositories. Currently, the modeling is done manually based on careful inspection of the documents that describe build sequences, dependencies and constraints for each component.

Figure 5.4 depicts the dependencies between components for one build of InterComm and PETSc. Table 4.1 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 that captures the configuration space for InterComm for that build does not contain the Fortran component and all its related dependencies (black nodes in the figure).

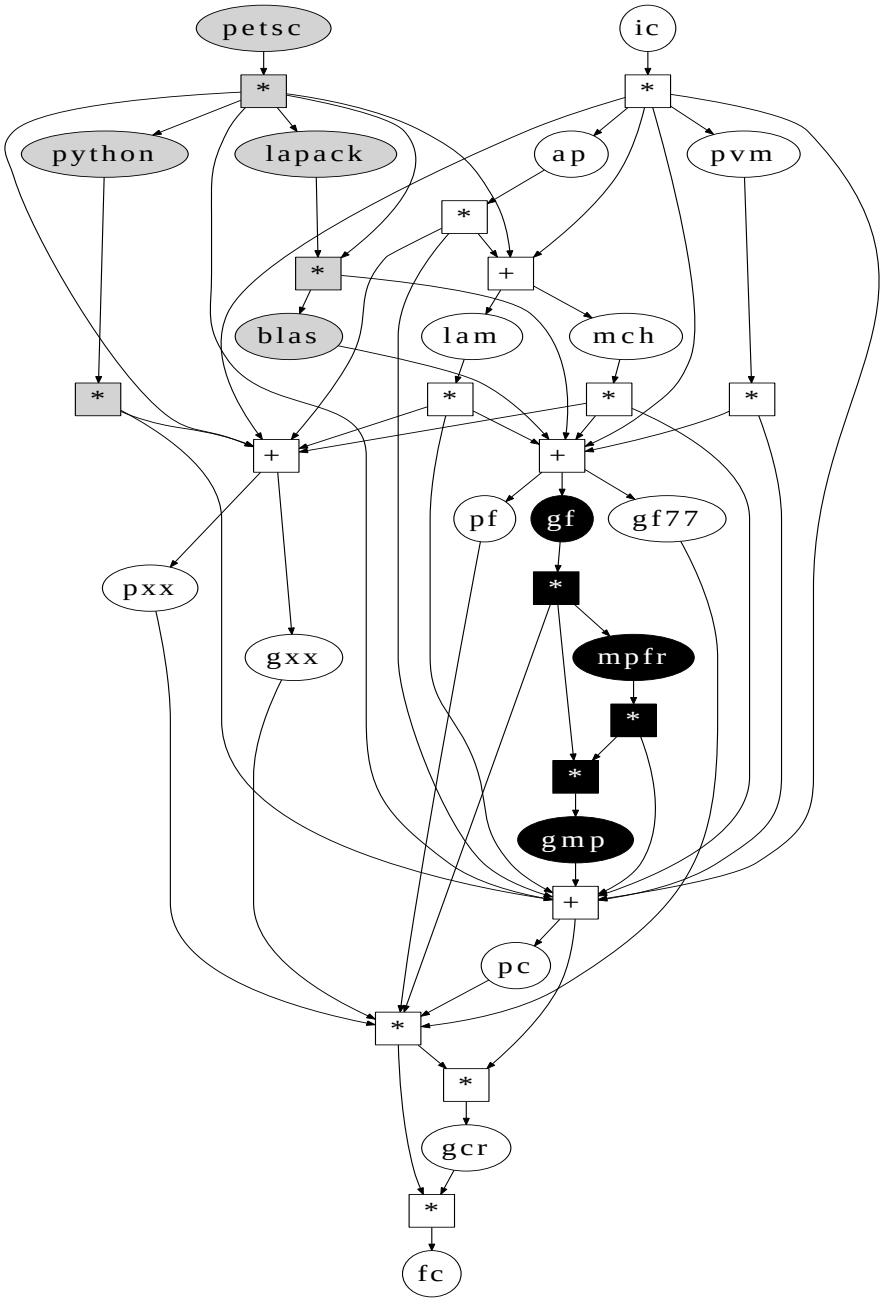


Figure 5.4: A Combined CDG for InterComm and PETSc. (Grey nodes are specific for PETSc. Black nodes are dependencies required for `gf` version `4.0.0` or later)

Table 5.1: History of version releases and code changes for components in the InterComm and PETSc builds

		Development Branches						**Version Release**										
Build	Date	ic	gcr gxx	gf77	gf	gmp	mpfr	ic	gcr gxx	gf77	gf	gmp	mpfr	lam	pvm	petsc	lapack	python
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	2.2.0	3.0	2.3.4
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														2.3.5,2.4
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			2.2.1		2.4.1
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					2.4.2
6	02/25/06		3.4d7, 4.0d4	3.4d7	4.0d4				3.4.5	3.4.5						2.3.0		
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											2.3.1		2.3.6,2.5
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					2.3.2	3.1.1	2.4.3,2.5.1
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					2.4.4
14	02/25/08		4.1d8		4.1d8		2.3d2						2.3.1					2.5.2
15	05/25/08		4.1d9		4.1d9													2.3.7,2.4.5
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			2.3.3		
18	02/25/09		4.1d11		4.1d11		2.3d5											2.4.6,2.5.3,2.5.4
19	05/25/09					4.3d1						4.3.0,4.3.1						

Table 5.1 shows the history of releases and source code changes for the components in each build. Each row corresponds to a specific build date (a snapshot), and each column corresponds to a component. For each build, entries in the last 11 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 ($build_7$).[2] 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 **Development 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.[3] For example, `1.1d2` in the third column of $build_5$ indicates that there were file changes in the InterComm development 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.

[2]The actual release date was 05/05/2006.

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

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 5.1 does not include versions for 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 (`mch`, `ap` and PGI compilers) or we considered only one version to limit the required test effort (`fc`). For this study, we assumed that these versions were available from the first build date. Also, we considered only 4 major GNU compiler versions and 3 major Python versions, due to the limited test resource availability for the experiments.

In addition to the CDGs and version annotations, InterComm places several constraints on configurations. First, 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. Second, only a single MPI component (i.e., `lam` or `mch`) can be used in a configuration. Third, only one C++ compiler, and only one of its versions (`gxx` version X or `pxx` version Y) can be

used in a configuration. Fourth, if both a C and a C++ compiler are used in a configuration, they must be from the same vendor (i.e., GNU Project or PGI). Fifth, compilers from the same vendor must be used to build the MPI components. Finally, the GNU compilers must be used for building `mpfr` in a configuration if `gf` is also contained in the configuration. These constraints eliminated configurations that we knew a priori would not build successfully.

5.4.2 Study Setup

To evaluate our incremental testing approach, we first gathered component compatibility data (i.e., the success or failure of each DD-instance) and the time required to build each component version. To obtain this data, we created a single configuration space model that contains identifiers for all released component versions and all branch snapshots 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 2.6.9-78.0.13.EL) and 32 client machines (Pentium 4 2.8GHz Dual-CPU machines with 1GB memory), all running Red Hat Enterprise Linux version 2.6.9-78.0.17.ELsmp, connected via Fast Ethernet. To support the experiment, we enhanced Rachet to work with multiple source code repositories, to retrieve source code for development branches. Currently, Rachet supports

CVS [5], SVN [4] and Mercurial [3] source code management systems.

For testing the InterComm builds, we obtained compatibility results for 15128 DD-instances. 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 build one or more components in the dependency part of the DD-instances. For example, all the DD-instances involving an InterComm version and the PVM component version `3.2.6` were untestable, because building that PVM version failed in all possible ways. For testing the PETSc builds, compatibility results for 24708 DD-instances were required. We obtained successful component builds for 6497 DD-instances and failed builds for 12883 DD-instances. 5328 DD-instances were untestable.

Using the data obtained from the integrated configuration space, we simulated a variety of use cases with different combinations of client machines and cache sizes. Our event-driven simulator, described in Section 4.9.2, used results obtained from the experimental run for calculating expected times required to execute test plans for the builds described in Section 5.4.1. Table 5.2 and 5.3 show the number of DD-instances that correspond to each region in the diagram in Figure 5.1.

For the i-th build in the InterComm and PETSc build sequences, the second column in the tables is the total number of DD-instances (DD^i_{all}) for building

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	577
2	403	141	186	76	170
3	781	141	186	454	756
4	945	271	320	354	809
5	1129	287	255	587	1154
6	1229	411	498	320	561
7	2480	416	341	1723	2854
8	2921	981	1170	770	1016
9	2921	1050	1488	383	758
10	4407	981	1170	2256	3546
11	4407	1450	1886	1071	1662
12	4407	1585	1940	882	904
13	5064	1585	1940	1539	2236
14	5296	2031	2514	751	1742
15	5296	2355	2622	319	706
16	5576	2193	2568	815	1840
17	6146	2586	2728	832	1607
18	6146	2877	2622	647	1721
19	7073	3301	2844	928	1745

Table 5.2: 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	55	0	0	55	74
1	85	39	0	46	52
2	133	61	8	64	155
3	499	95	14	390	746
4	627	291	90	246	520
5	852	347	147	358	808
6	1103	489	201	413	870
7	1993	657	254	1082	2274
8	1993	1342	404	247	518
9	2930	1342	404	1184	2318
10	4437	1933	668	1836	3909
11	9332	3076	950	5306	8319
12	9332	6667	2192	473	774
13	10030	6667	2192	1171	2828
14	11041	7009	2246	1786	3895
15	12583	7871	2554	2158	2684
16	12879	8825	2860	1194	3410
17	15415	9685	2932	2798	7669
18	17287	10901	3394	2992	5887
19	18214	13270	4016	928	3164

Table 5.3: Numbers of DD-instances for the PETSc build sequence

components in a 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 version identifiers of development branches with more recent ones. The next column is the number of DD-instances in DD^i_{all}, where results for the DD-instances were already determined in testing previous builds. The fourth column is the number of DD-instances in DD^i_{all}, where results for the DD-instances were *untestable* in the previous build. The last column is the number of nodes in the initial test plan for each build. In some builds, 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 generate the set of configurations that are merged into the test plans for the builds.

We ran the simulations with 4, 8, 16 and 32 client machines, each having 4 to 2048 cache entries. To distribute configurations, we used the plan execution strategy described in Section 5.3. For each machine-cache combination, we conducted multiple simulations to test the InterComm and PETSc 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 ($DD^i_{test} = DD^i_{all} - DD^{i-1}_{tested}$), (3) *c-forward*: *test-diff* with forwarding cached configurations across builds, (4) *new-replace*: *c-*

forward plus applying the improved cache management scheme (Section 5.3), (5) *c-aware*: *c-forward* plus applying cache aware configuration generation (Section 5.2), (6) *integrate-all*: *c-forward* also applying all optimization techniques. We measured the turnaround time for testing each build in the sequence, for all the simulations.

5.4.3 Retest All vs. Incremental Test

The configuration space for the subject systems grows over time because it incorporates more component versions. As a result, incremental testing is expected to be more effective for later builds. Figure 5.5 depicts the turnaround times for testing all 20 builds of InterComm and PETSc. 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 builds of InterComm and PETSc, *retest-all* takes about 6 days and 18 days, respectively, while incremental testing takes about one day for both systems.

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 testing time varied depending on the number of DD-instances tested by generated

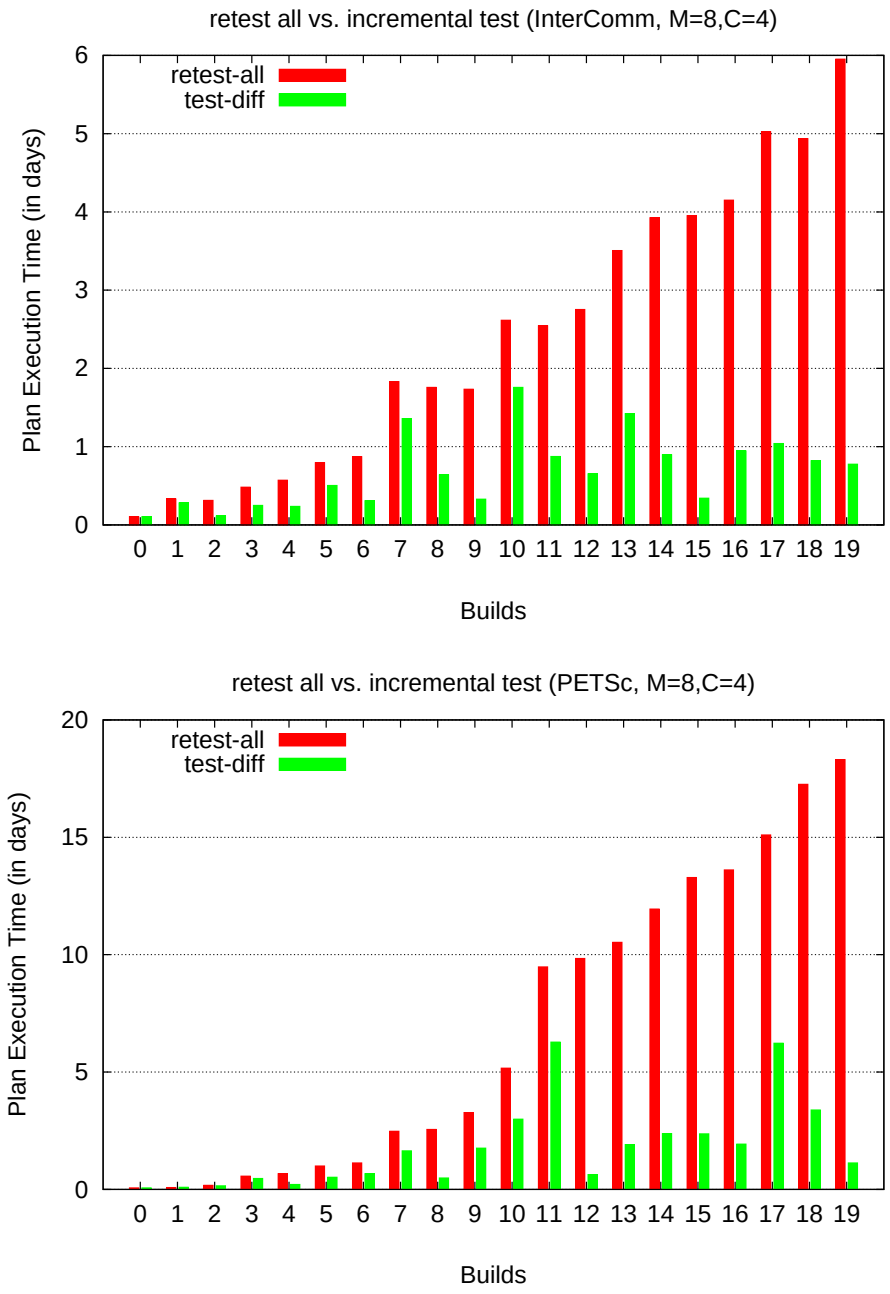
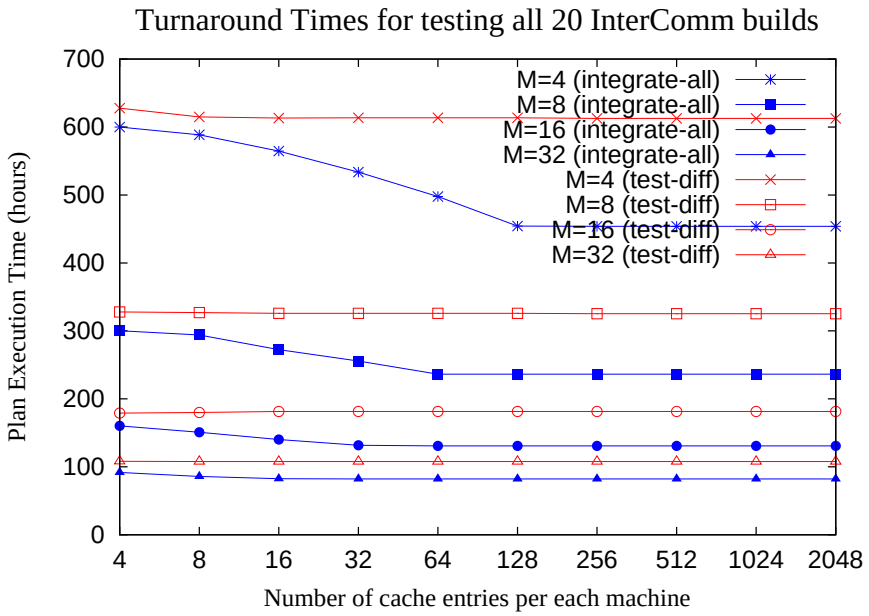


Figure 5.5: Turnaround times for testing DD^i_{all} and $DD^i_{all} - DD^{i-1}_{tested}$ for each build of InterComm and PETSc (8 machines (M=8) and 4 cache entries per machine (C=4))

configurations. For example, as seen in Table 5.2, 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 twice as much as the time for build 15. The reason is that 857 DD-instances in DD^i_{test} 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 that test those DD-instances due to build failures identified in earlier builds. We observe similar patterns for build 10 and 12 of PETSc. For build 12, we were able to generate configurations that cover only 485 DD-instances out of the 2665 DD-instances in DD^i_{test}.

5.4.4 Benefits from Optimization Techniques

Figure 5.6 depicts aggregated turnaround times required for testing 20 builds of InterComm and PETSc. The turnaround times are obtained by running incremental testing without reusing cached prefixes across builds (*test-diff*) and by running incremental testing with all optimization techniques applied (*integrate-all*). The x-axis is the number of cache entries per client and the y-axis is turnaround times. The simulations use 4 to 32 client machines and the number of cache entries per



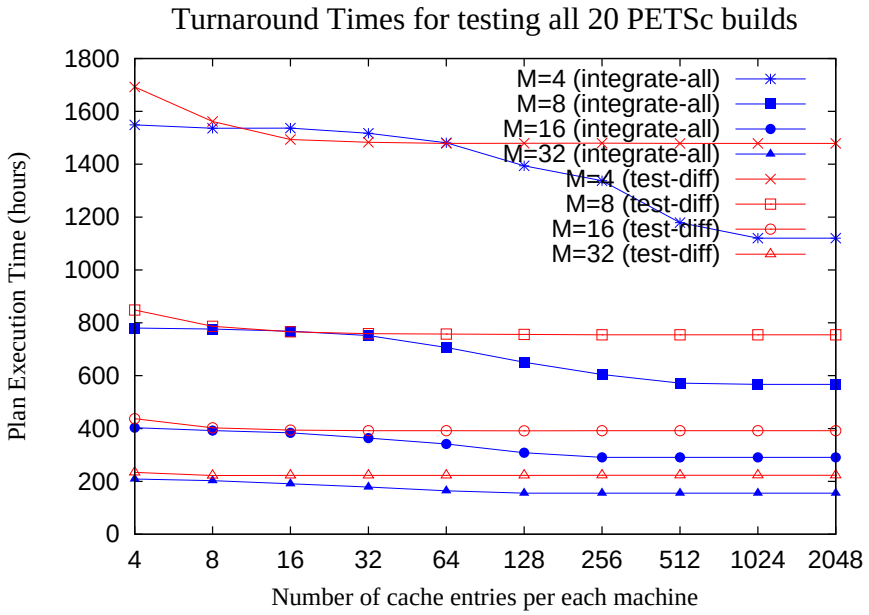


Figure 5.6: As the number of cache entries per machine increases, aggregated test cost decreases up to 24% for InterComm and up to 28% for PETSc when optimization techniques are applied, compared to the baseline incremental test.

machine varies from 4 to 2048.

As we increase the number of cache entries, we observe that the optimization techniques reduce turnaround times by up to 24% for InterComm and up to 28% for PETSc.[4] That is because a larger cache enables storing more prefixes between builds, so more configurations can be generated by extending prefixes in the cache and also cached prefixes can be more often reused for executing test plans in subsequent builds. On the other hand, for *test-diff*, we see few benefits from the increased cache size. The results for InterComm are consistent with results reported in our previous study [74], that little benefit was seen beyond a cache size of 8. Also, as described in Chapter 4, turnaround times decreased by almost half as the number of machines doubles. For PETSc, we did not observe further benefit beyond 16 or more cache entries per machine.

We also observed that the benefits from the optimization techniques decrease as more client machines are employed. For InterComm, with 4 machines, the turnaround time decreases by 24% when the number of cache entries per machine increases from 4 to 2048, but decreases by only 10% when 32 machines are used. Although this pattern is not clear for PETSc in Figure 5.6, in another simulation

[4]Even with 4 machines, turnaround times did not decrease further with more than 128 and 2048 cache entries per machine for InterComm and PETSc, respectively.

that employs 128 machines, we observed that the time decreases by 14% when we increase the cache size from 4 to 2048. There are two reasons for this effect. First, with more machines the benefits from the increased computational power offset the benefits from the intelligent cache reuse. With 32 or more machines, for InterComm builds, parallel test execution enables high performance even with only 4 cache entries per machine. Second, communication cost increases as more machines are used, because each machine is responsible for fewer nodes in a test plan and machines that finish their work faster will take work from other machines. In many cases, the best cached prefixes for the transferred work must be sent over the network for reuse.

As we previously noted, the cost savings from incremental testing vary depending on changes between builds. In Figure 5.7, 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.

For both InterComm and PETSc, we see significant cost reductions for several, but not all, builds – (1, 5, 7–8, 10–14) for InterComm, and (6, 9–11, 17) for PETSc. We found that for those builds there were new version releases for InterComm or PETSc. Since we have to first build all component versions required

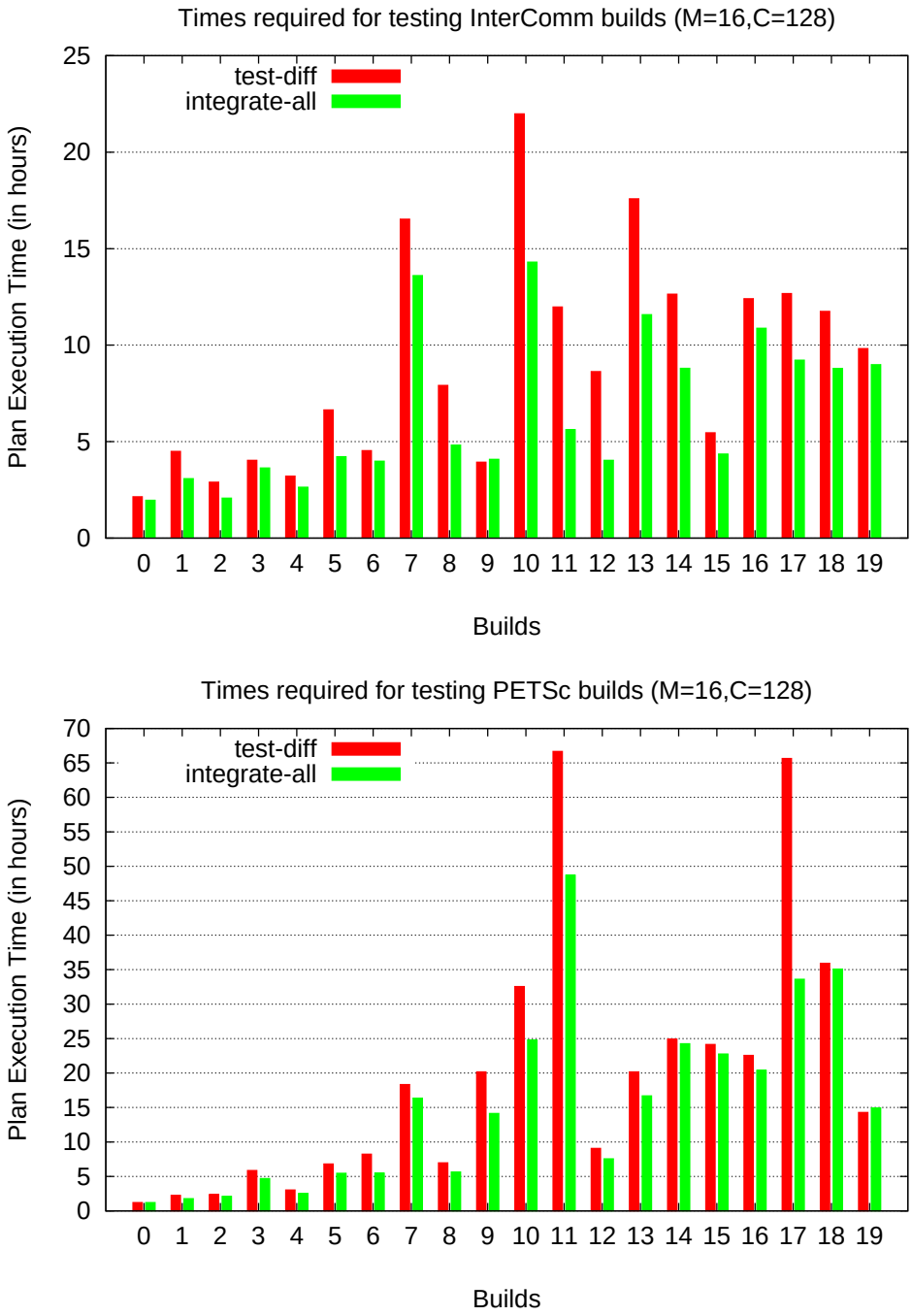


Figure 5.7: *test-diff* vs. *integrate-all*. There are significant cost savings for some builds from the optimization techniques.

for building InterComm or PETSc, we can significantly reduce the plan execution time for the builds of interest, by extending configurations that require adding fewer DD-instances in the process of configuration generation and also by reusing the configurations during test plan execution. In the results, for InterComm, we see a time decrease of more than 50% in build time for builds 11 and 12. For PETSc, we see a 49% decrease in time for build 17 and more than 30% for builds 6 and 9. On average, we see a 37% and 33% time reduction for the builds of interest for InterComm and PETSc, respectively.

The optimization techniques are heuristics, and do not always reduce testing time much. For example, there were smaller cost reduction for InterComm builds 0–4 and 15–19. 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 15–19, there were no changes for InterComm or for other components close to the top node in the CDGs, as seen in Table 5.1. Although the test plan sizes for those builds, as seen in Table 5.2, were comparable to those for other test plans where we achieved larger cost savings, for these builds we could only reuse configurations with fewer DD-instances that can be built quickly from an empty configuration, because changes are confined to components (e.g., compilers) close to the bottom

node in the CDGs. Similar results are seen in the results for PETSc. For instance, we see the maximum time reduction is for build 17, for which a new PETSc version (`2.3.3`) was available. Therefore, many prefixes in the cache could be reused for testing DD-instances for building the PETSc version. Other changes for the build were for components close to the bottom node of the CDG, and for the build the changes are not relevant to the DD-instances for PETSc.

5.4.5 Comparing Optimization Techniques

Figure 5.8 shows turnaround times for testing each build using 16 machines, with cache sizes of 4 (left) and 128 (right) per machine. We only show results for builds for which a new version is available for InterComm or PETSc, since we have seen large benefits for the builds in Figure 5.7 when both optimization techniques are applied. For each build of interest, we show results for five cases – *test-diff*, *c-forward*, *new-replace*, *c-aware* and *integrate-all*.

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

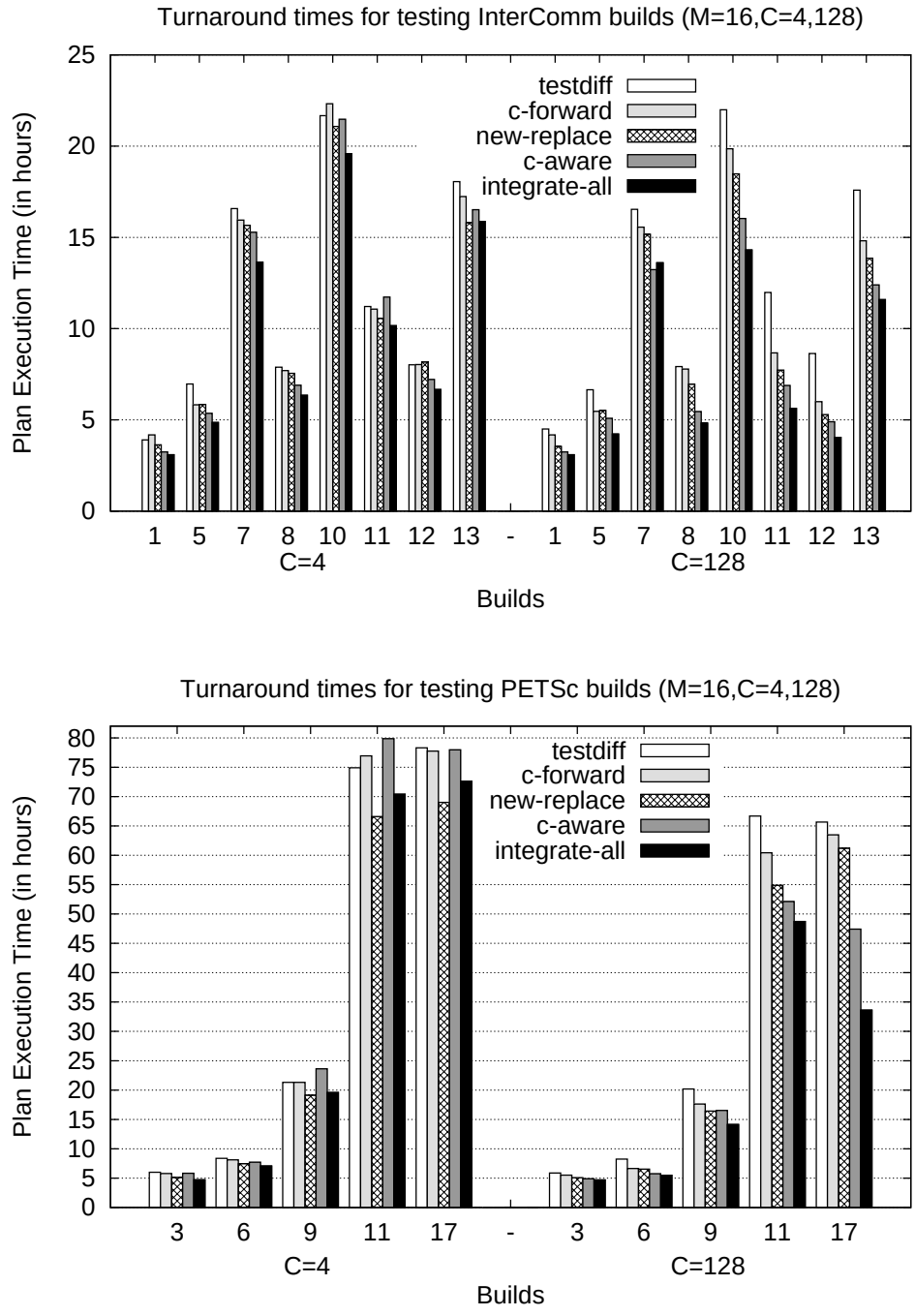


Figure 5.8: Each optimization technique contributes differently for different cache sizes.

in which configurations are cached and replaced.

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

With 128 cache entries, 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 configurations with a few additional DD-instances in the configuration generation step. Consequently, it takes less time to execute the resulting test plans. In both graphs, the new cache management policy (*new-replace*) did not greatly decrease test plan execution time. Since our plan execution strategy tests nodes in a test plan 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 bottom graph, with 128 cache entries per machine, we see some benefits for builds 11 and 17 with *test-diff*, compared to the cases with 4 cache entries. This is because the test plan sizes for the builds were large enough to get benefits by reusing prefixes cached in the process of executing the test plans for the builds.

In the InterComm simulation with 16 machines, each with 128 cache entries, there was no cache replacement for the entire build sequence. We still observe some additional time savings 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.

5.5 Summary

In this chapter, I have presented an approach that tests component compatibilities incrementally as components in a system model evolve over time. When there are component changes after the completion of a test session, I first identify the test obligations for the modified system and generate a set of configurations that satisfy the obligations. I also designed optimization techniques that make use of artifacts and test results from prior test sessions for the configuration selection and test process.

To identify the test obligations, I defined a test adequacy criterion, which is to test DD-instances that are *newly introduced* in a model or that were *untestable* in the previous test session, due to component build failures in the session. The test obligations are computed by applying set difference operations between the

sets of DD-instances between two consecutive builds. I applied the algorithm described in Chapter 4 for generating configurations that satisfy the test adequacy criterion. The results from simulations performed over the 5-year evolution history of components required to build InterComm and PETSc demonstrate that the plan execution time can decrease by a large amount by employing the incremental testing approach, compared to retesting all DD-instances from scratch for each build.

To further reduce the time to test DD-instances for a build, I developed two optimization techniques. One is to generate configurations for testing the DD-instances by extending prefixes cached in prior test sessions. This technique enables saving test effort by increasing the reuse chances of cached prefixes. The other technique is to keep prefixes with more reuse potential longer in the cache. I designed a heuristic that assesses the reuse potential of each prefix in the cache, by computing the expected time savings by reusing the prefix and how likely the prefix is to be useful for testing subsequent builds. The optimization techniques contributed together to significantly reduce the time required for executing test plans.

Chapter 6

Prioritizing Configurations with User Preference

Software developers are often more interested in some configurations than others. For instance, they may be more interested in configurations that test recently changed components or more popular versions of particular components. Considering that the time for compatibility testing is often limited, we need a systematic method for prioritizing configurations to quickly provide developers with results for configurations that test compatibilities between components and versions of more interest. In the software engineering community, there have been studies that prioritize test cases by incorporating user requirements or test history [46, 62], especially for regression testing [18, 35, 41, 63, 71], when the time and resources available for testing are limited.

Similar to the studies for regression testing that consider user requirements for

prioritizing test cases, I present in this chapter a method for testing configurations, taking into account differing developer preferences [76].

This chapter first describes a simple method for developers to specify their preferences, then discusses how the preferences are used for scheduling the order in which configurations are tested. Key objectives are to efficiently test configurations sampled by applying the **DD-Coverage** algorithm described in Chapter 4, and to obtain test results for configurations with higher preference before ones with lower preference.

6.1 Specifying Preferences

Modern systems can have an enormous number of configurations, and it is impractical for developers to explicitly specify preferences for all configurations to be tested. Therefore, we use a simple method in which developers first express preferences across all components in a CDG. Then for each component developers express preferences across all versions of that component.

Developer preferences may be represented in any form that specifies relative interest across components and their versions. For this work, we encode version preferences as positive integer values with larger values indicating higher prefer-

ence. If developers do not care to specify preferences for particular components or component versions, we assign a default preference – lower than all developer-specified preferences – in which components closer to the *top node* of the CDG are preferred over lower ones, and more recent versions of a component are preferred over older ones.

We interpret these partial preference assignments as capturing the developers' preferred ordering of test results. That is, we expect that developers want test results for configurations that test high-preference versions of high-preference components before test results for other configurations.

6.2 Computing Configuration Preferences

We now give more details on how to use developer preferences to guide the test process. Given a set of developer preferences, we could just opportunistically test configurations that contain the most preferred components and component versions first. That could be quite inefficient. As shown in Chapter 4, intelligently coordinating build effort across multiple configurations can save substantial time and effort. Thus, our testing approach needs to consider not only developer preferences, but also the structure of the test plan so that total test effort can be reduced.

To this end, we transform developer preferences (expressed over components and component versions) into preferences over nodes in a test plan. Recall that every prefix in a test plan (i.e., every path from the root node to a node in the test plan) corresponds to a *partial configuration*, and testing a plan node means building all components represented by the nodes on the path. Therefore, when we test a prefix from scratch, we build components on top of an empty configuration, starting from the one represented by the plan root, while testing a prefix by reusing a cached prefix means the builds start from somewhere in the middle of the path.

Logically, the preference for a component version is represented as a vector, called a *preference vector*. A preference vector has one element for each component in a CDG, with these elements ordered by component preference.[1] Values of the elements are assigned as follows: for a component C with version c_v, each element takes the value 0, except for the element associated with C, whose values takes the preference assignment of c_v. For example, consider the example system shown in Figure 2.1. Assume that the components A–G have preferences from

[1]To simplify the presentation we restrict discussion to cases in which preference assignments for components are unique. `Rachet` can, however, handle non-unique component preferences by allowing an element in a preference vector to encode version preferences for multiple components with identical preferences.

	Component & Version						
Pref. Value	A	B	C	D	E	F	G
1	A_1	B_1	C_1	D_1	E_1	F_1	G_1
2		B_2	C_2	D_2	E_2	F_2	
3					E_3	F_3	

Table 6.1: Example Preference Assignments (Bigger values are used for higher preferences)

7 down to 1 respectively. Assume further that the version preferences for each component are sequentially numbered preferences starting at 1, which is the lowest preference for the oldest version of the component. This data is represented graphically in Table 6.1.

Given these preference assignments, we write the preference vector for component B, version B_2, for example, as $(0, 2, 0, 0, 0, 0, 0)$. This is because B is the second highest preferred component and because version B_2 has preference assignment 2. Similarly, the preference vector for component F, version F_3 is $(0, 0, 0, 0, 0, 3, 0)$

Given the component preference vectors, we can now define preference vectors for every prefix in a test plan. For a given prefix, we do this by taking a component-wise vector sum of each node contained in the prefix. For example, looking back at Figure 4.3, consider the gray-shaded leaf node with the label starting with A_1. To build that node from scratch, we must build each node from the

root of the test plan to this node. Therefore, we compute the preference vector for this entire prefix by summing the preference vectors of all nodes appearing in that path: $\mathsf{G}_1 : (0,0,0,0,0,0,1)$, $\mathsf{E}_2 : (0,0,0,0,2,0,0)$, $\mathsf{B}_1 : (0,1,0,0,0,0,0)$, $\mathsf{F}_1 : (0,0,0,0,0,1,0)$, $\mathsf{D}_2 : (0,0,0,2,0,0,0)$, $\mathsf{A}_1 : (1,0,0,0,0,0,0)$. This gives a resulting preference vector of $(1,1,0,2,2,1,1)$. Note that the third vector element is zero because component C is not contained in the prefix.

In the next section we describe how we use preference vectors assigned to all prefixes in a test plan to guide the execution of the test plan.

6.3 Preference-Guided Plan Execution

As earlier, each test client repeatedly requests tests to perform from a test server – tests are encoded as prefixes (partial configurations) in the test plan under execution. For each request, the server selects the prefix to be tested next in a greedy fashion, by first ordering preference vectors computed for each prefix that ends with a plan node that has not yet been tested by any client. Logically, this ordering can be done by sorting preference vectors for all prefixes lexicographically. We give the description of the selection process in Figure 6.1.

In the algorithm, when a client requests a new configuration to test, we first

Algorithm **Prioritized-Execution**(*Plan, C, W*)

1: // *C*: requesting client, *W*: window size
2: *PrefixList* $\leftarrow$ an empty list
3: **for** each prefix p for not-yet assigned node $n \in Plan$ **do**
4: $pref_p \leftarrow$ the preference vector for p
5: $l_p \leftarrow$ the minimum number of components needed additionally for testing p on a prefix cached in *C*
6: $d_p \leftarrow n$'s depth in *Plan*
7: $steal_p \leftarrow$ the number of clients that contain a prefix reusable for testing p
8: Add $\{< p, pref_p, l_p, steal_p, d_p >\}$ to the list *PrefixList*
9: **end for**
10: Sort *PrefixList* by preference vectors
11: **if** no prefix cached in *C* is a prefix of any of the first *W* prefixes in *PrefixList* **then**
12: **return**, among the first *W* prefixes, the prefix with minimum $steal_p$, using minimum d_p as tie breaker
13: **else**
14: **return**, among the first *W* prefixes, the prefix with minimum l_p, using maximum d_p as tie breaker
15: **end if**

Figure 6.1: Algorithm for Preference-Guided Plan Execution

compute, for each prefix p in a test plan, three auxiliary variables used in the prioritized plan execution to reduce overall plan execution time. The algorithm also uses a parameter called a *window size*, which is related to trading off between reducing test effort and enforcing preferences and will be discussed further in this section.

The variable l_p is the minimum number of component versions that must be built additionally, when the client tests p by reusing a prefix previously cached in the client. When multiple clients are employed for executing the test plan, testing prefixes with the smallest l_p value can increase benefits from reusing cached prefixes.

The variable $steal_p$ is the number of clients that have in their cache space at least one prefix reusable for testing p. This is used for reducing redundant work across clients. When there is no prefix (in a test plan) that can be tested by reusing a prefix cached in the client, the server dispatches the prefix with the smallest $steal_p$ to the client, and this enables multiple clients to test non-overlapping regions of the test plan under execution.

The variable d_p is the number of DD-instances contained in p. It is used as the tie breaker when a test plan has multiple prefixes with the same l_p or $steal_p$ value for a given client. For a prefix request from a client, when there are multiple

prefixes with the same l_p value in the test plan, the server dispatches the prefix with maximum d_p value, the longest prefix. This is based on the heuristic that the longest prefix may require more time to build if it is deleted from the cache space. On the other hand, if the client must reuse a prefix cached in another client for testing any prefix that ends with a plan node that has not yet been tested, and if there are multiple prefixes with the same $steal_p$ value, the server dispatches the prefix with the minimum d_p value, the shortest prefix. This is based on the heuristic that a node closer to the root node will likely have a larger subtree beneath it than nodes deeper in the tree, as was done for the *hybrid* test plan execution strategy described in Section 4.6.

Note that Algorithm 6.1 has to be repeatedly applied for each client test request, since the plan execution state (including cache states at clients) changes continually, and the values for auxiliary variables are different depending on the client requesting a new test, and the current state of the test plan (i.e., which parts have been completed).

Although the auxiliary variables may be used to decrease overall plan execution time by efficiently sharing the effort necessary to test prefixes in a test plan, the most important concern for developers is still the preferences. Therefore, we sort the *PrefixList* in preference order, and always test the first prefix in the list,

which is the one with the highest preference vector, to produce results for more highly preferred prefixes earlier.[2] However, due to the large size of cached prefixes and the long time to transfer VMs across clients, scheduling prefixes by only taking into account preference values may increase the number of remote prefix requests and as a result, the rate of local cache reuses drops and total plan execution time can increase compared to a pure cost-based scheduling policy.

Rachet allows developers to determine how rigidly they want their preferences enforced. If they want to enforce preferences strongly, the test server always chooses the most highly preferred prefix that has not yet been tested by any client. If developers allow weaker preference enforcement, the scheduling considers other factors, such as prefix reuse locality and work redundancy across clients, which help to reduce the overall plan execution time, in exchange for allowing less highly preferred prefixes to be tested earlier.

The preference strength is expressed via a *window size* parameter, denoted by *W*. As shown in Algorithm 6.1, we inspect the first *W* prefixes in the *PrefixList* and select the one that requires the fewest component builds by reusing a prefix cached in that client. This means that less highly preferred prefixes can be chosen if they

[2]However, results may still not be produced in preference order because of varying component build times.

can be tested at low cost by reusing prefixes previously tested by that client. If such prefix reuse is not possible, the scheduling algorithm selects a prefix that has the smallest overlap with prefixes tested by other clients, to increase the chances for future reuse of the prefix.

An interesting case occurs when the window size is set to a value greater than or equal to the number of nodes in a test plan. In that case, with the scheduling policy described above, the test plan is executed in a similar order to the hybrid execution strategy.

6.4 Evaluation

We now evaluate our prioritization approach by constructing two scenarios that often occur during compatibility testing. In the first scenario, we prefer configurations that test more recent versions of the SUT. In the second scenario, we prefer configurations that use recent versions of specific components required for building the SUT. In this section, we describe these scenarios and apply our prioritization approach to the software systems described in Section 1. Then we analyze benefits and tradeoffs of the overall approach. Specifically, we want to compare our prioritization approach with the *hybrid* approach, which showed the best per-

formance in Chapter 4. We measured the times for executing the test plan created from configurations with DD-coverage for each system, and also recorded the times at which test results for configurations are produced. We also want to study the tradeoffs involved in varying the window size parameter of Algorithm 6.1, and in varying the number of clients and cache sizes used for executing the test plan.

6.4.1 Experiment Setup

For modeling subject systems for this study, we have extended the models for the two systems described in Section 4.9.1 by adding more versions for some components and also by specifying preferences on components and their versions. The component versions newly added are: version `1.1` and `1.6` for the InterComm (`ic`) component, version `2.3.2` for the MPFR (`mpfr`) component and version `4.2.4` for the GNU MP (`gmp`) component.

Scenarios

We constructed two scenarios to evaluate our approach. In the first scenario, developers want to test recent versions of the SUT, InterComm and PETSc. This scenario was actually encountered during the development of InterComm. When InterComm version 1.6 was released, InterComm developers wanted to test the

build-compatibility of the new InterComm version in various configurations. In addition, they also preferred to test configurations based on more recent versions of other components; they believed that a large portion of their user base had already updated the system components on their machines to recent versions. To meet this requirement, we assigned components preference ranks in the InterComm model by traversing the CDG in reverse topological order (i.e., InterComm had the highest component preferences). For version preferences, higher values are assigned to more recent component versions (the oldest version had value 1). We applied the same preference assignment method for the PETSc model.

In the second scenario, developers prefer to test configurations that contain recent versions of specific components required for building the SUT. That is, developers want to see first whether their systems are compatible with recently released versions of specific components. For this study, we model that developers prefer configurations that use recent versions of the MPFR and the GNU MP component for building InterComm and PETSc. Thus, we set high preference ranks for those components, and also set higher version preference values for recent versions.

Compatibility Results for DD-instances

From the CDG and annotations, there are a total of 639 DD-instances for components in the InterComm CDG, and Rachet produced 476 configurations satisfying the DD-coverage criterion. These configurations contain 4421 component builds, but the actual number of component builds is reduced to 1908 since the configurations are combined into a single test plan. For PETSc, there are 185 DD-instances and 88 configurations that contain 846 component builds, which is reduced to 522 component builds in the test plan.

In order to obtain compatibility results for DD-instances for components in the models, we first ran actual experiments with the test plans for InterComm and PETSc. For the InterComm model, 134 out of 639 DD-instances were tested without errors, which means that there were 134 successful ways to build component versions (18 for the top-level InterComm component). A total of 58 DD-instances failed to build (3 for InterComm). The remaining 447 DD-instances were untestable because there was no successful way to build at least one of the component versions needed to perform the component build for testing the DD-instances. For the PETSc model, 107 out of 185 DD-instances were tested successfully (8 for the top-level PETSc component) and 62 DD-instances failed (56 for PETSc). The remaining 16 DD-instances were untestable.

Simulations

In the remainder of this study, we use test results obtained by running the actual experiments, including the compatibility results for all DD-instances and the times required for building each component version. We run simulations with two test plan execution strategies (prioritized vs. hybrid), using different numbers of clients (4, 8, 16, 32), different number of cache entries per client (8, 16, 32, 64) and different window sizes (1, 16, 256, 2048) - window size only matters for the prioritized strategy. In all, for each scenario, we simulated the 80 possible combinations across the dimensions (16 for the hybrid and 64 for the prioritized strategy).

For each plan execution we recorded the time when testing configurations succeeded or failed. We say that testing a configuration *succeeded* if all component versions contained in the configuration are built without any error, and that testing a configuration *failed* if the process for building a component version contained in the configuration returned errors. Note that if a component version encoded by a DD-instance in a configuration failed, then all configurations that contain the DD-instance also fail. Thus, when a DD-instance appears in several branches in a test plan, all configurations that contain the DD-instance fail simultaneously.

6.4.2 Cost/Benefit Analysis of Prioritized Test

Prioritized vs. Hybrid

In Figure 6.2, we show the times at which testing configurations for the subject systems succeeded (shown as diamonds) or failed (shown as plus signs). The left graphs in the figure show results from executing the InterComm and PETSc test plans with the hybrid strategy for two test scenarios, and the right graphs show results with the prioritized strategy. The x-axis in each graph shows all configurations generated for executing the InterComm and PETSc test plan, sorted in their preference orders – the leftmost is the most preferred configuration). The y-axis shows the time at which test results for configurations are determined. In this result, we set the window size to 1 and the number of client machines to 4. From these plots, we clearly see that the prioritized strategy achieved results for highly preferred configurations quickly compared to the hybrid strategy. The hybrid strategy achieved some results for highly preferred configurations almost at the end of the plan execution.

Failed configurations formed multiple bands in the graphs for the prioritized strategy. This is because multiple configurations failed simultaneously when a component version encoded by a DD-instance failed to build. Many of those configurations were different by only one or two DD-instances so had similar

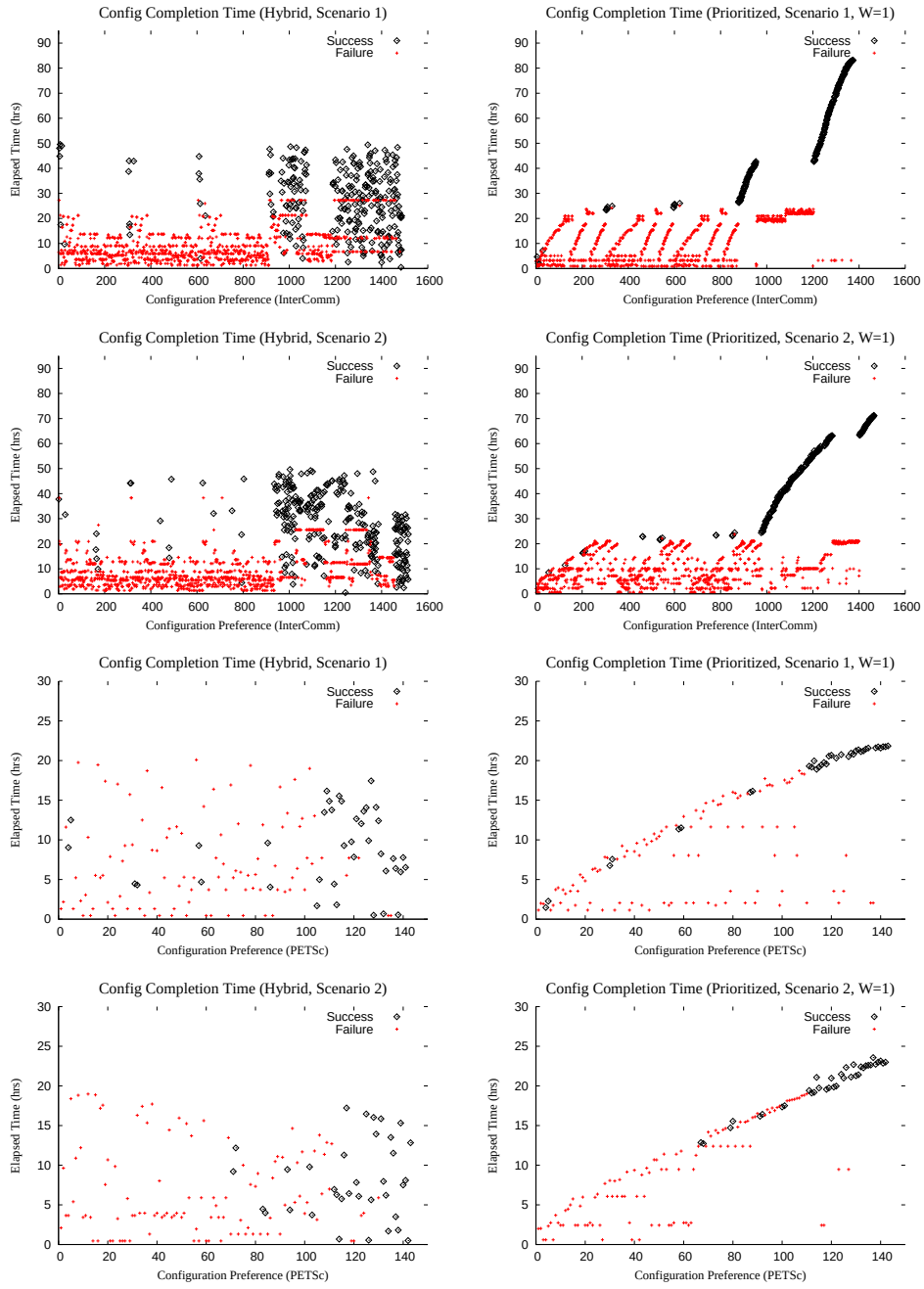


Figure 6.2: Prioritized (W=1) vs. Hybrid strategy for InterComm (top 4 graphs) and PETSc (bottom 4 graphs) with 4 clients (M=4) and 8 cache entries per client (C=8)

priority differences.

Each plot in the figure contains a different number of configurations for the prioritized and hybrid plan execution strategies because Rachet applies contingency planning when there are build failures, and so generates additional configurations to test those DD-instances affected by the failures in alternate ways. The number of additional configurations differs depending on the time and the order failures are discovered, and that is the reason that the graphs for InterComm contain almost ten times more configurations than those for PETSc. Although we initially produced 5 times more configurations for InterComm, more configurations were added to the test plan by contingency planning during the execution of the InterComm test plan.

We see that plan execution with the prioritized strategy took longer than with the hybrid strategy. For InterComm, by employing 4 machines, each with 8 cache entries per machine, the prioritized strategy took 84% more time for the first scenario and 44% for the second scenario, for a window size of 1. For PETSc, the prioritized strategy took 17% and 23% lower, respectively. This is mostly attributed to *low prefix reuse locality* during the execution of a test plan. When we strongly guide the plan execution by developers' preference, Rachet always schedules the most preferred prefix to a requesting client, without considering po-

tential cost savings from reusing a prefix already cached in the client. That is, in many cases, the newly dispatched prefix to the client does not share build effort with the prefixes tested previously by the client. As a result, to test the dispatched prefix, the client ends up reusing a prefix cached in another client, by transferring the VM with the reused prefix. This process is always more expensive than reusing a locally available prefix and increases the overall plan execution time.

Varying the Number of Cache Entries per Client

The prioritized strategy with the window size of 1 took more time in all cases for executing a test plan. However, as demonstrated in Figure 6.3, we observed that the plan execution time can decrease when there is more space for caching prefixes during the execution of a test plan. The top graph shows that for executing the InterComm test plan with the first scenario, the prioritized strategy took 84% more time with 8 cache entries per machine compared to the *hybrid* strategy, but 31% more with 64 cache entries.

With the prioritized plan execution, each client is more likely to test prefixes that do not share build effort with prefixes cached in the client and also the prefixes in the cache may be replaced before they are reused. Therefore, larger cache size can increase the reuse opportunities for cached prefixes and enables saving test

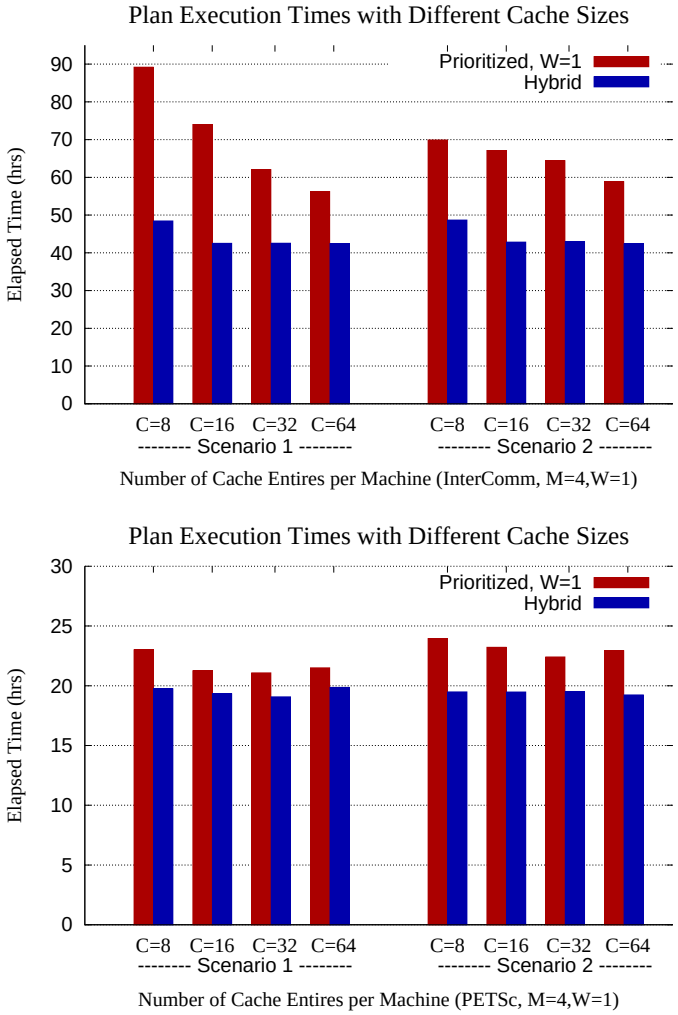


Figure 6.3: Turnaround time difference between the prioritized and the hybrid strategy with different cache entries per client

effort for building components contained in prefixes subsequently dispatched to the client. Note that the benefits from a larger cache are limited for the hybrid strategy, because the strategy executes a test plan mostly in depth-first order.

For the PETSc test plan, we did not see much benefit from additional cache entries. This is because the test plan has fewer branches than the InterComm test plan and also because we cache a prefix only if the last node of the prefix has two or more child nodes. Hence, fewer prefixes were cached during the execution of the test plan.

Varying Window Size

The algorithm in Figure 6.1 allows developers to control how strongly their preferences are enforced, by modifying the *window size* parameter. A window size of 1 means that developers only care about their preferences, not overall plan execution cost. In this case, Rachet always schedules the most preferred prefix for any client request. As the window size increases, Rachet considers other factors, including prefix reuse locality, that can reduce the overall plan execution time.

Figure 6.4 shows that the InterComm and PETSc test plans were executed faster with larger window sizes for both scenarios. When the window size is equal to or greater than the number of nodes in a test plan (the case with W = 2048), the

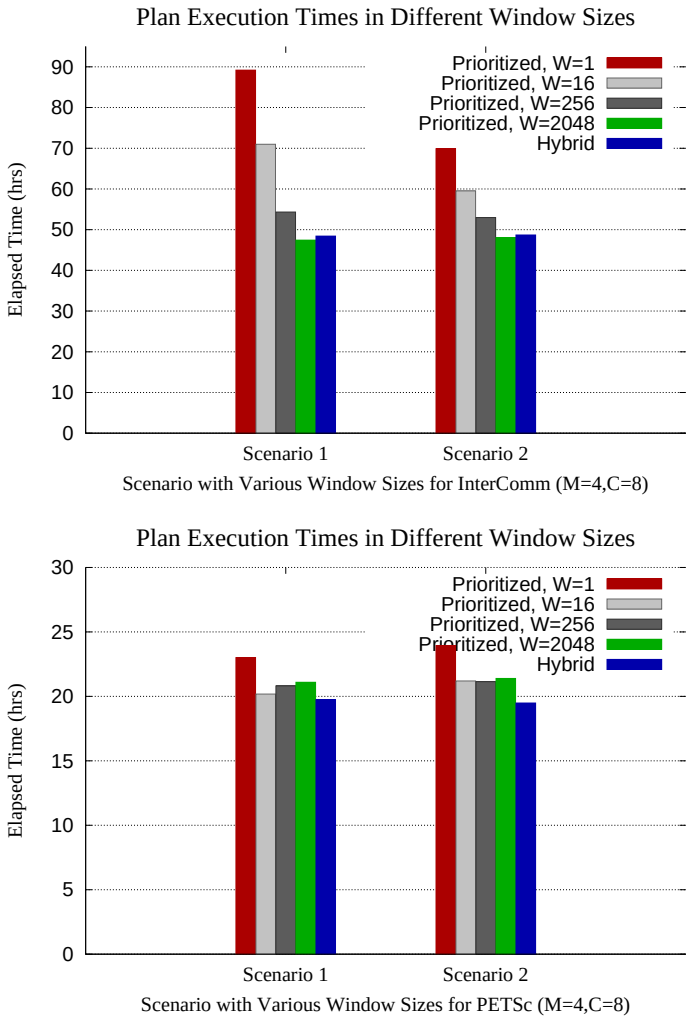


Figure 6.4: Turnaround Times for InterComm and PETSc Test Plan in Different Window Sizes

execution time with the prioritized strategy was comparable to the hybrid strategy. In this case the prioritized strategy ignores the developers preferences, and instead executes the test plan so as to maximize the reuse of prefixes cached in each client and to minimize redundant work across clients. This trend is less clear for the PETSc test plan execution, because the PETSc test plan is much smaller than the InterComm test plan and the benefit of larger window size comes from the increased chances to reuse cached prefixes.

The cost to gain the improved overall performance, as seen in Figure 6.5 and Figure 6.6, is that test results for less highly preferred configurations are produced earlier than some more highly preferred configurations with larger window sizes.

6.4.3 Quantitative Analysis

In the previous section, we measured the costs and benefits of the prioritized strategy by visually inspecting patterns in the scatter plots.

To evaluate the results more quantitatively, we developed a metric to measure conformance to preference order. Specifically, whenever the test result for a configuration is identified, we compute the ratio between: (1) the number of already tested configurations whose preferences were greater than the current configuration's preference and (2) the number of already tested configurations.

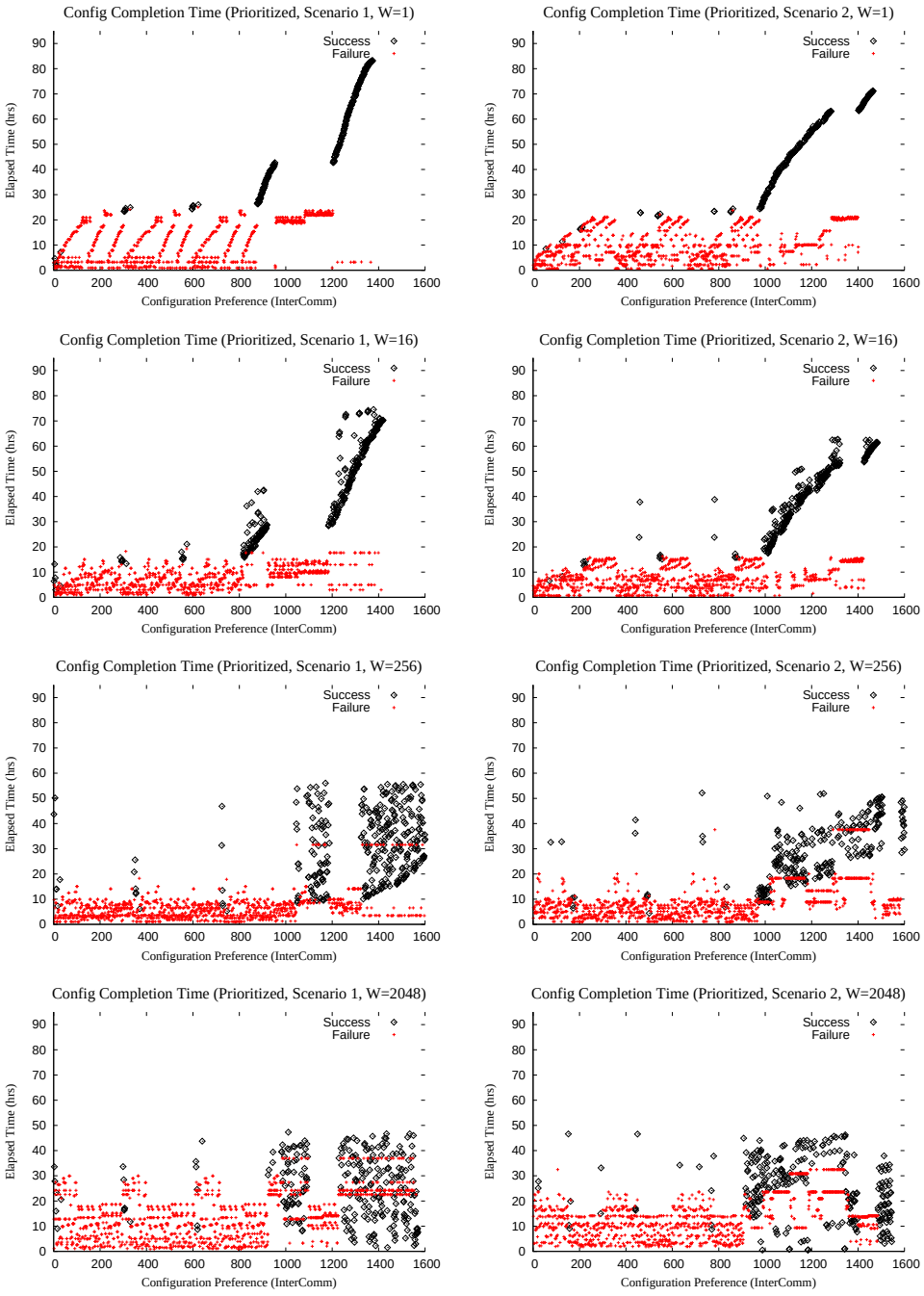


Figure 6.5: Configuration completion times with different window sizes (InterComm, Scenario 1 (left 4 graphs) and Scenario 2 (right 4 graphs), M=4, C=8

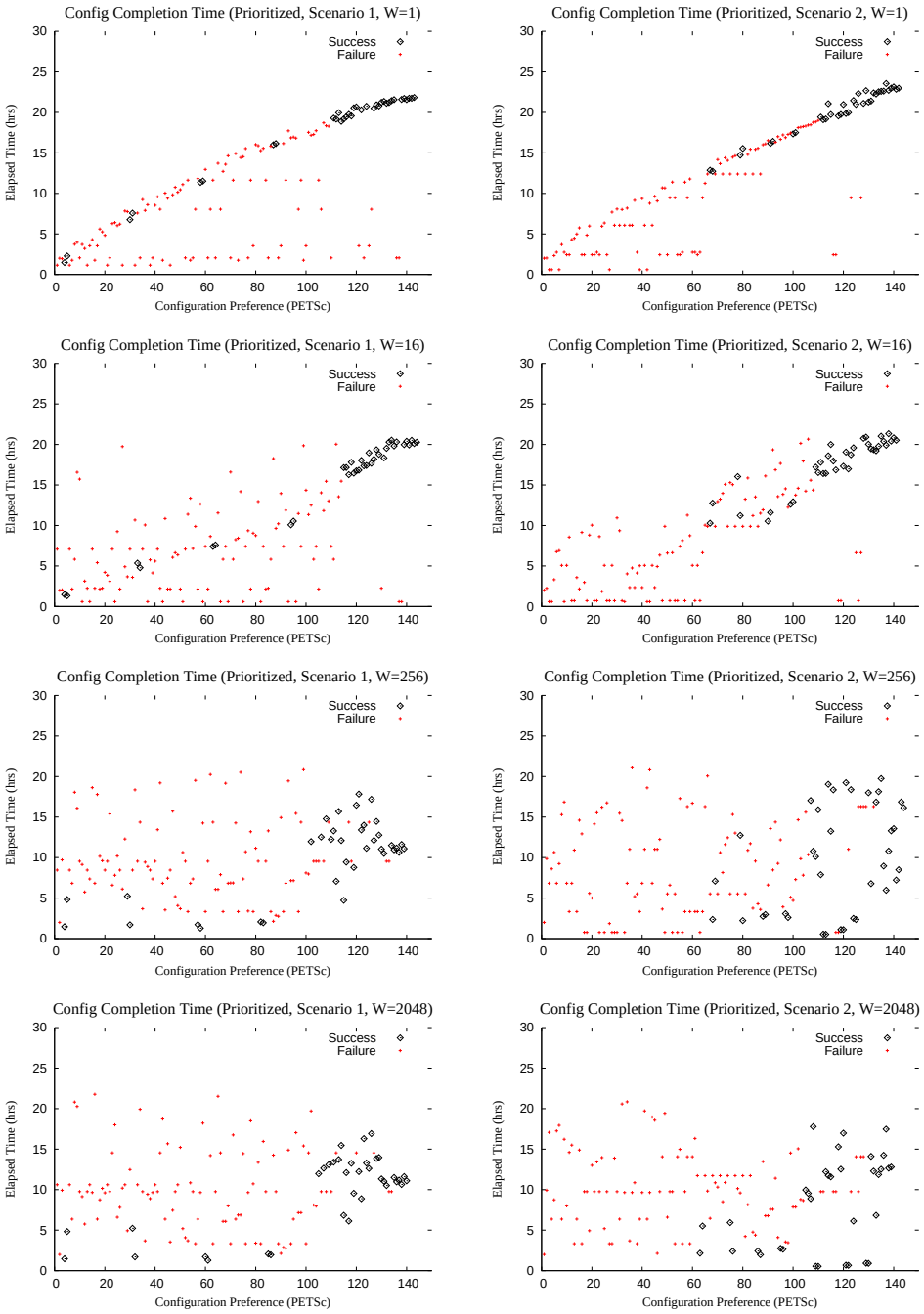


Figure 6.6: Configuration completion times with different window sizes (PETSc, Scenario 1 (left 4 graphs) and Scenario 2 (right 4 graphs), M=4, C=8

If all configurations finish testing in preference order, then this metric will always be 1. In fact, for many of our experiments using the prioritized strategy with a window size of 1, the metric stayed very close to 1.

However, in some cases test results come out of order. This occurs for several reasons. First, the times for building components contained in various configurations are different, and client machine speeds can also vary. Although we schedule more highly preferred configurations earlier, results for those configurations may be produced out of order. Second, when we fail to build a component version encoded by a DD-instance, multiple configurations that contain the DD-instance are classified as failures at the same time, while other more highly preferred configurations are still being executed.

Figure 6.7 and 6.8 show conformance to preference order for successfully tested configurations, when we execute the InterComm and PETSc test plans for the two scenarios with the hybrid strategy and with the prioritized strategy, with different window sizes. The x-axis in the figures is the plan execution time, normalized to the range between 0 to 1, because plan execution times are different across simulations, and the y-axis is conformance to preference order.

Since test results for configurations can come out of preference order (even for the proritized strategy with the window size of 1), and the order is ignored in the

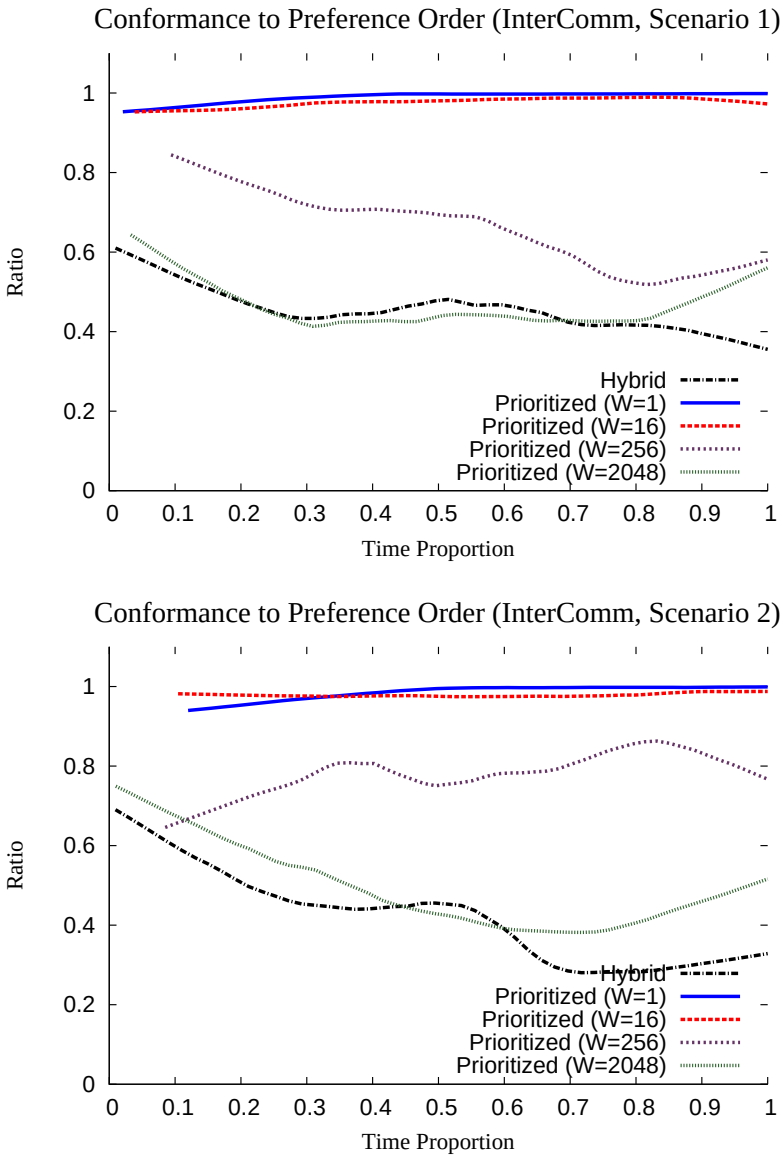


Figure 6.7: Conformance to preference order varying the window size (InterComm, M=4, C=8, Successfully tested configurations)

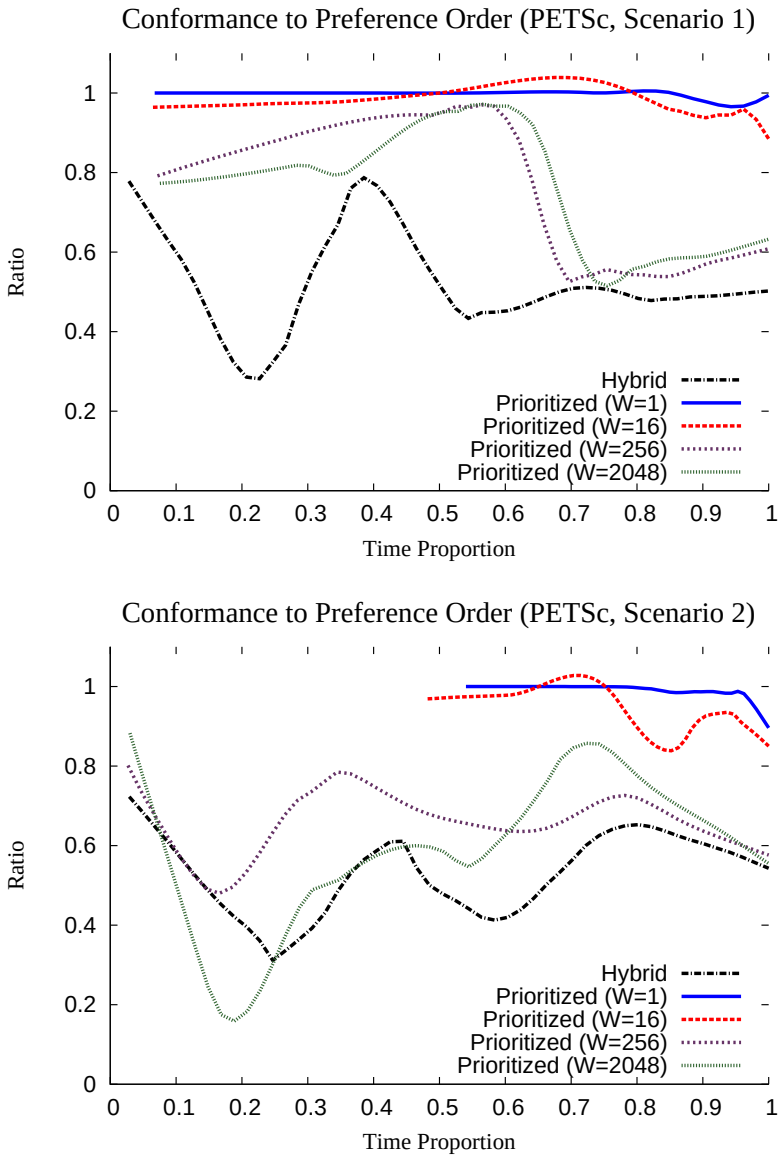


Figure 6.8: Conformance to preference order varing the window size (PETSc, M=4, C=8, Successfully tested configurations)

hybrid strategy and prioritized stratetgy with large window sizes, it was difficult to see a pattern if we merge test results from the simulations into a graph. Therefore, we applied a smoothing technique called *Loess smoothing* [21, 22] for making it easier to analyze the results. As seen in the figures, with a window size of 1 the plan execution conforms almost completely to developer preferences and that the degree of conformance drops as we increase the window size. An extreme case is when we execute the plans with a window size equal to the plan size. For that case, the prioritized strategy shows similar behavior to the hybrid strategy, since both strategies execute the test plan completely ignoring the developer specified preferences.

6.5 Summary

In this chapter, I have presented a systematic method for prioritizing the order in which configurations are tested, in order to obtain test results for configurations of higher importance rapidly, in resource-constrained situations. To accomplish this goal, I first designed a method for specifying developers' preferences over components and their versions, and then I used the preferences for determining priorities of configurations, by computing *preference vectors* for all configurations, refer-

ring to the specified component and version preferences.

I also presented a test plan execution strategy that schedules configurations considering both the priorities of configurations and the cost required for testing the configurations. In order to test more important configurations earlier during the test plan execution, I ordered configurations by priorities and then dispatched the one with the highest vector value to a client requesting a new configuration.

Results from our empirical studies clearly show that our technique can help developers obtain results for preferred configurations early in the overall testing process, compared to a cost-based strategy. The results from the most important configurations may be produced almost at the end of the plan execution without the prioritization technique. In addition, while exploring the performance varying the number of employed client machines and cache entries per machine, I also observed that the preference-guided plan execution can achieve more benefits with larger cache sizes.

Developers can control the strength of preference enforcement by specifying a parameter, the *window size*. The prioritized strategy uses the parameter value for increasing prefix reuse locality and for decreasing work redundancy between multiple clients. The simulation results showed that the prioritized plan execution can perform comparably to the cost-based strategy with larger window sizes

and can produce results for more important configurations rapidly with smaller window sizes.

Chapter 7

Conclusions and Future Work

In this chapter, I conclude this dissertation by reviewing the thesis and its contributions and present several directions for future work.

7.1 Thesis and Contributions

In this dissertation, I supported the following thesis: *Direct-dependency-based configuration sampling techniques can be effectively employed for testing build-compatibility of component-based systems*. The goal of this research was to develop effective and efficient methods and tools for performing compatibility testing of complex and evolving component-based systems. The contributions made by this dissertation include:

An effective and efficient method for sampling and testing configurations

I have developed and presented in this dissertation the first approach for testing the compatibility of component-based systems by systematically sampling and testing configurations. That approach consisted of a well-defined test process, a formal model for capturing the configuration space of component-based systems, algorithms for sampling and testing configurations, and finally a tool that realizes the algorithms.

Based on the observation that a successful component build is mostly influenced by other components on which the component directly depends, the configuration sampling algorithm can produce a set of configurations that effectively identify compatibilities between directly-dependent components (DD-instances) and those configurations can be tested efficiently on multiple machines in parallel. Compared to testing all possible configurations, which is infeasible in many cases, results from experiments and simulations on two scientific software systems showed that the presented approach can quickly identify compatibility results (successes/failures of DD-instances), and for the systems evaluated in this dissertation, the results were identical to the results from the exhaustive approach.

A set of techniques to support incremental compatibility testing

I have developed a set of techniques to support incremental compatibility testing as components in a system model evolve over time. The method consists of a test adequacy criterion that defines DD-instances that should be tested for a modified model and an algorithm for producing configurations that satisfy the criterion. The method is incremental in that DD-instances tested in a test session are not tested again in subsequent sessions unless they are contained in configurations for testing other DD-instances. In addition, two cache-aware optimization techniques were developed to further decrease test effort by utilizing past test results. Simulations over the 5-year evolution history of components in the models for two large-scale systems showed that the testing time can decrease by a large amount by applying incremental testing and also the optimization techniques can significantly reduce the time required for the test.

A method for prioritizing configurations via developers' preferences

It is important to provide developers with the compatibility results they have the most interest early in the test process when test resources are limited. To achieve that goal, I have developed a method that consists of a simple way of specifying developers' preferences on components and their versions, an algorithm for com-

puting the priorities of sampled configurations based on the preferences and finally a scheduling policy that guides the test order of configurations by considering both the priorities of configurations and the cost required for testing the configurations. Developers are allowed to control how rigidly to enforce the preferences over time savings.

The results from empirical studies demonstrated that the presented method can enable developers to acquire more important compatibility results early in the testing process, and also showed that the scheduling policy can perform comparably to the pure cost-based strategy when a large cache space is available or developers weaken the rigidity of preference enforcement.

7.2 Future Work

There can be many possible extensions and improvements to the work presented in this dissertation. Although a set of algorithms and techniques have been developed for testing compatibilities between components effectively and efficiently, this work may still be improved in several directions.

Automatic extraction of constraints and dependencies

The CDGs and Annotations used for experiments and simulations in this dissertation were created manually by carefully inspecting the information acquired by working with system developers or from available documents. However, manual modeling can be error-prone and faulty models can produce misleading compatibility results. Therefore, it is necessary to investigate methods that can extract component dependencies and constraints automatically from component distributions. In fact, there is no standard way of building components, but there are common practices used by many developers. Although component developers can use any method in which they check dependency requirements and build their components, it may be possible to extract dependencies and constraints systematically for components packaged with well-established component distribution methods, such as RPM [11, 37], Autotools [31, 69] and Ant [44].

Exploring other coverage criteria

Through extensive experiments and simulations with real-world systems, this dissertation has demonstrated that the cost required to perform compatibility tests can be reduced by a large amount and compatibilities between components can be discovered effectively by testing a reduced set of configurations that satisfy a test

adequacy criterion, namely DD-coverage. Despite the observed benefits, it is still necessary to explore new types of coverage criteria that can produce fewer but perhaps a more effective set of configurations. Especially if a component directly depends on multiple other components, each with many versions, the number of DD-instances for the component increases by a large amount and it is very expensive to build all configurations for testing the DD-instances.

This problem may be addressed by employing combinatorial interaction testing techniques developed for generating test cases that cover interactions between test factors of a system [19, 23, 24, 25, 27, 28, 52, 53, 67]. To use the techniques, it is necessary to investigate how to compute test obligations and how to enforce constraints for the computation.

Extending to functional and performance testing

This research has focused on testing clean component builds as the first step to support compatibility testing of component-based systems. In the build process, many component build tools check whether basic features required by a component are provided by other existing components in a configuration. However, component developers often provide test cases that test the correct behavior of their components after deployment, since components cleanly built on a configuration

can show incorrect behavior at run-time.

This means that the functional and performance test should be performed in addition to build testing. It is possible that testing direct dependencies is not enough to ensure the correct behavior of a component in a configuration. Moreover, it is also possible that a configuration for running a performance test suite cannot be realized as a virtual machine. Therefore, it would be needed to investigate a new coverage criterion and also methods for provisioning configurations on physical machines, when configurations cannot be realized as virtual machines.

Adapting the Cloud computing paradigm

As observed from the experimental and simulation results in this research, testing the compatibility of a component-based system requires a large amount of storage and computing power – even for building components without performing other types of tests. Moreover, developers often cannot test compatibilities with commercial components because they cannot afford those components, or because source code for the components cannot be obtained.

Although the Rachet tool has been developed with a classic client-server architecture and all experiments have been run on a cluster of machines, the tool could also be implemented to be run as a service in a Cloud computing environment.

Considering the recent growth of Cloud computing [10, 40], such an extension may open up the possibility for developers of independent systems to save test effort when configurations for testing the compatibility of the systems are shared.

BIBLIOGRAPHY

[1] Accelerate software development, testing and deployment with the VMware virtualization platform - whitepaper, VMware Inc.

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

[3] http://mercurial.selenic.com.

[4] http://subversion.tigris.org.

[5] http://www.nongnu.org/cvs.

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

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

[8] Virtual lab automation (a quantum leap in it cost reduction and application development process improvement) - whitepaper. VMware, 2006.

[9] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. D. Croz, S. Hammarling, J. Demmel, C. H. Bischof, and D. C. Sorensen. LAPACK: A portable linear

algebra library for high-performance computers. In *Proceedings Supercomputing '90*, pages 2–11, Nov. 1990.

[10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Feb. 2009.

[11] E. C. Bailey. *Maximum RPM*. Sams, first edition, 1997.

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

[13] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.

[14] K. Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.

[15] B. Boehm. Some future trends and implications for systems and software engineering processes. *Systems Engineering*, 9(1):1–19, Jan. 2006.

[16] B. Boehm and C. Abts. COTS integration: Plug and pray? *IEEE Software*, 32(1):135–138, Jan. 1999.

[17] J. Bosch and P. Bosch-Sijtsema. From integration to composition: On the impact of software product lines, global development and ecosystems. *Journal of Systems and Software*, 83(1):67–76, Jan. 2010.

[18] R. C. Bryce and C. J. Colbourn. Test prioritization for pairwise interaction coverage. *ACM Software Engineering Notes*, 30(4):1–7, Jul. 2005.

[19] R. C. Bryce, C. J. Colbourn, and M. B. Cohen. A framework of greedy methods for constructing interaction test suites. In *Proceedings of the 27th IEEE/ACM International Conference on Software Engineering (ICSE05)*, pages 146–155, May 2005.

[20] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[21] W. Cleveland. Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 83:829—836, 1979.

[22] W. Cleveland and S. Devlin. Locally weighted regression: An approach to regression analysis by local fitting. *Journal of the American Statistical Association*, 83:596–610, 1988.

[23] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatiorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, Jul. 1997.

[24] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–88, Sep. 1996.

[25] M. B. Cohen. *Designing Test Suites for Software Interaction Testing*. PhD thesis, University of Auckland, New Zealand, Sep. 2004.

[26] M. B. Cohen, M. B. Dwyer, and J. Shi. Coverage and adequacy in software product line testing. In *Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis (ROSATEA06)*, pages 53–63, Jul. 2006.

[27] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th IEEE/ACM International Conference on Software Engineering (ICSE03)*, pages 38–48, May 2003.

[28] C. J. Colbourn, M. B. Cohen, and R. C. Turban. A deterministic density algorithm for pairwise interaction coverage. In *Proceedings of the IASTED International Conference on Software Engineering (IASTED04)*, pages 242–252, Feb. 2004.

[29] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, Jan./Mar. 2005.

[30] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *Proceedings of the 11th International Software Product Line Conference (SPLC 2007)*, pages 23–34, Sep. 2007.

[31] M. B. Doar. *Practical Development Environments*. O'Reilly Media, first edition, 2005.

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

[33] A. Duarte, G. Wagner, F. Brasileiro, and W. Cirne. Multi-environment software testing on the Grid. In *Proceeding of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD06)*, Jul. 2006.

[34] A. N. Duarte, W. Cirne, F. Brasileiro, and P. Machado. GridUnit: Software testing on the Grid. In *Proceedings of the 28th IEEE/ACM International Conference on Software Engineering (ICSE06)*, May 2006.

[35] 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.

[36] T. Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall, 2005.

[37] E. Foster-Johnson. *Red Hat RPM Guide*. Red Hat, first edition, 2003.

[38] M. Fowler. http://martinfowler.com/articles/continuousintegration.html, May 2006.

[39] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, Nov. 1995.

[40] R. Giordanelli and C. Mastroianni. The cloud computing paradigm: Characteristics, opportunities and research issues. Technical Report RT-ICAR-CS-10-01, Institute of High Performance Computing and Networking, Apr. 2010.

[41] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *Proceedings of the 20th IEEE/ACM International Conference on Software Engineering (ICSE98)*, pages 188–197, Apr. 1998.

[42] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. 10(2):184–208, Apr. 2001.

[43] W. D. Gropp and B. F. Smith. Scalable, extensible, and portable numerical libraries. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*, pages 87–93, 1994.

[44] S. Holzner, D. Nehren, and B. Galbraith. *Ant: The Definitive Guide*. O'Reilly & Associates, Inc, first edition, 2005.

[45] M. N. Huhns and M. P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, Jan./Feb. 2005.

[46] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th IEEE/ACM International Conference on Software Engineering (ICSE02)*, pages 119–129, May 2002.

[47] J.-Y. Lee and A. Sussman. Efficient communication between parallel programs with InterComm. Technical Report CS-TR-4557 and UMIACS-TR-2004-04, University of Maryland, Department of Computer Science and UMIACS, 2004.

[48] 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 (HIPS-HPGC 2005)*, Apr. 2005.

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

[50] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE06)*, pages 199–208, Sep. 2006.

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

[52] K. Meagher, L. Moura, and L. Zekaoui. Mixed covering arrays on graphs - to appear. *Journal of Combinatorial Designs*.

[53] K. Meagher and B. Stevens. Covering arrays on graphs. *Journal of Combinatorial Theory Series B*, 95(1):134–151, Sep. 2005.

[54] 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.

[55] A. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Transactions on Software Engineering and Methodology*, 18(2), Nov. 2008.

[56] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *Proceedings of the 26th IEEE/ACM International Conference on Software Engineering (ICSE04)*, pages 459–468, May 2004.

[57] A. Porter. Towards a distributed continuous certification process. In *Proceedings of the 21th International Parallel & Distributed Processing Symposium (IPDPS07)*, 2007.

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

[59] B. Robinson and L. White. Testing of user-configurable software systems using firewalls. In *Proceedings of the 19th International Symposium on Software Reliability Engineering (ISSRE08)*, pages 177–186. ABB, Case Western Reserve Univ., Nov. 2008.

[60] B. Rumpe and A. Schröder. Quantitative survey on Extreme Programming projects. In *Proceedings of the 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, pages 95–100, May 2002.

[61] R. Sabharwal. Grid infrastructure deployment using SmartFrog technology. In *Proceedings of the 2006 International Conference on Networking and Services*

(INCS06), Jul. 2006.

[62] H. Srikanth and L. Williams. On the economics of requirements-based test case prioritization. *ACM Software Engineering Notes*, 30(4):1–3, Jul. 2005.

[63] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. In *Proceedings of 2005 International Symposium on Empirical Software Engineering*, Nov. 2005.

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

[65] A. Sussman and H. Andrade. Enabling coupled scientific simulations on the Grid. In J. Dongarra, K. Madsen, and J. Waśniewski, editors, *Proceedings of the PARA 2004 Workshop on State-of-the-Art in Scientific Computing*, volume LNCS 3732 of *Lecture Notes in Computer Science*, pages 217–224. Springer-Verlag, 2006.

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

[67] K.-C. Tai and Y. Lei. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28(1):109–111, Jan. 2002.

[68] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. OPIUM: Optimal package install/uninstall manager. In *Proceedings of the 29th IEEE/ACM International Conference on Software Engineering (ICSE07)*, pages 178–188, May 2007.

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

[70] K. Wallnau, S. Hissam, and R. Seacord. *Building Systems from Commercial Components*. Addison Wesley Professional, 2001.

[71] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the 8th International Symposium on Software Reliability Engineering (ISSRE97)*, pages 230–238. bell communications, Nov. 1997.

[72] C. Yilmaz, M. B. Cohen, and A. A. Porter. Covering arrays for efficient fault characterization in complex configuration space. *IEEE Transactions on Software Engineering*, 32(1):20–34, Jan. 2006.

[73] C. Yilmaz, A. Porter, and D. C. Schmidt. Distributed continuous quality assurance: The skoll project. In *Proceedings of the 1st International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS03)*, May 2003.

[74] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Direct-dependency-based software compatibility testing. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE07)*, Nov. 2007.

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

[76] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Prioritizing component compatibility tests via user preferences. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM09)*, Sep. 2009.