MONOGRAPHS IN COMPUTER SCIENCE

SOFTWARE CONFIGURATION MANAGEMENT USING VESTA

Allan Heydon
Roy Levin
Timothy Mann
Yuan Yu

Springer

Monographs in Computer Science

Editors

David Gries
Fred B. Schneider

Monographs in Computer Science

Abadi and Cardelli, **A Theory of Objects**

Benosman and Kang [editors], **Panoramic Vision: Sensors, Theory, and Applications**

Bhanu, Lin, Krawiec, **Evolutionary Synthesis of Pattern Recognition Systems**

Broy and Stølen, **Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement**

Brzozowski and Seger, **Asynchronous Circuits**

Burgin, **Super-Recursive Algorithms**

Cantone, Omodeo, and Policriti, **Set Theory for Computing: From Decision Procedures to Declarative Programming with Sets**

Castillo, Gutiérrez, and Hadi, **Expert Systems and Probabilistic Network Models**

Downey and Fellows, **Parameterized Complexity**

Feijen and van Gasteren, **On a Method of Multiprogramming**

Herbert and Spärck Jones [editors], **Computer Systems: Theory, Technology, and Applications**

Heydon, Levin, Mann, and Yu, **Software Configuration Management Using Vesta**

Leiss, **Language Equations**

McIver and Morgan [editors], **Programming Methodology**

McIver and Morgan [editors], **Abstraction, Refinement and Proof for Probabilistic Systems**

Misra, **A Discipline of Multiprogramming: Programming Theory for Distributed Applications**

Nielson [editor], **ML with Concurrency**

Paton [editor], **Active Rules in Database Systems**

Poernomo, Crossley, Wirsing, **Adapting Proofs-as-Programs: The Curry-Howard Protocol**

Selig, **Geometrical Methods in Robotics**

Selig, **Geometric Fundamentals of Robotics, Second Edition**

Shasha and Zhu, **High Performance Discovery in Time Series: Techniques and Case Studies**

Tonella and Potrich, **Reverse Engineering of Object Oriented Code**

Allan Heydon
Roy Levin
Timothy Mann
Yuan Yu

Software Configuration Management Using Vesta

Springer

Allan Heydon
Guidewire Software
2121 S. El Camino Real
San Mateo, CA 94403
U.S.A.

Roy Levin
Microsoft Research–Silicon Valley Center
1065 La Avenida
Mountain View, CA 94043
U.S.A.

Timothy Mann
VMware, Inc.
3145 Porter Dr.
Palo Alto, CA 94304
U.S.A.

Yuan Yu
Microsoft Research–Silicon Valley Center
1065 La Avenida
Mountain View, CA 94043
U.S.A.

Series Editors:
David Gries
Cornell University
Department of Computer Science
Ithaca, NY 14853
U.S.A.

Fred B. Schneider
Cornell University
Department of Computer Science
Ithaca, NY 14853
U.S.A.

Library of Congress Control Number: 2005936522

ISBN-10: 0-387-00229-4 e-ISBN: 0-387-30852-0
ISBN-13: 978-0-387-00229-3

Printed on acid-free paper.

Printed in the United States of America. (MP)

9 8 7 6 5 4 3 2 1

springeronline.com

To our former colleagues at the
DEC/Compaq Systems Research Center

Preface

The core technologies underlying software configuration management have changed little in more than two decades. Development organizations struggle to manage ever-larger software systems with tools that were never designed to handle them. Their development processes are warped by the inadequacies of their building and version management tools. Developers must take time from writing and debugging code to cope with the operational problems thrust upon them by their build system's inadequate support of large-scale concurrent development.

Vesta, a novel system for large-scale software configuration management, offers a better solution. Through a unique integration of building and version management facilities, Vesta constructs software of any size repeatably, incrementally, and consistently. Since modern software development occurs worldwide, Vesta supports concurrent, multi-site, distributed development. Vesta's core facilities are methodologically neutral, allowing development organizations a wide range of flexibility in the way they arrange their code repositories and structure the building of system components. In short, Vesta advances the state of the art in configuration management.

The idea behind Vesta is simple. Conceptually, every system build, no matter how extensive, occurs from scratch. That means that Vesta has a complete description of the source files from which the system is constructed, plus a complete and precise procedure for putting them together. By making these files and procedures immutable and immortal, Vesta ensures that a build can always be repeated. By extensively caching the results of builds, Vesta converts a conceptual scratch build into an incremental one, reusing previously built components when appropriate. By automatically detecting the dependencies between the system's parts, Vesta guarantees that incremental builds are consistent. What makes Vesta interesting and useful is its ability to do all this for software systems comprising millions of lines of code while being practical and even pleasant for developers and their management.

This book presents a comprehensive explanation of Vesta's architecture and individual components, showing how its novel and ambitious properties are achieved. Vesta's functionality is compared with that of standard development tools, highlighting how Vesta overcomes their specific deficiencies while matching or even exceeding their performance. Detailed examples demonstrate Vesta's facilities as they

appear to a developer, and a particular methodology of proven utility for large system development shows how Vesta works on an organization-wide scale. For the reader who wants to see Vesta "with the covers off", the book includes a substantial treatment of the subtle and challenging aspects of the implementation, as well as references to the open-source code.

Audience and Scope

The audience for this book includes anyone who has ever struggled with the problems of managing a substantial evolving software code base and wondered, "Isn't there a better way to do this?" While the book is not a "how-to" manual, it does demonstrate specific tools and techniques, founded on Vesta's core version management and building technologies, that are eminently practical. The Vesta system embodies and encourages principled development, and so will interest software engineering researchers, especially those inclined toward the creation of practical tools. Readers with a need to design and deploy configuration management solutions will find Vesta's flexible description language and build system a powerful, original approach to the persistent problem of coping with complex dependencies among software components.

The Vesta system builds on many computer science specialties, including programming language design and implementation, garbage collection, file systems, concurrent programming, and fault-tolerance techniques. Some familiarity with these topics is assumed.

Acknowledgements

The Vesta system was many years in the making. The core idea behind Vesta first grabbed the attention of one of the authors of this book (RL) around 1979. The problems Vesta addresses — version management and system building — are as central to software development today as they were then, but in the past couple of decades the standard tools in this area haven't progressed much. Why not? We believe it is for the same reason that we still use the QWERTY keyboard: early *de facto* standardization on ultimately limiting technology. There are better system-building tools (and better keyboards), but they are non-standard. Standard system-building tools have brought software developers to a local hilltop. Vesta, we argue in this book, offers a view from a different, higher one.

The path to that hilltop hasn't been straight. The development of a practical system embodying our core idea — the notion of an exhaustive, machine-interpretable description of the construction of a software system from source code — proved surprisingly difficult. The first steps occurred in the context of the Cedar experimental programming environment [35, 36], A full-scale project to explore the subject didn't get underway for several years, as part of the Taos system at the DEC Systems Research Center (SRC). This project, called Vesta but later renamed Vesta-1,

produced a usable but idiosyncratic system capable of repeatable, incremental, consistent builds of large-scale software. It saw significant use at SRC (but nowhere else) in the early 1990s [11, 13, 25, 40]. Vesta-2, the subject of this book, came along several years later after considerable analysis of the use of Vesta-1, followed by a complete redesign and reimplementation.

Of course, no system just "comes along". The Vesta systems owe their existence to the hard work of many colleagues who generously gave their ideas, opinions, insights, code, encouragement, bug reports, and comradeship. With so many participants over so many years, it is impossible to thank them all, but we want to acknowledge a number of key contributors.

The initial inspiration for Vesta came from Butler Lampson and his work with Eric Schmidt and Ed Satterthwaite on Cedar and its predecessor systems at Xerox PARC. Butler guided our thinking on numerous occasions throughout the Vesta-1 and Vesta-2 projects, contributing to the designs for the system modeling languages and repositories. He also played a major role in designing the Vesta-2 function cache and weeder described in chapters 8 and 9.

The Vesta-1 system was developed by Bob Ayers, Mark R. Brown, Sheng-Yang Chiu, John Ellis, Chris Hanna, Roy Levin, and Paul McJones, several of whom also assisted in the analysis of Vesta-1's use that informed the design of Vesta-2.

Jim Horning and Martín Abadi, with Butler's participation, helped design the Vesta-2 evaluator's fine-grained dependency algorithm. Together with Chris Hanna, Jim also contributed to the design of the system description language and the initial implementation of the evaluator.

Bill McKeeman's incisive and insistent suggestions led us to make the description language syntax simpler and more readable. Our fingerprint package on which Vesta's repository and cache depend heavily descends directly from ideas and code of Andrei Broder. Jeff Mogul and Mike Burrows helped track down a serious performance problem in our RPC implementation. Chandu Thekkath helped with NFS performance problems and gave helpful comments on an early draft of this book. Emin Gün Sirer implemented the Modula-3 bridge and made several improvements to the performance of the entire system. Mark Lillibridge gave us many useful comments on an earlier draft of Appendix A. Cynthia Hibbard and Jim Horning provided numerous suggestions for improvement on various drafts of the manuscript. Neil Stratford coded an early version of the replication tools and some of the repository support for them.

Tim Leonard initiated our contact with the Araña (Alpha microprocessor) development group, which became Vesta's first real user community outside SRC, and Walker Anderson and Joford Lim led that group's initial evaluation of Vesta. Matt Reilly and Ken Schalk championed the use of Vesta in the Araña group, seeing it through to eventual adoption and production use. Both were involved in the port of Vesta to Linux, and Ken has become the driving force in evolving the present open-source Vesta system. It is through his tireless efforts that developers unconnected with the original work at DEC have an opportunity to evaluate Vesta as a practical alternative to conventional configuration management tools. Scott Venier created Vestaweb, a very useful web interface for exploring a Vesta repository.

Finally, we owe a debt of gratitude to Bob Taylor, whose regular encouragement kept us from abandoning Vesta when it seemed unlikely it would ever see use outside the research lab. Without Bob's unflagging support over many years and two companies, Vesta would probably never have happened.

This book, like the Vesta system itself, has been many years in the making. It began as a Compaq technical report [27], and we thank Hewlett-Packard for permission to use portions of that report. We also are indebted to John DeTreville for the Vesta logo that appears on the cover. But the book would not exist without the support of two key individuals. Fred Schneider, as series co-editor for Springer's *Monographs in Computer Science*, persuaded us to undertake the production of this book when the complexities of our day jobs made it seem impossible. Our editor at Springer, Wayne Wheeler, showed remarkable patience in the face of repeated underestimates of the work involved. We are grateful to Fred and Wayne and the staff at Springer (notably Frank Ganz, Ann Kostant, and Elizabeth Loew) for their continuous support during the preparation of the book, and we hope that the result justifies their faith.

Palo Alto, California
December 2005

Allan Heydon
Roy Levin
Tim Mann
Yuan Yu

Contents

Preface vii

Part I Introducing Vesta

1 Introduction 5
1.1 Some Scenarios 6
1.2 The Configuration Management Challenge 8
1.3 The Vesta Response 9

2 Essential Background 13
2.1 The Unix File System 14
2.1.1 Naming Files and Directories 14
2.1.2 Mount Points 14
2.1.3 Links 15
2.1.4 Properties of Files 15
2.2 Unix Processes 16
2.3 The Unix Shell 17
2.4 The Unix Programming Environment 18
2.5 Make 20

3 The Architecture of Vesta 21
3.1 System Components 21
3.1.1 Source Management Components 22
3.1.2 Build Components 24
3.1.3 Storage Components 27
3.1.4 Models and Modularity 28
3.2 Vesta's Core Properties 29

Part II The User's View of Vesta

4 Managing Sources and Versions 35
4.1 Names and Versions 36
4.1.1 The Source Name Space 36
4.1.2 Versioning 37
4.1.3 Naming Files and Packages 38
4.2 The Development Cycle 40
4.2.1 The Outer Loop 40
4.2.2 The Inner Loop 41
4.2.3 Detailed Operation of the Repository Tools 42
4.2.4 Version Control Alternatives 44
4.2.5 Additional Repository Tools 45
4.2.6 Mutable Files and Directories 45
4.3 Replication 46
4.3.1 Global Name Space 46
4.3.2 A Replication Example 48
4.3.3 The Replicator 49
4.3.4 Cross-Repository Check-out 50
4.4 Repository Metadata 52
4.4.1 Mutable Attributes 52
4.4.2 Access Control 55
4.4.3 Metadata and Replication 57

5 System Description Language 59
5.1 Motivation 59
5.2 Language Highlights 60
5.2.1 The Environment Parameter 62
5.2.2 Bindings 63
5.2.3 Tool Encapsulation 65
5.2.4 Closures 67
5.2.5 Imports 68

6 Building Systems in Vesta 71
6.1 The Organization of System Models 72
6.2 Hierarchies of System Models 74
6.2.1 Bridges and the Standard Environment 76
6.2.2 Library Models 77
6.2.3 Application Models 79
6.2.4 Putting It All Together 80
6.2.5 Control Panel Models 81
6.3 Customizing the Build Process 84
6.4 Handling Large Scale Software 88

Part III Inside Vesta

7 Inside the Repository 93
7.1 Support for Evaluation and Caching 93
7.1.1 Derived Files and Shortids 93
7.1.2 Evaluator Directories and Volatile Directories 94
7.1.3 Fingerprints 96
7.2 Inside the Repository Implementation 98
7.2.1 Directory Implementation 98
7.2.2 Shortids and Files 100
7.2.3 Longids 101
7.2.4 Copy-on-Write 103
7.2.5 NFS Interface 104
7.2.6 RPC Interfaces 105
7.3 Implementing Replication 105
7.3.1 Mastership 105
7.3.2 Agreement 106
7.3.3 Agreement-Preserving Primitives 108
7.3.4 Propagating Attributes 110

8 Incremental Building 113
8.1 Overview of Function Caching 113
8.2 Caching and Dynamic Dependencies 115
8.3 The Function Cache Interface 119
8.4 Computing Fine-Grained Dependencies 120
8.4.1 Representing Dependencies 120
8.4.2 Caching External Tool Invocations 121
8.4.3 Caching User-Defined Function Evaluations 123
8.4.4 Caching System Model Evaluations: A Special Case 131
8.5 Error Handling 132
8.6 Function Cache Implementation 134
8.6.1 Cache Lookup 135
8.6.2 Cache Entry Storage 138
8.6.3 Synchronization 139
8.7 Evaluation and Caching in Action 139
8.7.1 Scratch Build of the Standard Environment 139
8.7.2 Scratch Build of the Vesta Umbrella Library 142
8.7.3 Scratch and Incremental Builds of the Evaluator 144

9 Weeder 147
9.1 How Deletion is Specified 148
9.2 Implementation of the Weeder 149
9.2.1 The Function Call Graph 149
9.2.2 Concurrent Weeding 152

Part IV Assessing Vesta

10 Competing Systems 161
10.1 Loosely Connected Configuration Management Tools 161
10.1.1 RCS 162
10.1.2 CVS 162
10.1.3 Make 163
10.2 Integrated Configuration Management Systems 165
10.2.1 DSEE 165
10.2.2 ClearCASE 167
10.3 Other Systems 168

11 Vesta System Performance 171
11.1 Platform Configuration 172
11.2 Overall System Performance 172
11.2.1 Performance Comparison with Make 173
11.2.2 Performance Breakdown 175
11.2.3 Caching Analysis 177
11.2.4 Resource Usage 178
11.3 Repository Performance 180
11.3.1 Speed of File Operations 181
11.3.2 Disk and Memory Consumption 183
11.3.3 Speed of Repository Tools 186
11.3.4 Speed of Cross-Repository Tools 188
11.3.5 Speed of the Replicator 189
11.4 Function Cache Performance 190
11.4.1 Server Performance 190
11.4.2 Measurements of the Stable Cache 191
11.4.3 Disk and Memory Usage 192
11.4.4 Function Cache Scalability 192
11.5 Weeder Performance 193
11.6 Interprocess Communication 194

12 Conclusions 197
12.1 Vesta in the Real World 198
12.2 Vesta in the Future 199

A SDL Reference Manual 203
A.1 Introduction 203
A.2 Lexical Conventions 204
A.2.1 Meta-notation 204
A.2.2 Terminals 204
A.3 Semantics 205
A.3.1 Value Space 205

A.3.2 Type Declarations 206
A.3.3 Evaluation Rules 207
A.3.3.1 Expr 208
A.3.3.2 Literal 209
A.3.3.3 Id 209
A.3.3.4 List 209
A.3.3.5 Binding 212
A.3.3.6 Select 213
A.3.3.7 Block 214
A.3.3.8 Stmt 215
A.3.3.9 Assign 215
A.3.3.10 Iterate 216
A.3.3.11 FuncDef 216
A.3.3.12 FuncCall 219
A.3.3.13 Model 220
A.3.3.14 Files 220
A.3.3.15 Imports 224
A.3.3.16 File Name Interpretation 227
A.3.3.17 Pragmas 228
A.3.4 Primitives 228
A.3.4.1 Functions on Type t_bool 229
A.3.4.2 Functions on Type t_int 229
A.3.4.3 Functions on Type t_text 230
A.3.4.4 Functions on Type t_list 232
A.3.4.5 Functions on Type t_binding 234
A.3.4.6 Special Purpose Functions 237
A.3.4.7 Type Manipulation Functions 238
A.3.4.8 Tool Invocation Function 239
A.3.4.9 Diagnostic Functions 243
A.4 Concrete Syntax 244
A.4.1 Grammar 244
A.4.2 Ambiguity Resolution 247
A.4.3 Tokens 247
A.4.4 Reserved Identifiers 249

B The Vesta Web Site 251

References 253

Index 257

Software Configuration Management Using Vesta

Part I

Introducing Vesta

The first part of this book sets the stage for an in-depth presentation of the Vesta system. Chapter 1 presents the key problems that Vesta addresses and lays out the essential properties of Vesta's solution. Chapter 2 provides some technical background on Unix, the operating system on which Vesta is implemented, chiefly targeted at the non-specialist. Chapter 3 then surveys the architecture of the Vesta system, presenting its major components and their interactions, and laying the foundation for a more detailed survey of Vesta's functionality in Part II.

1

Introduction

This book describes Vesta [26,28,43], a system for software versioning and building that *scales to accommodate large projects*, is *easy to use*, and guarantees *repeatable, incremental, and consistent builds*. Vesta embodies the belief that reliable, incremental, consistent building is overwhelmingly important for software construction and that its absence from conventional development environments has significantly interfered with the production of large systems. Consequently, Vesta focuses on the two central challenges of large-scale software development — versioning and building — and offers a novel, integrated solution.

Versioning is an inevitable problem for large-scale software systems because software evolves and changes substantially over time. Major differences often exist between the source code in various shipped versions of a software product, as well as between the latest shipped version and the current sources under development, yet bugs have to be fixed in all these versions. Also, although many developers may work on the current sources at the same time, each needs the ability to test individual changes in isolation from changes made by others. Thus a powerful versioning system is essential so that developers can create, name, track, and control many versions of the sources.

Building is also a major problem. Without some form of automated support, the task of compiling or otherwise processing source files and combining them into a finished system is time-consuming, error-prone, and likely to produce inconsistent results. As a software system grows, this task becomes increasingly difficult to manage, and comprehensive automation becomes essential. Every organization with a multi-million line code base wants an automated build system that is reliable, efficient, easy-to-use, and general enough for their application. These organizations are very often dissatisfied with the build systems available to them and are forced to distort their development processes to cope with the limitations of their software-building machinery.

Versioning and building are two parts of a larger problem area that is often called *software configuration management* (SCM). The broadest definition of SCM encompasses such topics as software life-cycle management (spanning everything from requirements gathering to bug tracking), development process methodology, and the

specific tools used to develop and evolve software components. Vesta takes the view that these aspects of SCM, although important to the overall software development process, can be sensibly addressed only after the central issues of versioning and building. Further, in contrast to most conventional SCM systems, Vesta takes the view that these two problems interact, and that a proper solution integrates them so that the versioning and building facilities leverage each other's properties. That integrated solution then serves as a solid base upon which to construct facilities that address other SCM problems.

1.1 Some Scenarios

To motivate Vesta's focus on versioning, building, and their integration, here are some scenarios that conventional software development environments do not always handle well.

Scenario 1. A developer must check out a library to make a change necessary for his currently assigned task, but he can't because someone else has it checked out.

The problem: the source control system doesn't allow parallel development.

Scenario 2. Dave is having difficulty debugging a change because a library used by his code is behaving in an unexpected way. The library is a large and complex one but was built without including information required by the debugger. Dave knows nothing about the procedure for rebuilding the library to include the debugging information he needs.

The problem: the build system does not support the parameterization necessary for the developer to be able to say easily "rebuild this library including debugging information" and as a result, he must delve into the library's build instructions to determine how to set the necessary switch and build it manually.

Scenario 3. Alice is ready to begin debugging a substantial new feature, but to do so she requires several other components to be rebuilt with a new definition for a data structure that they share. She is unable to do this herself without setting up an environment comparable to that used by her organization's nightly build.

The problem: the build system and process do not enable developers to build substantial subportions of the complete system in order to test and debug their changes with other affected components.

Scenario 4. Susan, a developer in California, learns that her colleague Anoop needs to build Susan's software component at the Indian development lab. She would like to help, but is uncertain about the ways in which her component depends on local conditions that may be different in his development environment. She also has no way to determine what additional files she needs to send to Anoop in order to ensure that her component will build properly in India.

The problem: the build system does not ensure that building instructions are complete and capture all dependencies.

Scenario 5. Fred types "make", and his program compiles and links without errors, but it exhibits mysterious bugs. After a long fruitless debugging session, Fred tries "make clean; make" to build the program from scratch. The program then works.

The problem: the build system trusts the developers to supply dependency information rather than computing that information itself, and Fred — or some developer who had previously worked on this program — left some out.

Scenario 6. A developer comes into work and performs a "sync" operation, which copies recently checked-in files to her workstation. This keeps her local file tree from falling too far behind the work her colleagues are doing. However, after building her code with the new files, she finds that it no longer works as it did yesterday. There's no easy way for her to find the problematic change or to roll back to where she was before the "sync".

The problem: the version management system provides only coarse-grained updating and supports versioning only in the central code pool, not on behalf of individual developers.

Scenario 7. A developer is implementing a new feature. In the course of the implementation, he decides that the approach is flawed, so he deletes what he has been doing and goes home. Overnight, he has an idea about how to salvage a significant portion of his previous work, but since he didn't check the code in before deleting it from his workstation, it's gone.

The problem: the version management system provides no support for versioning except in the shared source pool, so it can't help the developer in this situation.

Scenario 8. John needs to make a small change to a library, so he checks it out. He makes the change, but when he tries to compile, the compiler gets a mysterious fatal error. He reports the problem to his colleague Mary, who checked in the library the previous day. Mary tries the same build on her workstation and it works. After some head-scratching and discussion, they discover that John and Mary have different versions of the compiler. Investigating further, they find that John was supposed to download a new compiler several weeks before, but the email telling him to do so came when he was absorbed making a delicate change to his code, so he put the message aside and ultimately forgot about it.

The problem: the build system and build instructions do not reflect or capture dependencies on the versions of tools used during the build process.

Scenario 9. A customer reports an error in an old but still supported release of a product. The developers attempt to reproduce the problem, but they are unable to rebuild the old system from source. Investigation reveals that a third-party library used in the old release was not included in the build tree and that when an updated version of that library was installed for use in a later release of the product, it overwrote the old one.

The problem: the version management and build facilities are not integrated and do not require that build instructions constitute a complete description of the system, causing an essential component to be inadvertently discarded.

1.2 The Configuration Management Challenge

The common theme highlighted by the preceding scenarios is the failure of conventional software configuration management systems to address the realities of building and evolving large systems. Effective SCM becomes more difficult as the size of the software system grows, as the number of developers using the SCM system increases, as the number of geographically distributed development sites grows, and as more releases are produced. To handle large-scale, multi-developer, multi-site, multi-release software development, an SCM system must guarantee that builds are *repeatable*, *incremental*, and *consistent*. Existing SCM systems generally fail to provide at least one of these properties (see Chapter 10 for specifics).

Repeatability. When multiple versions are being developed in parallel, the ability to repeat a previous build exactly is invaluable. For example, if a customer reports a bug in an older version of a product, developers must be able to recreate the faulty program, debug it, and develop a modified version that fixes the bug (scenario 9).

Repeatability is an easy goal to state and to appreciate, but a difficult one to attain. Most build systems in use today do not guarantee repeatability because their build results are dependent on some aspect of the building environment that the system does not control. This produces the all-too-common situation in which one developer says to another, as in scenario 8: "It works on my machine, what's different about yours?"

Incrementality. For the practical development of large systems, the builder must be incremental, reusing the results of previous builds wherever possible. Without reliable incremental building, a development organization is forced to perform some (if not all) of its builds from scratch. The slow turnaround time for such scratch builds increases the time required for development and testing. Incremental building, on the other hand, allows many developers to efficiently edit, build, debug, and test different parts of the source base in parallel. (Contrast with scenario 3.) Even large integration builds that combine work from many developers can be accelerated by incremental building — any components that have already been built, whether in the last integration build or in isolation by individual developers, are candidates for reuse.

Good performance in the incremental builder itself is also important. As software systems grow, even incremental building can be too slow if the running time of the builder (exclusive of the compilers and other tools it invokes) depends on the total size of the system to be built rather than the size of the changes. This problem can easily arise. For example, a simple incremental builder might work by checking each individual compiler invocation in the build to see whether it must be redone. If these checks have significant cost, such a builder will scale poorly. Indeed, this is the norm in most SCM systems.

Consistency. The build process performs a sequence of actions on source files (files created by developers, also called *sources*) and derived files (files previously created by the build system, also called *deriveds*). A build is *consistent* if every derived file it incorporates is up to date relative to the files from which it was produced. The

obvious way to achieve consistency is to perform every build from scratch (that is, starting from sources), which of course sacrifices incrementality. Correspondingly, a partial system build introduces the potential for inconsistency because some derived file may be out of date with respect to a source file, to another derived file, or to some aspect of the build environment on which it depends. When this happens, the semantics of the source and derived files no longer correspond. Such a system generally exhibits unwanted behavior that is difficult to debug, as in scenario 5.

Achieving these three essential properties is thus the central challenge for an effective SCM system.

1.3 The Vesta Response

This book shows how the Vesta system successfully addresses the SCM challenge. Specifically, it explains and justifies the claim at the beginning of this chapter:

> Vesta is an SCM system that *scales to accommodate large software*, is *easy to use*, and guarantees *repeatable, incremental, and consistent builds.*

Vesta subdivides the general problem of versioning into *version management* and *source control.* Building breaks down into *system modeling* and *model evaluation.*

Version Management. Version management is the process of assigning names to evolving sequences of related source files and supporting retrieval of those files by name. Some SCM systems apply version management to derived files as well, in the sense that derived files receive versioned, human-sensible names just as sources do. By contrast, Vesta's version management assigns human-sensible names to sources only, while derived files receive machine-oriented names and are managed automatically.

Source Control. Source control is the process of controlling or regulating the production of new versions of source files. Operations commonly associated with source control include *check-out* and *check-in*, which respectively reserve a new version name (typically incorporating a number) and supply the file or files to be associated with a previously reserved version name. Source control may be coupled with concurrency control as well, so that checking out a particular version may limit the ability of other users to check out related ones. Vesta adopts a unique perspective on source control, quite different from that of conventional SCM systems, that enables it to avoid the kinds of problems evident in the scenarios of the preceding section.

System Modeling. A *system model* describes how to build all or part of a software system. It names the software components that are combined to produce larger components or entire systems, names the tools used to combine them, and specifies how the tools are applied to the components. *Configuration description, system description,* and *building instructions* are equivalent terms for system model.

Conventional build systems typically do not require and therefore rarely have comprehensive building instructions. Instead, they depend on the environment, which

might comprise files on the developer's workstation and/or well-known server directories, to supply the unspecified pieces. This partial specification prevents repeatable builds. The first vital step toward achieving repeatability is to store source files and build tools immutably and immortally, as Vesta does, so that they are available when needed. The second step is to ensure that building instructions are *complete*, recording precisely which versions of which source files went into a build, which versions of tools (such as the compiler) were used, which command-line switches were supplied to those tools, and all other relevant aspects of the building environment. Vesta's system models do precisely that.

Model Evaluation. A system model can be viewed either as a static description of a system's configuration, or as an executable program that describes how to build the system. *Model evaluation* means taking the second view: running a *builder* or *evaluator* (the terms are used synonymously) to construct a complete system by processing and combining a collection of software components according to a system model's instructions.

By following those instructions to the letter, the builder performs in effect a scratch build of the system. Completeness of the instructions makes the build repeatable, but for practicality it must also be incremental. Incrementality means skipping some build actions and using previously computed results instead, an optimization that risks inconsistency. To ensure that an incremental build is consistent, the Vesta builder records every dependency of every derived file on the environment in which it was built. This includes dependencies on source files, other derived files, the tools used in the build, environmental details, and the building instructions themselves. Then, if anything on which a derived file depends has changed, the builder detects it and performs the necessary rebuilding. If not, the builder can be incremental and skip an unnecessary rebuilding step. Recording dependencies for use in this way is obviously impractical unless automated, and worthless unless exhaustive. Vesta's coupling of automated dependency analysis and incremental building distinguishes it from conventional SCM systems.

As these brief descriptions indicate, the four central topic areas are not independent. For that reason, the remainder of the book does not address them in order, taking instead a top-down approach. Part I presents an overview of Vesta's architecture. Part II describes the Vesta system as a software developer sees it, emphasizing the user-level concepts rather than the implementation. This part examines Vesta's facilities for storing files and manipulating them in the course of the development cycle. It also introduces the language in which system models are written and shows how it is used to describe large systems effectively. By the end of Part II, the reader will understand why Vesta is easy to use and how it can scale to handle large software systems while guaranteeing repeatable, incremental, and consistent builds.

Part III examines the implementation of the functionality described in Part II. Achieving each of the key properties — repeatability, incrementality, consistency — requires the solution of significant technical problems. This part focuses on those problems and their solutions, providing sufficient description of the relevant parts of the implementation to evaluate Vesta's design and engineering choices.

Finally, Part IV compares Vesta against other leading SCM systems, both in function and performance. It shows that development organizations need not sacrifice the former for the latter; the key SCM properties are achieved with similar or even superior performance as compared to "industry-standard" builders.

2

Essential Background

The essential problems of software versioning and building transcend particular platforms and development environments. Nevertheless, concrete solutions to those problems are created for specific platforms and environments, and Vesta is no exception. The Vesta designers sought to address the central issues in a way that was minimally dependent on the environment, but inevitably there are dependencies of style, terminology, and implementation detail. This book presents Vesta in sufficient detail that these dependencies are visible, which therefore requires that the reader understand something of that dependent context.

To this end, this chapter presents a brief overview of the environment in and for which Vesta was originally built: Digital Equipment Corporation's Tru64® operating system.[1] Tru64 is a multi-generation descendent of the Berkeley (BSD) version of Unix. Vesta uses few notions that are peculiar to Unix, so the key Vesta concepts and most of the technical specifics transfer easily from Unix to other popular operating systems. Those specifics of Vesta are nevertheless shaped by the Unix context, so this chapter outlines that context as background for the material in the remainder of the book.

Readers who are conversant with Unix can quickly skim this chapter or skip it entirely. Those who are unfamiliar with Unix will likely find that the essentials described below have natural analogs in the environments with which they are familiar. This brief chapter is certainly not a reference on Unix concepts.[2] It occasionally sacrifices a bit of technical precision in the interest of remaining concise and conveying the key ideas necessary to understand Vesta, a fact that Unix and Tru64 aficionados will undoubtedly recognize.

[1] Vesta has been ported to a number of other Unix platforms, including Linux.

[2] The classic reference is Kernighan and Pike [33].

2.1 The Unix File System

2.1.1 Naming Files and Directories

File names are subdivided into a *name* and an *extension*, separated by a period (". "). This is only a convention; Unix has no machinery for associating semantics with file extensions, as is the case for some other operating systems (e.g., Microsoft Windows®). File extensions are very frequently used to identify the "type", that is, the internal format, of files. Because extensions are only conventional, they may be of any length, although between one and four characters is typical. For some kinds of files, the absence of an extension is the norm, but in such cases the usage of the file is such that a single fixed name (like `readme` or `Makefile`) is commonly used.

A *directory* is a collection of names, each of which may identify a file or another directory. These names do not distinguish the things they name; thus, the name `foo.bar` might be a directory or a file, although conventionally a name with an embedded dot is used for a file, not a directory.

The files and directories on a disk partition are arranged in a tree-structured name space. (This is a simplification, to be corrected shortly.) Within this tree, a *path* (sometimes called a *filename path*) is a sequence of names separated by the character "/". The root of the tree is named "/", so a path from the root might be `/x/y/z`. In such a path, every name, with the possible exception of the last, must be a directory, so in the path `/x/y/z`, `x` is a directory containing a directory named `y` containing `z` (`z` may name either a file or a directory). A path like `/x/y/z` is called *absolute* because it explicitly originates at the root. A path like `x/y/z` is called *relative*, meaning that it is to be interpreted relative to some directory that depends on the context in which the path is used.

Every directory contains the special name ". ", which refers to the directory itself. Every directory except the root also contains the special name ". .", which refers to the directory's parent in the naming tree.

2.1.2 Mount Points

The file name space that Unix programs and users see is created by connecting the directory trees on individual disk partitions via a mechanism called *mount points*. A directory tree T_1 is attached to a particular node N in tree T_2 by *mounting* it there, that is, by effectively splicing T_2 so that N becomes the name of the root of T_1. So, for example, if `a/b/c` names a file in T_1 and `x/y/z` is a path in T_2, mounting T_1 at `x/y` makes the file accessible as `x/y/a/b/c`. Note that, as a result of the mount, `x/y/z` is no longer in the name space.

The mount point mechanism enables the construction of large file name spaces out of the smaller ones that correspond to individual disk partitions. The individual disk partitions may be on separate computers; that is, a mount point may span file servers connected by a local area network. File servers may implement their file systems differently as long as they adhere to recognized protocols, of which NFS [49, 54] is a particularly common one. Vesta's storage machinery (Chapters 4 and 7) exploits this property.