January 22, 2002

SRC Research Report **177**

The Vesta Software Configuration Management System

Allan Heydon
Roy Levin
Timothy Mann
Yuan Yu

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.compaq.com/SRC/

Compaq Systems Research Center

SRC's charter is to advance the state of the art in computer systems by doing basic and applied research in support of our company's business objectives. Our interests and projects span scalable systems (including hardware, networking, distributed systems, and programming-language technology), the Internet (including the Web, e-commerce, and information retrieval), and human/computer interaction (including user-interface technology, computer-based appliances, and mobile computing). SRC was established in 1984 by Digital Equipment Corporation.

We test the value of our ideas by building hardware and software prototypes and assessing their utility in realistic settings. Interesting systems are too complex to be evaluated solely in the abstract; practical use enables us to investigate their properties in depth. This experience is useful in the short term in refining our designs and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this approach, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical character. Some of that lies in established fields of theoretical computer science, such as the analysis of algorithms, computer-aided geometric design, security and cryptography, and formal specification and verification. Other work explores new ground motivated by problems that arise in our systems research.

We are strongly committed to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences, while our technical note series allows timely dissemination of recent research findings. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

The Vesta Software Configuration Management System

Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu

January 22, 2002

Allan Heydon, Roy Levin, and Timothy Mann are no longer with Compaq. They can be reached by electronic mail at the addresses caheydon@yahoo.com, roy@levin.net, and tim@tim-mann.org.

The Vesta project's Web site is at http://www.vestasys.org/.

Abstract

Vesta is a system for software configuration management. It stores collections of source files, keeps track of which versions of which files go together, and automates the process of building a complete software artifact from its component pieces. Unlike other software configuration management systems, Vesta was specifically designed to handle very large projects—tens of millions of lines of code and beyond. Vesta's novel approach gives it three important properties not available in other systems. First, every build is *repeatable*, because its component sources and build tools are stored immutably and immortally, and its configuration description completely describes what components and tools are used and how they are put together. Second, every build is *incremental*, because results of previous builds are cached and reused. Third, every build is *consistent*, because all build dependencies are automatically captured, recorded, and checked, so that a cached result from a previous build is reused only when doing so is certain to be correct. In addition, Vesta's flexible language for writing configuration descriptions makes it easy to describe large software configurations in a modular fashion and to create variant configurations by customizing build parameters. This report describes the Vesta technology in detail and discusses the performance of our implementation.

Contents

Chapter 1

Introduction

This report describes Vesta [25, 26, 35], a system that addresses two core problems in developing large software projects: *versioning* and *building*.

Versioning is a significant problem for large-scale software systems because software evolves and changes over time. Serious disparities often exist between the source code used to produce the last shipped version of a software product and the current sources under development, yet bugs need to be fixed in both. Many developers may be working on the current sources at the same time, yet each needs to test his changes in isolation from changes made by others. Thus a powerful versioning system is essential: developers must be able to create, name, track, and control many versions of the sources.

Building is also a significant problem. Without some form of automated support, the task of compiling or otherwise processing source files and combining them into a finished system would be time-consuming, error-prone, and likely to produce inconsistent results. As a software system grows, this task becomes increasingly difficult. No existing automated build system is reliable, efficient, easy-to-use, and general enough to handle a multi-million line software project satisfactorily.

Versioning and building are two parts of a larger problem area that is often called *software configuration management* (SCM). Considered broadly, SCM is sometimes taken to include such areas as software life-cycle management, process management, and the specific tools used to develop and evolve software components. We take the view that these aspects of SCM, although important to the overall software development process, are secondary to the core issues of versioning and building. We have therefore focused the Vesta project on solving the core problems, constructing a solid base upon which we believe solutions to the other problems can be built.

In our approach, the general problem of versioning breaks down into two parts:

version management and *source control*. Building breaks down into *system modeling* and *model evaluation.* We now define these terms.

Version Management. Version management is the process of assigning names to a series of related source files and supporting retrieval of those files by name. Version management can also apply to derived files (files created mechanically by the build system). Vesta's version management system, however, deals only with sources; derived files (or *deriveds* for short) are managed entirely by the build system.

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 *checkout* and *checkin*, which respectively reserve a new version name (typically a number) and supply the data for a previously reserved version. Source control may be coupled with concurrency control as well, so that checking out a particular version limits the ability of other users to check out related versions.

System Modeling. A *system model* names the software components that are to be combined to produce larger components or entire systems, names the tools that are to be used to combine them, and specifies how the tools are to be applied. *Configuration description* is an equivalent term.

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* against a model, in order to construct a complete system by processing and combining its components according to the model's instructions.

The SCM problem becomes more difficult as the size of the software under development 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, we believe an SCM system must guarantee that builds are *repeatable*, *incremental*, and *consistent.* Existing SCM systems often fail to provide these properties.

Repeatability. In an environment where multiple versions are being developed in parallel, the ability to exactly repeat a previous build is invaluable. For example, if a customer reports a bug in an older version of a product, it is important to be able to quickly recreate the faulty executable, debug it, and develop a modified version that fixes the bug.

Repeatability is an easy goal to state and to appreciate, but a difficult goal 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. Hence, the all too common “it works on my machine” syndrome. The first vital step to achieving repeatability is to store source files and build tools immutably and immortally, so that they are available when needed. The second is to ensure that system models are *complete*, recording precisely which versions of which sources 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.

Incrementality. It is crucial for performance that the builder 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 of 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. 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 change. This problem can easily arise; for example, a simple incremental builder might work by checking each individual tool invocation in the build to see whether it must be redone. If these checks have significant cost, such a builder will scale poorly.

Consistency. A build is *consistent* if every derived file it incorporates is up to date relative to the sources from which it was produced. One way to achieve consistency is to perform every build from scratch. The potential for inconsistency arises, however, if builds are done incrementally. In particular, if some derived file used in a build is out of date with respect to a source file, to another derived file, or to any aspect of the build environment on which it depends, the build will be inconsistent and hence, unsound. An inconsistently built program may fail to link or may exhibit mysterious bugs that are not evident in the source code.

An incremental build can be made consistent by recording every dependency of every derived file on the environment in which it was built. This includes dependencies on source files, other derived files, environment variables, the tools used in the build, and the building instructions themselves. Then, if any element on which a derived file depends has changed, the need to rebuild it can be detected.

We now come to our central thesis: *Vesta is an SCM system that scales to* *large software, is easy to use, and produces repeatable, incremental, and consistent builds.* Although we can directly demonstrate the claims of repeatability, incrementality, and consistency, our arguments about scale and ease of use are indirect. To support the latter points, we describe the design decisions we made to ensure that the system will scale well and be easy to use, and we cite our experience with the Vesta-1 and Vesta-2 prototypes, both of which saw serious use. We also present performance measurements of our current system on small and medium-sized builds, and extrapolate from those experiments.

The rest of this report is organized as follows. First, we describe some widely used SCM systems and their shortcomings. Next, we give an overview of the Vesta system, stressing the technical features that guarantee repeatable, incremental, and consistent builds. We continue by describing the five major components of our implementation in detail: the repository, the system modeling language, the function cache, the evaluator, and the weeder. We then discuss the performance of our Vesta-2 prototype, comparing it to traditional SCM tools. We conclude by recapitulating the strengths and weaknesses of the Vesta approach. An appendix presents the detailed syntax and semantics of the Vesta system modeling language.

The majority of this report was written more than four years before its eventual publication. We have updated it to reflect subsequent developments, but in some places the four year delay really shows, most notably in the performance chapter.

At this writing (January 2002), Vesta is about to become available as free software running on the Linux operating system. The original implementation ran only under Tru64 Unix on Alpha processors, but ports to both Alpha Linux and 32-bit Intel Linux have recently been completed. Compaq Computer Corporation has approved the release of the Vesta source code under the LGPL [19], and it will soon be made available for download from the Vesta web site [51].

Chapter 2

Related Work

Most software configuration management systems in use today are a collection of loosely integrated tools. These tools were initially developed with small-scale systems in mind, so problems arise when they are applied to larger systems. As a result, development organizations tend to build upon and modify the core tools to suit their own needs. But because the core tools were never designed to scale well, because they were not designed in an integrated way, and most importantly, because there are serious flaws in their approach to the SCM problem, these customized solutions still have problems. Since these tools are so widely used, it is worth considering some of them in more detail.

Perhaps the most common SCM tools in use today are RCS (the Revision Control System) [47, 48], CVS (the Concurrent Versions System) [22], and Make [17]. They are often used together, with RCS or CVS handling version management and source control, and Make handling system modeling and building. Later in this chapter we also discuss the integrated systems DSEE, ClearCASE, and Vesta's predecessor Vesta-1. We conclude the chapter with a brief survey of ideas from other systems.

2.1 RCS

RCS is a tool for storing multiple versions of individual source files. A file's version history can branch into an arbitrarily complex tree. RCS provides locking to enforce source control, and includes tools for merging changes made by different developers.

RCS stores multiple versions of a source file in a single disk file, in a way that avoids duplicating material that is common to more than one version. This technique saves disk space, but makes the individual versions inconvenient to access: a particular version cannot be accessed directly, but must be extracted into a separate file first. In a modern computing system, the benefits of storing multiple versions of a source file compactly are slight, since the disk requirements of most development environments are dominated by the space consumed by derived files. Moreover, the price of disk space is dropping exponentially, while the programmers who create source files are not typing any faster.

In place of RCS, some organizations use SCCS (the Source Code Control System) [43], an older but essentially similar system.

2.2 CVS

One disadvantage of both RCS and SCCS is that source files are versioned individually. Although RCS provides mechanisms for *tagging* a group of versioned files, those mechanisms are manual and error-prone. CVS attempts to remedy this problem. CVS is a front-end to RCS that extends the notion of version control from individual files to arbitrary directory trees called *modules*. In CVS, the unit of checkout is an entire module. Hence, it is easier to work on a group of related files with CVS. A drawback is that whenever a user checks out a module for the first time, all the files are copied, which can be slow.

There is another important difference between CVS and the other systems: CVS does not use locking to enforce source control. Instead, it uses an optimistic concurrency control model in which each developer is free to modify (a copy of) any source in the central repository at any time. CVS includes a facility for mechanically merging changes made by other developers into one's own source tree. Developers typically apply this facility just before checking in their own changes, without giving much thought to whether the two sets of changes are compatible. The assumption is that if the changes do not both affect the same region of the same file, there is no problem. Changes in the same region are reported as conflicts, and the developer is required to fix them before checking in. But if two developers make semantically conflicting edits to different files, or even to distinct portions of the same file, such conflicting changes are not reported. Despite these dangers, some people prefer the CVS approach to concurrency control.

A problem with the CVS implementation is that operations on the CVS repository are not atomic: if user *A* is checking in a module while user *B* is checking it out, *B* may get some but not all of *A*'s changes. Hence, there are time windows in which users can see inconsistent versions of the same CVS module.[1]

[1] This problem is documented as existing in the current release of CVS at this writing, version 1.11.1p1.

2.3 Make

Make is a widely used system modeling and building tool. Make is driven by *makefiles*, which specify the dependencies between the components of a system and provide instructions for building them. When it is run, Make examines the dependencies and rebuilds any components that are out of date. People like Make because the makefile syntax is simple (if a little cryptic), the tool is fairly easy to use, and it can be adapted to other tasks besides building software.

The biggest problems with Make are that (1) it does not maintain dependencies automatically, and (2) many dependencies are simply inexpressible or too costly to express in practice [18]. Because the dependency relation is complicated and changes frequently over time, it is easy to specify too few or too many dependencies. Specifying too many dependencies can lead to unnecessary work being done during a build, while specifying too few dependencies can lead to inconsistent builds.

To alleviate the first problem, tools like *makedepend* [12] have been developed to compute dependencies automatically. But makedepend suffers from at least two deficiencies. First, it detects only certain kinds of dependencies, namely, dependencies between C/C++ source files and any files directly or indirectly included by them. Second, there is no mechanism to run makedepend automatically when dependencies change. Since makedepend can take a substantial amount of time to run, developers tend not to use it as often as necessary. Failure to do so can result in inconsistent builds.

There are no solutions to the problem that some dependencies are impossible or too costly to express in a makefile. Even if makedepend is used religiously, makefiles rarely if ever capture all the dependencies on the environment. For example, every derived file produced by a makefile depends on the building instructions in the makefile itself, but developers typically omit this dependency, because including it would force every derived file to be rebuilt any time the makefile changed. They must thus resort to manually invalidating or deleting those derived files that are affected by such makefile changes, an error-prone process. Other dependencies that are cumbersome or impossible to specify in Make include dependencies on the particular versions of tools (compilers and linkers) used, on command-line switches, and on environment variables.

Make also fails to scale well. A large software system can be specified by a hierarchy of makefiles, in which one makefile invokes Make recursively on other makefiles. However, when Make is invoked on the root makefile, it must run all the way down to the leaves of the dependency tree to check for stale dependencies. Because Make uses file timestamps to test whether a derived file is up to date, the test for stale dependencies requires Make to determine the last-modified time of *every source and derived file* comprising a system. Although some of these timestamps may be cached by the file system, they often require network round-trips to file servers and/or disk reads. Thus, Make's dependency checking can be an extremely time-consuming process for multi-million line software systems. Ironically, the Make approach requires all of those dependency checks even if the entire system to be built is up to date!

A big problem with Make is that it is not integrated with any of the version management tools described above. The lack of integration manifests itself in two important respects. The first and most obvious problem is that the sources to be built must be explicitly checked out before they can be built. Some variants of Make have been hacked to do RCS checkouts on missing source files, but such support is limited. The second problem is that explicit source versions are not specified in Makefiles. Put another way, Make provides no configuration management support: it does not give developers a way to specify which source versions go together. It is the developer's responsibility to check out the correct versions of all components before performing a build, a laborious and error-prone process if one is not building the latest version.

Another problem with Make is that it does not lend itself to hierarchical descriptions. It is possible to get Make to build a hierarchically arranged collection of software components by invoking itself recursively, but there are some problems with doing so. The main impediments to structuring Makefiles recursively are that all subcomponents must be checked out before building and that there are performance costs associated with invoking Make recursively. Specifically, the recursion always has to be done in full—there is no way for Make to determine that a particular recursive invocation can be skipped, except by copying all its dependencies into the parent Makefile, which would defeat the purpose of the recursive structuring. The larger the scale of the software being built, the worse these problems become.

Make's reliance on last-modified times can lead to inconsistent builds in several ways. One instance of this problem occurs when building an older version of a system. When the old source versions are checked out, they may have timestamps that precede the timestamps of the derived files from the most recent build. Hence, Make will conclude that the derived files are up to date. The developer's only safe recourse is to force a scratch build by deleting all of the derived files.

Bell Laboratories' Nmake [18] addresses most of these problems with Make, as well as others, though it does not fully solve them. Nmake includes a built-in static dependency generator that does its own parsing of source files (looking, for example, for `#include` statements in C code); this greatly reduces the likelihood of omitted dependencies, but the dependencies generated can be overly conservative, increasing the amount of rebuilding work needed. Moreover, new parsing support has to be written whenever Nmake is used on code written in a new language. Nmake includes an improved timestamp checking algorithm that fails only when an erroneous system clock gives two different versions of a file the same timestamp, and it caches timestamps and other state to speed up its dependency analysis somewhat.

2.4 DSEE

The DOMAIN Software Engineering Environment (DSEE) effectively addressed many of the problems with Make and standard version management systems [31, 30].

For source control, DSEE used a custom file system that allowed individual versions of source files to be named directly; no separate checkout step was required. However, source files were not usually referenced using explicit version numbers. Instead, files were named without version numbers, and the user's currently selected *configuration thread* would bind a file name to a particular version of the file in the file system. A configuration thread was a list of rules for making such associations. For example, a configuration thread could specify that the checked-out version of a file would be used if one existed, or that the latest version on the main branch would be used otherwise. Hence, the meaning of a name could change over time. In particular, an action taken by one developer could change the meaning of a name used by another developer, a dangerous situation. When many developers are working on a system in parallel, any one of them could "break the build", thereby interrupting everyone else's work.

For building, DSEE read *system models* that enumerated the sources to be built, their dependencies (e.g., header files), and the build rules for constructing the software. DSEE provided automatic derived file management and the capability for the derived files produced by one build to be reused in another. Unfortunately, the DSEE papers do not describe the system modeling language in any detail, nor do they discuss either the consistency guarantees or the performance characteristics of building with derived reuse.

In addition to version management and software construction facilities, DSEE also included work flow facilities required by the broader software engineering process. For the most part, these facilities were independent of the configuration management facilities described above, but there was a small degree of integration between them. For example, checking in a source module might cause a task on a task list to be recorded as being completed. DSEE also included a facility for specifying human-sensible semantic dependencies between sources. For example, the dependence between a program's interface code and its documentation might be recorded as a semantic dependency. Whenever the program's interface was modified, a technical writer would be informed of the need to update the documentation.

2.5 ClearCASE

In the early 1990s, the DSEE developers started a company to build ClearCASE [4], a commercial SCM system based on the DSEE philosophy and now sold by Rational Software Corporation.

ClearCASE's source control mechanism is quite similar to DSEE's. In ClearCASE, the configuration threads are called *views*, but the idea is exactly the same: rules are used to map unversioned file names to particular source versions. ClearCASE views are implemented by a custom *multiversioned* file system that plugs into the operating system's file system switch. As in DSEE, looking up a file in a ClearCASE view requires some form of database access, so an extra step is required on each file access.

ClearCASE differs from DSEE in two important respects. First, ClearCASE is more portable. It runs on both Windows and Unix systems. Second, ClearCASE is Make-based. That is, ClearCASE system models retain the syntax and semantics of Make. However, ClearCASE includes an alternative builder called ClearMake that corrects many of Make's problems. In particular, ClearMake includes a mechanism that, during a build, automatically records the file dependencies and results of each external tool invocation (e.g., compilers and linkers). These cached dependencies and results are then used during subsequent builds to bypass tool invocations if the specified files are unchanged. This produces more reliable incremental builds than standard Make, since it does not depend for correctness on dependency lists created by a user. However, build-order dependencies and dependencies on files outside of ClearCASE's control must still be listed explicitly.

Although the test for stale dependencies has been automated, it is no faster than with standard Make. Only invocations of external tools are cached, so as with Make, ClearMake builds programs upwards from the leaves, rather than downwards from the root. As noted above in the discussion on Make, this approach has serious performance problems in building large systems.[2]

One advantage provided by ClearMake over standard Make is that derived files are managed by the system, and can be shared. Hence, developers can benefit from each other's builds. However, heuristics are used to select the candidate derived files for sharing, so the exact cases in which such sharing is possible are not clear. It is possible for the heuristics to fail to select a valid candidate for sharing, in which case an unnecessary tool invocation will occur. In Vesta, these events are

[2] In Vesta parlance, the ClearMake strategy is analogous to caching only `_run_tool` calls. We show in Section 9.2.3 that much better performance can be achieved by caching larger units of work.

termed *false cache misses,* and they are quite rare. Worse, ClearMake's dependency detection is incomplete, so it can sometimes produce inconsistent builds. In Vesta, dependency detection is automatic and complete, and inconsistent builds do not occur.

The ClearCASE MultiSite product supports replication of ClearCASE source repositories across multiple, geographically distributed sites [3]. Vesta's approach to replication (Section 4.4) differs sharply from that of ClearCASE. In ClearCASE, the choice of what to replicate is made at a coarse grain—an entire Versioned Object Base, of which there are typically only one or a few per site. In Vesta, one can choose what to replicate down to the level of individual versions of software packages, if desired. ClearCASE replicas exhibit eventual consistency; that is, an update algorithm is used that would eventually make the replicas identical if they all were to stop changing for sufficiently long, but there are no clear guarantees on what differences can exist between replicas when changes have been made recently, and an unchanging backup copy of old versions is not consistent with current replicas in any useful sense. Vesta defines a simple, flexible notion of consistency for its replicas that takes advantage of the fact that sources are immutable once they have been added to the repository. ClearCASE's replica update algorithm is operation-based and requires knowledge of the full set of replicas; that is, each ClearCase replica must keep a history of recent operations that have changed it and keep track of which changes have not yet been propagated to each other replica. Vesta's algorithm is state-based and works when the set of replicas is unknown and changing; the replication tool simply compares the states of two replicas, copying any data from the first that is missing and desired in the second. The ClearCASE approach has some advantages; in particular, fewer administrative decisions are required as to what to replicate, and the operation-based approach to updates should scale better when replicas share a great deal of data but very little is changing. However, the Vesta approach is much simpler, provides a clearly defined level of consistency, supports usage patterns where the replicas are more loosely coupled, and has performed well in our experience.

2.6 Vesta-1

The Vesta-2 system described in this report is a follow-on to the earlier Vesta-1 work [11, 13, 24, 32]. Both Vesta-1 and Vesta-2 were based on the idea of building from immutable source files using complete build descriptions. As a result, Vesta-1 shared Vesta-2's benefits of producing repeatable, incremental, and consistent builds of large-scale software.

Vesta-1 saw extensive use at our lab for a period of about 15 months by a group of 25 programmers. It was used to build a sophisticated experimental software environment, which included a custom operating system, a file server, a novel compiling system, an experimental multi-threaded window system, a number of mathematical, graphical, and symbol-manipulation libraries, and a spectrum of applications, including a text editor, a graphical editor, ray-tracing software, an algorithm animation system, some mathematical tools, and Vesta-1 itself. In aggregate, this code base comprised 1.4 million source lines. Most of the components were built with the experimental compiling system (which was continually evolving), and were targeted for two different platforms: a VAX multiprocessor running our in-house operating system, and a MIPS uniprocessor running Unix. In short, this workload provided a serious test of the Vesta approach on a scale comparable to many real-world development projects.

Although Vesta-1 demonstrated the promise of the Vesta approach, like most experimental prototypes it suffered from several design and implementation flaws. Vesta-2 was designed and implemented to address these problems. In particular, Vesta-2's caching is more comprehensive than Vesta-1's, its system modeling language is simpler, its performance is better, its repository is more general, and its implementation is more portable.

Throughout this report, the name Vesta refers to Vesta-2. Where a distinction with Vesta-1 is required, the names Vesta-1 and Vesta-2 are used explicitly.

2.7 Other Systems

We have chosen to discuss only a few software configuration management tools and systems in this chapter, but numerous others are described in the research literature, available commercially, or available as open source [2, 5, 8, 14, 33, 34, 37, 41, 46, 49, 50]. Many of these systems do their version management and source control using paradigms very similar to that of RCS or CVS, and a great many handle system modeling and building using versions of Make with various improvements. Thus, although RCS, CVS, and Make are now very old tools, they are still all too representative of the state of the industry. Of course, the commercial systems often have sophisticated features for bug tracking, workflow management, high-level project dependencies, and graphical user interfaces that go beyond the scope of what we have attempted in Vesta, while the research and open source systems explore a wide variety of ideas.

Some systems use alternative approaches to version management. Several systems name and manage the sets of *changes* in a software artifact rather than complete versions of the artifact itself [8, 37, 49]. One motivation in such systems is the idea that users can synthesize many different versions of an artifact by mixing and matching various change sets. We are skeptical about this idea, since changes from different sets seem quite likely to conflict with one another. Some systems version every object in a hierarchical directory tree, including the directories themselves [33, 34, 50], while others have even more complex models for version management [16]. These approaches seem interesting, but we did not see the need for them and therefore chose a more conservative model of version management for Vesta.

Chapter 3

System Overview

In this chapter, we describe Vesta's main components, the foundations of the Vesta approach, and the system's design parameters.

3.1 System Components

Figure 3.1 shows the major components of the Vesta implementation. In the figure, components that are highly visible to the ordinary user appear toward the left; those that are mostly hidden within the implementation or visible only to administrators appear toward the right. The components in the bottom row are shared servers; at each Vesta installation, or *site*, there is exactly one instance of each. In contrast, the components in the top row can run on any user machine.

We now give a brief overview of each component, then describe in a bit more detail how the components interact.

The *repository server* (lower left) is responsible for long-term data storage in files and directories. Versioned, immutable sources[1] are stored in *immutable source directories*. Developers use *mutable working directories* while editing sources to create new source versions. Vesta's build process uses *temporary build directories* while running tools like compilers and linkers to create new derived files; these directories are not directly visible to users. The immutable source directories and the mutable working directories are typically mounted as `/vesta` and `/vesta-work`, respectively.

Standard file browsing and editing tools are not a part of Vesta proper, but we include them in the diagram to stress that developers can use all their familiar

[1] By *source*, we mean any file stored in the Vesta repository that is not derived mechanically by the Vesta builder. Hence, even files such as build tools and library archives copied into the Vesta repository are considered to be sources.

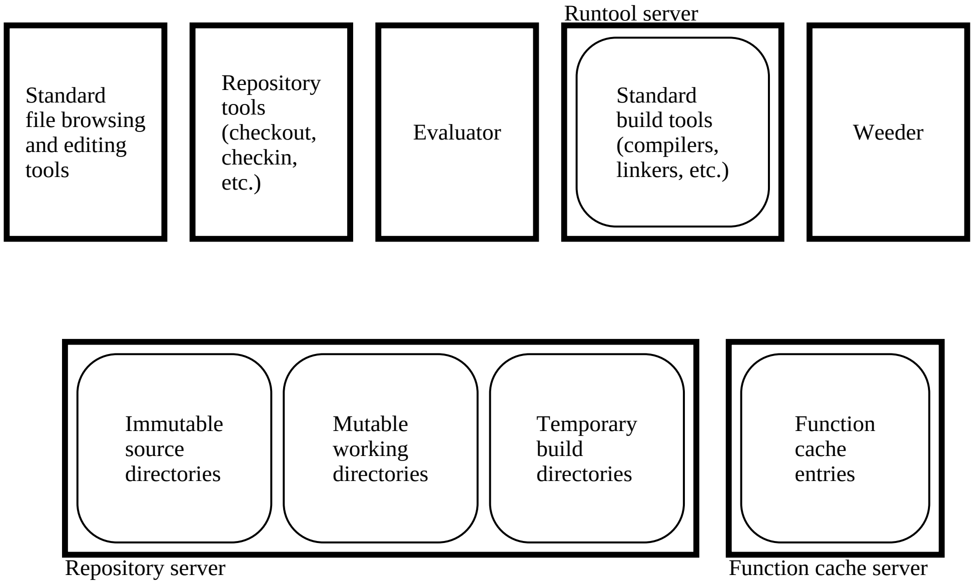

Runtool server
Standard file browsing and editing tools
Repository tools (checkout, checkin, etc.)
Evaluator
Standard build tools (compilers, linkers, etc.)
Weeder
Immutable source directories
Mutable working directories
Temporary build directories
Function cache entries
Repository server
Function cache server

Figure 3.1: The major components of the Vesta implementation.

text browsing, comparing, editing, and processing tools on files in the repository. Developers invoke the *repository tools* to create new source versions or to do other operations on the repository that do not fit directly into the standard file system paradigm.

The *evaluator* is Vesta's builder; it evaluates system models written in Vesta's system description language to construct complete software systems from their components. The evaluator makes use of the *runtool server* to run standard build tools like compilers and linkers when needed; it makes use of the *function cache server* to store intermediate and final results of each build for later reuse.

Finally, the *weeder* is a garbage collector for long-term storage; it triggers the deletion of cache entries and derived files when an administrator declares they are no longer needed.

3.1.1 Source Control Components

Figure 3.2 highlights Vesta's source control components and shows how they interact. Chapter 4 describes these components in detail.

The Vesta repository stores immutable sources in a hierarchical namespace, similar to a Unix or Windows directory tree. Every version of every source is included in the tree; by convention, different versions of the same source are dis-

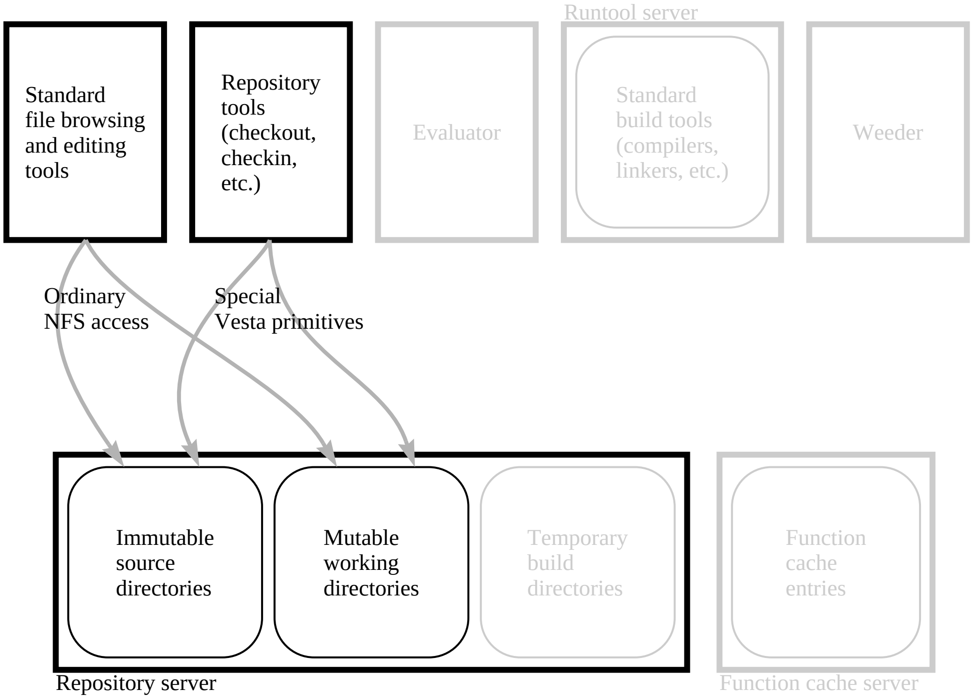

Runtool server
Standard file browsing and editing tools
Repository tools (checkout, checkin, etc.)
Evaluator
Standard build tools (compilers, linkers, etc.)
Weeder
Ordinary NFS access
Special Vesta primitives
Immutable source directories
Mutable working directories
Temporary build directories
Function cache entries
Repository server
Function cache server

Figure 3.2: Source control components and their interactions.

tinguished by having a version name or number as one pathname component. As shown, the repository makes this tree available as a network-accessible file system, using the standard NFS protocol [44]. Thus, ordinary file browsing and editing tools running on any user workstation that supports NFS can access all versions of all sources directly.

In the repository, sources are conventionally organized in *packages*. A package is a collection of related files, such as the sources to build a single program or library. By convention, Vesta sources are versioned at the package level, not at the level of individual files. Thus a version of a package consists of a directory tree of related files. Contrast this with the more conventional (RCS) method of versioning every source file, which provides no natural method for identifying which versions go together.

Vesta uses a checkout-checkin source control paradigm, but the process works in a slightly unusual way. Because source files are immutable, a checkin operation never deletes existing files or renders them inaccessible. Instead, checkout-checkin operations add to the name space of package versions. Checking out a package reserves a version name and makes a mutable working copy of the existing files and subdirectories from the package's previous version (if any). Standard tools can then be used to modify, create, delete, or rename files and directories in the working copy. To ensure that builds are reproducible, the builder operates only on immutable snapshots of the working copy, not on the working copy itself. Checking in the package binds the previously reserved version name to the final snapshot of the working copy. Checkin, snapshotting, checkout, and other repository operations that do not fit the NFS file access paradigm are handled by the repository tools; as shown in Figure 3.2, the tools work by invoking special repository primitives through an RPC (remote procedure call) interface.

To support development of software across geographically distributed sites, the repository server at one site can replicate some or all of its sources from repository servers at other sites. Vesta's support for such *partial replication* is described in Section 4.4. To achieve partial replication, repository servers at different sites communicate with each other through the RPC interface (not shown in the figure).

3.1.2 Build Components

Figure 3.3 highlights the components that participate in Vesta builds and shows how they interact. Building is quite a complex process, involving many components that interact in subtle ways to ensure that builds are reproducible, incremental, and consistent.

The Vesta evaluator is the center of the build process. The evaluator reads a system model and acts on it, building what the model describes. As shown in Figure 3.3 (arrow 1), models are always read from the immutable part of the repository, to help ensure that builds are always reproducible. A model describes how to build a software artifact from source, and the sources it refers to are also stored in the immutable part of the repository. Models are written in the Vesta system description language (SDL), a small functional programming language whose data types and primitives are specialized for software construction. Chapter 5 describes the language in general, Appendix A gives its complete syntax and semantics, and Chapter 7 describes the evaluator.

Whenever the evaluator encounters a function call in a model, it looks in the function cache (arrow 2) to see if a sufficiently similar call has already been evaluated in a previous build. If so, it reads the result from the cache instead of evaluating the function again. Function cache hits can occur at any level in the call graph, from the leaves (usually individual calls to a standard build tool such as a compiler or linker) up to the root (the entire build being requested). Unlike Vesta, most other build systems cache only at the leaves, and thus do not scale well to large builds. Chapter 6 describes the function cache.

What does it mean for a previous function call to be *sufficiently similar* to the current one? In more detail, we need the same function to have been called in

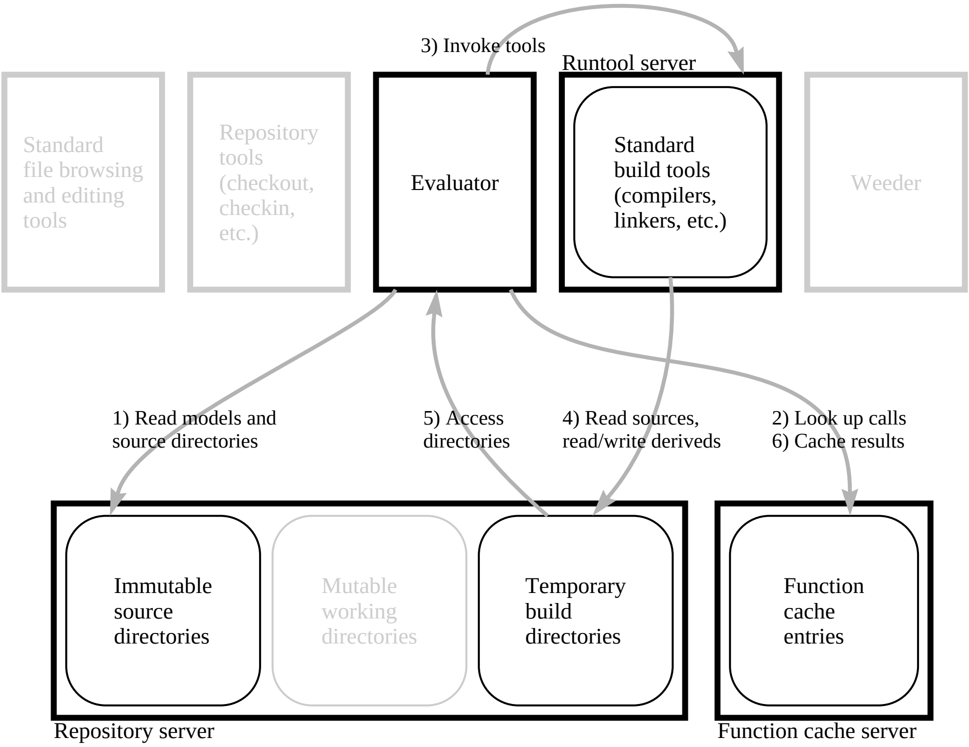

3) Invoke tools
Runtool server
Standard file browsing and editing tools
Repository tools (checkout, checkin, etc.)
Evaluator
Standard build tools (compilers, linkers, etc.)
Weeder
1) Read models and source directories
5) Access directories
4) Read sources, read/write deriveds
2) Look up calls
6) Cache results
Immutable source directories
Mutable working directories
Temporary build directories
Function cache entries
Repository server
Function cache server

Figure 3.3: Component interactions during a build.

a sufficiently similar naming environment, including both arguments and names bound from the function's static scope, that the result is guaranteed to be the same. It would of course be correct to check that *all* names in the environment are the same, but doing so would give us an unacceptably low cache hit rate. For example, when the C compiler is invoked, all of the .h files in /usr/include are in its environment, but a typical .c file uses only a few of these .h files. The result of the compilation does not depend on the whole environment, only on the part that is actually referenced.

Therefore, when Vesta evaluates a function, it records the *dynamic fine-grained dependencies* of the function's result on its naming environment. By *dynamic*, we mean that the evaluator records only what is referenced during this particular evaluation. By *fine-grained*, we mean that when only part of a composite value is referenced, the evaluator records a dependency on just that part, not on the whole value. In the C example, if a particular .h file is not used, no dependency on it is recorded. On cache lookups, then, a hit occurs whenever we can find a cache entry for the current function whose dependencies were bound to the same values in both the entry's original environment and the current environment. Vesta's dependency analysis does not make use of any knowledge of how the build tools work; it is thus *semantics-independent* in the terminology of Gunter [23].

When a call to a build tool misses in the cache, the evaluator must invoke the tool itself to do the work. It does so by making a remote procedure call to the runtool server (arrow 3), which is responsible for starting the tool, waiting for it to complete, and reporting its outcome back to the evaluator. We place this functionality in a separate server so that the evaluator can invoke tools on remote machines. This lets us support parallel compilation (by invoking runtool servers on multiple machines), and cross-platform development (by invoking the runtool server on a machine with a different architecture from the local machine, when a cross-compiler is not available).

Build tools are run in an *encapsulated environment*. That is, Vesta controls not only the tool's command line and environment variables, but also the entire file system content that the tool sees. Moreover, Vesta monitors and records each file system reference that the tool makes. We accomplish this by forcing all of the tool's file and directory references to go through the repository (arrow 4).[2]

The files and directories that an encapsulated tool sees are defined as a data structure within the ongoing model evaluation, but it is the repository that manifests this data structure to the tool, as a tree of temporary build directories (see figure). The evaluator does not pass the data structure to the repository all at once; instead, whenever the tool makes a directory reference that the repository has not seen before, the repository calls back to the evaluator (arrow 5) to obtain the corresponding value, and the evaluator notes the reference as a dependency. Thus the evaluator is able to record fine-grained dependencies not only for functions written entirely in the Vesta SDL, but also for tool invocations. At the end of a tool execution, the repository reports to the evaluator what new files and directories the tool created (and any other changes the tool made) as its output. Section 4.2.2 describes the repository machinery for tool encapsulation in more detail.

After the evaluator finishes executing a function, it writes a new cache entry (arrow 6) to record the function result and its dependencies. Vesta uses a persistent, shared cache server so that a build done today can benefit from work already done in the past, and so that a build requested by one user can benefit from work already done on behalf of another user.

As a final step, not shown in the figure, the evaluator can optionally *ship* the results of the build. That is, it can copy out some or all of the results of the evalu-

[2] Under Unix, we use the `chroot` system call to redefine the tool's root directory ("/"), thus ensuring that it can reference only files and directories supplied by the repository.

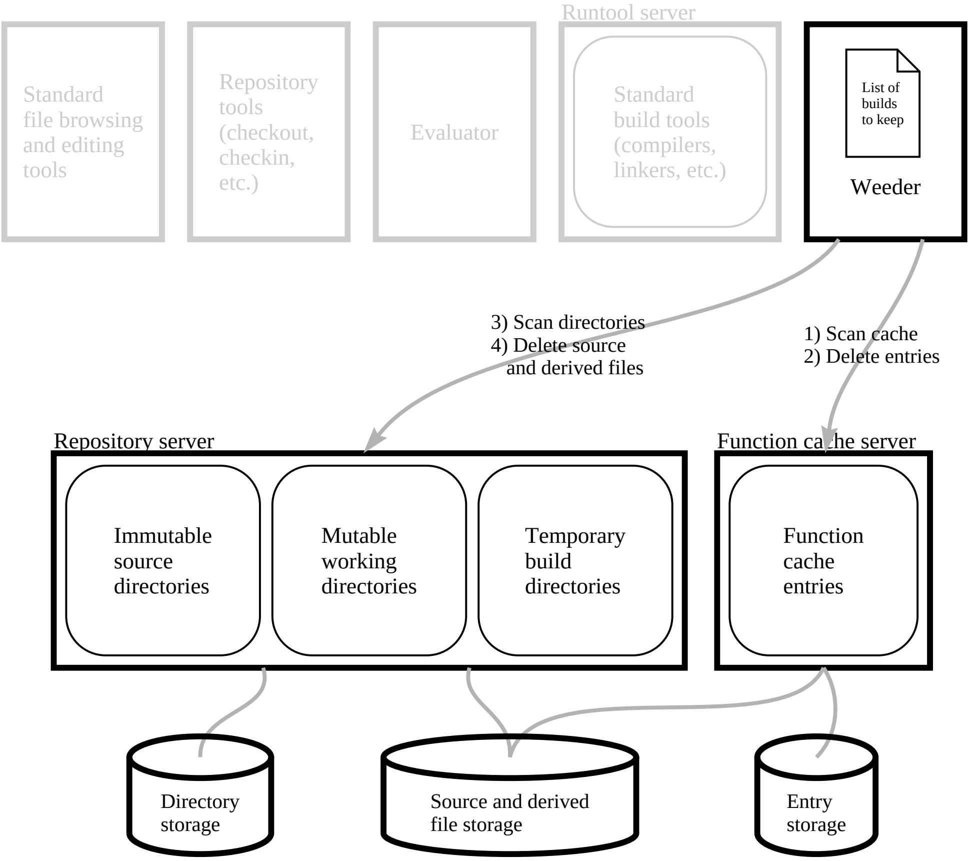

Runtool server
Standard file browsing and editing tools
Repository tools (checkout, checkin, etc.)
Evaluator
Standard build tools (compilers, linkers, etc.)
List of builds to keep
Weeder
3) Scan directories
4) Delete source and derived files
1) Scan cache
2) Delete entries
Repository server
Function cache server
Immutable source directories
Mutable working directories
Temporary build directories
Function cache entries
Directory storage
Source and derived file storage
Entry storage

Figure 3.4: Disk storage and the weeder.

ation's top-level function call to permanent files and directories. Note that even if the results of a build are not shipped, they remain in the function cache and can be quickly retrieved by repeating the build.

3.1.3 Storage Components

Figure 3.4 shows the three pools of long-term disk storage used by Vesta components and illustrates the operation of the weeder, an administrative tool for reclaiming storage that is no longer needed.

As shown at the bottom of the figure, the repository has a private storage area for directory entries and the function cache has a private area for cache entries, but they share a common pool of storage for source and derived files. This pool is managed using garbage collection: when neither a source directory entry nor a function cache entry points to a file, it can be deleted. At different times in its history, the same file can be pointed to by a directory entry, a cache entry, or both. Section 4.2.1 describes the file pool in more detail.

As builds are performed, cache entries and derived files accumulate in the function cache and file pool, and can eventually grow to fill the available disk space. Fortunately, any or all cache entries can be deleted without affecting the repeatability or consistency of builds; however, some future builds that would otherwise get cache hits (and hence be incremental) may have to be done partly or entirely from scratch. Deciding which entries are best to remove and which should be retained is a task that cannot be entirely automated; users and Vesta administrators must decide which ones are worth keeping, based on their knowledge of what builds are likely to be requested in the future.

Unwanted cache entries and derived files are deleted by the Vesta *weeder*, an administrative program that is run periodically. Given a specification of which package builds to keep, it deletes all cache entries that did not participate in those builds (steps 1 and 2 in the figure). It then contacts the repository (steps 3 and 4) and deletes all files in the shared pool that are neither pointed to by remaining cache entries nor pointed to by the repository directory structure. Since weeding can take a relatively long time (minutes to hours), the function cache, repository, and weeder have been designed so that the weeder can be run concurrently with client builds without adversely affecting normal build performance. The weeder is described in Chapter 8.

3.1.4 Models and Modularity

Vesta system models are *modular*—that is, each model can import other models and use the functions they define. A model can describe how to build a collection of sources into a subsystem, then export that subsystem as a unit, for use by higher-level models that assemble the subsystems into a complete system.

Modularity is essential for scalability to large systems, but it is also important even for building small programs. In today's programming environments, even a small "hello world" program is compiled and linked against a large runtime library of input/output and operating system interface routines. Moreover, the commands needed to invoke tools like compilers, linkers, stub generators, and the like can be complex. Vesta provides a *standard environment* model that encapsulates these libraries and common building actions and makes them available to user-written models in a simple form. The standard environment can be quite complex internally—for example, it can build some or all of the standard libraries and tools from source—without exposing any of these complexities to the ordinary user. At the same time, because it is written as a model rather than being hardwired into

Vesta's implementation, more advanced users can extend, modify, or replace the standard environment to suit their unique needs.

Models for ordinary applications are typically written so that they can be imported too. One can then write a *release model* that imports one version of the standard environment and builds a whole collection of applications against it. Thus a development shop that maintains a suite of applications can easily create a consistent release, in which all the applications are known to have been built with the same tools against the same libraries.

The system modeling language, system models, and the standard environment are described more fully in Chapter 5.

3.2 Foundations

We have claimed that Vesta provides repeatability, consistency, incrementality, and scalability in software version management and building. The foundation for these claims lies in Vesta's novel combination of structural features. Now that we have given an overview of these features, we are prepared to summarize the essential contributions of each one to Vesta's overall goals.

Immutable, Immortal, Versioned Sources. All Vesta builds are performed on immutable, immortal, versioned sources. Before each build, a developer's mutable sources are copied to an immutable, versioned form. The immutability and immortality of sources are essential for achieving repeatable builds.

Complete, Source-Based Build Descriptions. A Vesta build description gives a complete recipe for building a software artifact from versioned sources. By complete we mean that no aspect of the build relies on any aspect of the computing environment outside of Vesta's control, including environment variables, library archives, and build tools. The complete nature of Vesta's build descriptions is essential for achieving consistent builds. Because build descriptions name versioned sources, the meaning of a name cannot change over time.

Automatic Dependency Detection. A build system that aspires to perform consistent builds must know the dependencies of every derived file. Vesta detects and records all such dependencies automatically. By using automatic detection rather than relying on user-supplied (and thus error-prone) dependency specifications, Vesta can collect all the information needed to determine when (re)building is necessary, and can thereby ensure that the results of its builds are consistent.

Caching. To support incremental building, Vesta automatically caches the results of tool invocations (and their associated dependencies) for later re-use. For scalability, larger units of work — such as the construction of entire libraries — are also cached. Because Vesta uses a shared site-wide cache, each developer can benefit from the builds of all others.

Hierarchical Build Descriptions. The Vesta system modeling language allows build descriptions to be structured hierarchically. Hierarchical, modular structuring is the only effective way we know to describe large-scale software configurations. It also permits re-use of common build functionality, such as that provided by Vesta's standard construction environment.

Source Replication. As software grows larger and as long-distance telecommuting becomes more common, the need for development of software systems across multiple, geographically-distributed sites increases. However, groups at different sites may need to share only some of their sources. Vesta allows some or all of the sources at one site to be replicated at other sites, using an algorithm that avoids copying sources unnecessarily. Moreover, control of each package can be shifted to the site developing it most actively, allowing users at that site to create new versions autonomously.

3.3 Design Targets

Before designing and implementing Vesta-2, we needed to decide how large a software system we expected it to be able to handle. That decision had immediate consequences on the load imposed on each of the system's components and on the system's overall resource usage. Once chosen, these targets also played a critical role in our subsequent engineering decisions.

The Vesta-2 design targets given below were based on our experience with Vesta-1, which had been used to build an actively changing code base of 1.4 million source lines. Our goal for Vesta-2 was that it build software systems at least an order of magnitude larger than those built with Vesta-1.

Code Size. Vesta-2 should be able to build a system comprising approximately 20 million lines of source code.

Derived Files. There should be approximately 1 million derived files of interest at each site. Hence, the system should be able to accommodate 2 million derived files comfortably (before weeding becomes necessary).

Cache Entries. We expect 5–6 cache entries for each derived file. Hence, the function cache must support 10–12 million cache entries.

The implementation should be able to comfortably accommodate a software system of this size. Moreover, when used to build somewhat larger systems, the system should continue to work; it should not suffer any major performance problems. Our intent in setting these targets was not to place an upper bound on the size of system Vesta could handle, but a lower bound. The targets reminded us to "think big" throughout the design process.

Although we did not have the opportunity to test the Vesta implementation on a software system of this size, we were able to apply it to large enough problems to partly validate our design decisions. Examples of design decisions that were made with an eye toward scalability appear throughout this report. We present the results of our performance testing in Chapter 9, and we briefly discuss our experience with real users in Chapter 10.

Chapter 4

Repository

The Vesta repository is responsible for long term storage of source and derived files. The repository provides two distinct sets of services to two kinds of clients: users and the evaluator. Users are mainly concerned with sources; they read existing source files and create new ones. The evaluator is concerned with both sources and deriveds; it reads system models (which are sources), and it runs tools that read sources and deriveds and write out new deriveds. The source storage service and its replication support are sufficiently comprehensive and independent that the repository could be useful alone, without the evaluator or the rest of Vesta.

The repository is implemented in two layers. The *repository server* is responsible for source naming and storage, while the *repository tools* provide a user interface to the server. The distinction between the server and the tools is significant. The tools we have built support a particular style of naming sources and carrying out common development tasks, but the repository server itself is quite general and could support other styles of use. Most of the complexity of the system is in the repository server. The existing repository tools are short command-line programs, each less than 1000 lines of code.

In the next three sections, we discuss the main features of the repository from the user's point of view, from the evaluator's point of view, and from the implementor's point of view. In the final section of the chapter, we discuss the repository's support for source replication and the replication tools we have built.

4.1 The User's View

This section describes the aspects of the repository that are directly visible to users.

4.1.1 Source Namespace

The Vesta repository provides a hierarchical source namespace, similar to a Unix or Windows directory tree, but with a few additional features and restrictions to support configuration management.

For user convenience, Vesta makes the source namespace available as a subtree of the file namespace on each client machine. Users can apply all their existing, familiar tools for browsing ordinary files and directories to repository files and directories. The repository server makes this possible by exporting the source tree as an NFS volume that can be imported and mounted by client machines. The server also exports two remote procedure call (RPC) interfaces for access to features that do not map well onto NFS.

To support Vesta's goal of repeatable builds, all source files accessible to the evaluator are *immutable.* Once a file is created and placed in the source namespace, the file's contents cannot be modified. In addition, Vesta's philosophy of software development says that source code ought to be *immortal*; all sources should be kept permanently so that builds can always be reproduced. We recognize, however, that sources sometimes must be deleted for practical reasons, such as a lack of disk space or the expiration of a licensing agreement. So we do allow source files and directories to be deleted, but to preserve consistency in builds, we do not allow a deleted source's full pathname to be reused later for a different source. Thus, repeating a build will either succeed and produce an identical result, or will fail because some source file has been explicitly deleted. The latter case will not occur if users follow our development philosophy.

Vesta source directories thus must not be arbitrarily mutable. Vesta in fact supports two kinds of directory with limited mutability: *immutable* and *appendable.*

Like an immutable file, an immutable directory cannot be changed once it is populated with files and subdirectories and placed in the source namespace. Every file in such a directory must be immutable, and so must every subdirectory, so that the entire tree rooted at it can be treated as one immutable unit.

Appendable directories support and enforce Vesta's rule that names cannot be reused. An appendable directory is similar to an append-only file. New names can be created in the directory, but existing names cannot be unbound or freely rebound to different contents.

We do allow certain strictly limited forms of rebinding, involving special objects called *ghosts* and *stubs.* If the Vesta evaluator encounters a ghost or stub during a build, it halts with an error message, does not produce a result, and does not record the error in the Vesta function cache. Therefore these cases of rebinding cannot cause the same build to have different results on different occasions; as with deletion, they can at worst cause a build that succeeded on one occasion to fail on another.

Ghosts support Vesta's deletion semantics. When a user deletes an object from an appendable directory, Vesta replaces the object with a ghost. The ghost keeps the old name from being reused; Vesta does not allow a name bound to a ghost to be rebound to a new object, and attempting to delete a ghost has no effect.

Stubs support more specialized features of Vesta—name reservation and partial replication—which we describe in Sections 4.1.4 and 4.4 below. A stub is a placeholder for a file or directory that may be supplied in the future. It is permissible to replace an existing file or directory with a stub, as long as it is guaranteed that only the *same* file or directory will later replace the stub.

Both ghosts and stubs are manifested through the NFS interface as zero-length files that cannot be read or written. Their access permission bits are set to distinctive values that make them distinguishable from real files in a directory listing.

4.1.2 Versioning

How can an append-only namespace of immutable files and directories support software development? Software systems that are under active development constantly grow and change through modification of the files they contain. The solution, of course, is to adopt a naming convention in which each pathname includes a version number. Then instead of modifying a file or directory in place, one creates a new version.

The repository server makes no judgment as to what part of a pathname should be the version number, but for several reasons, we have found versioning at the level of directory trees to be convenient, and have designed the repository tools to use it.

Most programs consist of several files that make up a logical unit, which we call a *package*. At minimum a package includes one file of source code and one of building instructions. A package can also be larger, consisting typically of several closely related files of code, interfaces, and documentation, perhaps organized in a tree of subdirectories. Large programs can generally be decomposed into several packages, each relatively independent of the others. It is convenient to store each package as a separate directory or directory tree.

When a package is modified, often several files in it must change together for consistency. In systems like CVS, where every file is versioned separately, a separate data structure and tools are required to keep track of which versions of the files in a package go together to make a coherent version of the whole package. In Vesta, each coherent package version simply corresponds to one immutable directory with one hierarchical file name, with the version number as a component of the name. For example, the directories `thread/5` and `thread/10` would be versions 5 and 10 of the thread package, and the files `thread/5/foo.c` and `thread/5/foo.h` would be consistent versions of its components foo.c and foo.h. If foo.c is unchanged between two versions of a package, the repository links the same immutable file into both version directories; only one copy exists on disk.

All package versions are directly available for browsing and building at all times. A user can easily see what is in a particular version by changing to its directory and looking around, and can easily compare two versions with existing tools like **diff**. There is no need to check out a private copy of a package unless it is to be modified.

A secondary reason for versioning at the level of directories is that existing Unix and Windows tools are not prepared to deal with version numbers in file names. Vesta "hides" the numbers from them by placing them earlier than the last component in the pathname. For example, a Unix or Windows C compiler is better adapted to deal with file names of the form `2/foo.c` than `foo.c/2` or `foo.c;2`. The repository server could support any of these three versioning schemes, as it allows immutable files to be placed directly in appendable directories, but the repository tools support only the first.

Of course, when a large software system is made up of several separately versioned packages, we still need a way to keep track of which versions of the *packages* go together to make a coherent system. Rather than trying to solve that problem within the repository, Vesta deals with it in the system modeling language; see Section 5.3.3. One might argue that if a mechanism for naming package versions is required in the modeling language, we might just as well use that mechanism to name individual file versions and dispense with package-level versioning in the repository. But although this approach would be technically feasible, it would be far less convenient for users than the approach we have chosen—models would be cluttered with large numbers of versioned file names instead of containing only a few versioned package names.

4.1.3 Naming Convention

A hierarchical namespace gives great freedom in assigning names to files. To avoid chaos, one needs to organize the namespace: to establish conventions on how files are named so that people can find them. Vesta's repository server does not enforce any particular naming convention, but the repository tools do. Figure 4.1 shows part of a typical repository directory structure.

The root of the subtree in the figure is named `/vesta/west.vestasys.org`. This name is chosen to be globally unique across all Vesta installations, for reasons discussed in Section 4.4 below.

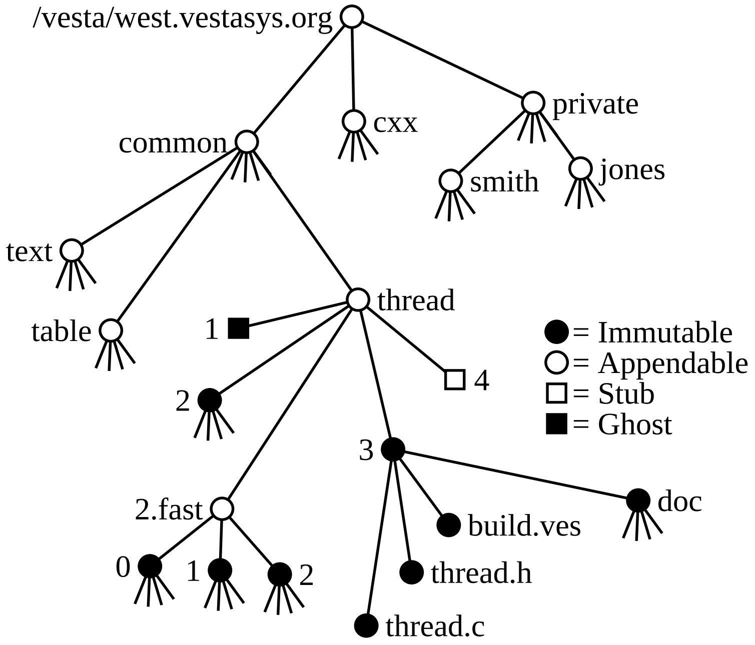


Figure 4.1: Naming convention example.

Below the root is a tree of appendable directories used for categorizing packages. In this example, packages that are generally useful are placed in the `common` directory, packages that are part of the C++ compilation system are in `cxx`, and private packages owned by users Smith and Jones are in `private/smith` and `private/jones`. This directory tree can have arbitrary shape; the repository tools place no restrictions on it.

At the next level down in the tree are individual packages such as `text`, `table`, and `thread`. The thread package is shown in detail in the figure. The immutable subdirectories `thread/2` and `thread/3` are versions of the package; the latter is shown as containing three immutable files and an immutable subdirectory. Version `thread/1` has been deleted, leaving a ghost in its place to keep the name from being reused. The name `thread/4` is bound to a stub, reserving it for a new version that a user is working on and has not checked in yet.

The appendable directory `thread/2.fast` is a *branch* of the thread package. While versions 1, 2, 3, . . . represent the main line of development, the versions under `2.fast` are another line of development branching off from version 2. Essentially, `thread/2.fast` is a new package, whose version 0 is identical to version 2 of the thread package. Branches off of branches are also possible; for example, a branch from `thread/2.fast` could be named `thread/2.fast/1.bugfix`.

4.1.4 Development Cycle Tools

The *development cycle* is Vesta's name for the sequence of steps that users typically go through when developing software. The outline below shows the cycle and gives the name of the Vesta command used in each step. The command **vesta** is the Vesta evaluator; the other commands shown are repository tools.

1. Check out a package: **vcheckout**
2. Modify the package
   (a) Edit: any text editor
   (b) Advance: **vadvance**
   (c) Build: **vesta**
   (d) Test
   (e) Go back to step 2a until done.
3. Check in the results: **vcheckin**.
4. Optionally go back to step 1.

Outer Loop

In the outer loop of the development cycle, one checks out a package (1), modifies it (2), and checks in the result as the next version (3). We call one trip around this cycle a *session*. Checkout creates a mutable working copy of the package. Modification is the inner loop of development, discussed next. Checkin writes an immutable snapshot of the working copy into the repository. Both checkout and checkin use copy-on-write to improve performance and space efficiency, as described in Section 4.3.4 below.

The repository tools we have implemented use a locking paradigm for checkout. Checking out a package reserves a name (that is, a version number) for the modified version that is to be checked in later. No other user can reserve the same name, and normally only the user who reserved a name is given permission to check the package back in under that name. By default, **vcheckout** tries to reserve a version number that is one greater than the highest checked-in version, so it will report a conflict if two users try to check out the same package.

For example, in Figure 4.1, a user (say, Smith) has checked out the thread package with the command **vcheckout common/thread**, thereby reserving the name `/vesta/west.vestasys.org/common/thread/4` for the next version. The reservation is manifested in the repository namespace as a stub, owned and writable only by Smith. If user Jones asks to check out the thread package too, he will be told that version 4 is already reserved by Smith. Jones is thus alerted to speak to Smith and make sure their intended changes will not be redundant or conflicting.

If Jones wants to proceed in parallel with Smith, he can ask **vcheckout** to reserve a different version number, in any of several ways. In the unusual case where Jones's changes are meant to completely subsume Smith's, Jones can leapfrog Smith by asking to reserve version `thread/5`. If Jones is just making one variant version, he can choose a name outside the main sequence of versions, say `thread/3-jones`. If Jones is starting a new line of development that may proceed for several versions before merging back into the main line, or may never merge back in, he can create a branch using the **vbranch** tool. In our example, Jones could type **vbranch common/thread/3.jones**, which would create a new package with the specified name. Version `common/thread/3.jones/0` would be identical to `common/thread/3`. Jones could then check out the branch and work on it as in a normal package.

In none of these cases does Jones need to "break a lock." Unlike RCS, Vesta does not lock the old version that Smith started from; instead, Vesta locks the new version by creating a stub for it. So there is nothing to interfere with Jones starting a different line of development that branches off from the same old version.

Returning to the main line of the outer loop, when Smith has finished modifying the package, he simply types **vcheckin**. The **vcheckin** tool replaces the reservation stub that was created by **vcheckout** with an immutable snapshot of Smith's mutable working copy. It also deletes Smith's working copy, so that he cannot inadvertently continue to edit it after his session has ended.

Inner Loop

The inner loop of the development cycle is the familiar edit-build-test sequence, with the addition of one step that is unique to Vesta: advance. As stated previously, the Vesta evaluator guarantees build reproducibility by building only from immutable sources stored in the repository. Even the private builds that an individual developer does on work he is not ready to check in for public use are handled in this way. After editing and before building, a Vesta developer runs the **vadvance** command. This command takes an immutable snapshot of the developer's mutable working copy and puts it into the repository namespace under a new version

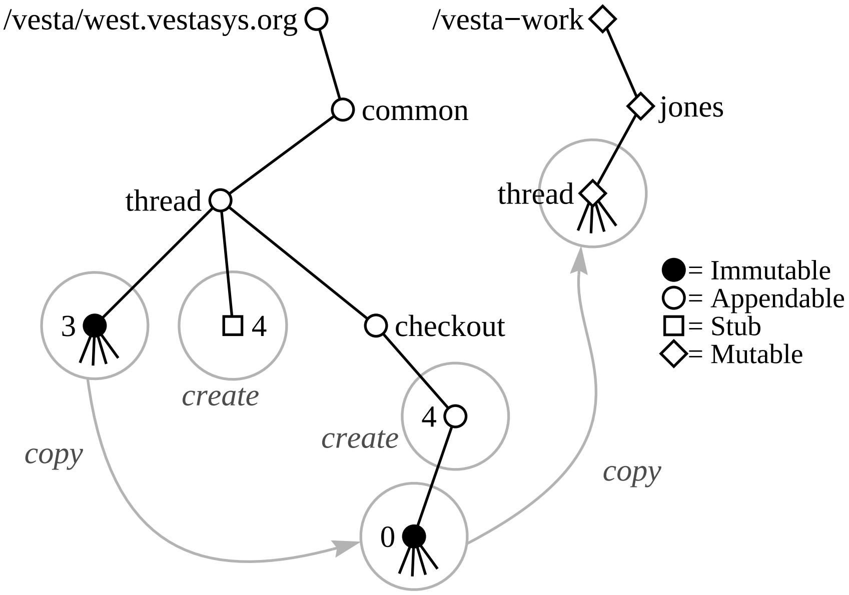


Figure 4.2: Action of the command **vcheckout common/thread**.

number within the session.

To keep the many snapshots created during a session from cluttering up the shared namespace, the repository tools put them in a directory that is separate from the main-line versions of the package, called the *session directory*. A session directory is essentially a branch with a special name, created by **vcheckout**.

Repository Tool Details

Figure 4.2 illustrates the complete operation of **vcheckout**. Initially, `thread/3` is the latest version of the thread package. When user Jones types **vcheckout common/thread**, the system creates the reservation stub `thread/4` and the session directory `thread/checkout/4` in the repository's appendable source tree `/vesta`. It creates the mutable working copy `jones/thread` in a separate mutable directory tree named `/vesta-work`. The session is given an initial version `thread/checkout/4/0` whose contents are immutable and identical to the last checked-in version in the main line of development, `thread/checkout/3`. The working copy initially has the same contents as well, but it is fully mutable.

Jones can now edit the files in his working copy with any text editor or other file manipulation tool. He can freely create new files or subdirectories and delete or rename existing ones, using ordinary commands like **cp**, **mv**, **rm**, **mkdir**, and

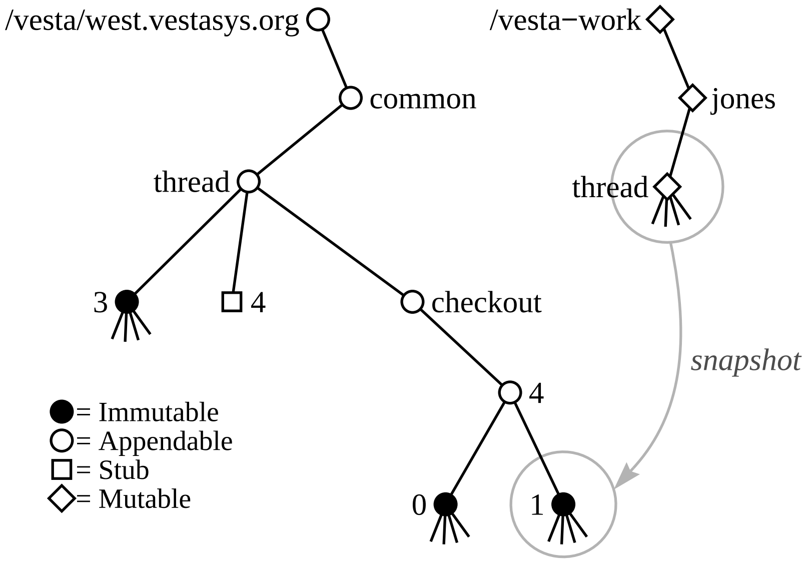


Figure 4.3: Action of the command **vadvance** in the directory `/vesta-work/jones/thread`.

rmdir.

Whenever Jones wants to try compiling his modified files, he types **vadvance** to save an immutable snapshot as the next higher version in the current session directory. Figure 4.3 illustrates the operation of **vadvance**. It simply makes an immutable copy of the working directory in the package's session directory; the name of this copy is the next available version number in the session. Jones can of course use **vadvance** even when he is not about to do a build; for example, he could use it to checkpoint his current work in preparation for making experimental changes.

Jones uses the **vesta** command to invoke the Vesta evaluator (see Chapter 7) and to build the latest version in the session. He will typically use a short shell script to do both the advance and evaluation in one step.

Next, if his program has built without errors, Jones tests it. If changes are needed, he returns to the editing step and goes around the inner loop again. Any time Jones needs to reexamine an old version or back out a change, he can simply look back at the old snapshot—whether it is in the current session, an old session, the main line of checkins, or elsewhere—and compare or copy the files to his working directory.

Finally, when Jones is satisfied, he ends the outer loop by running **vcheckin**, as

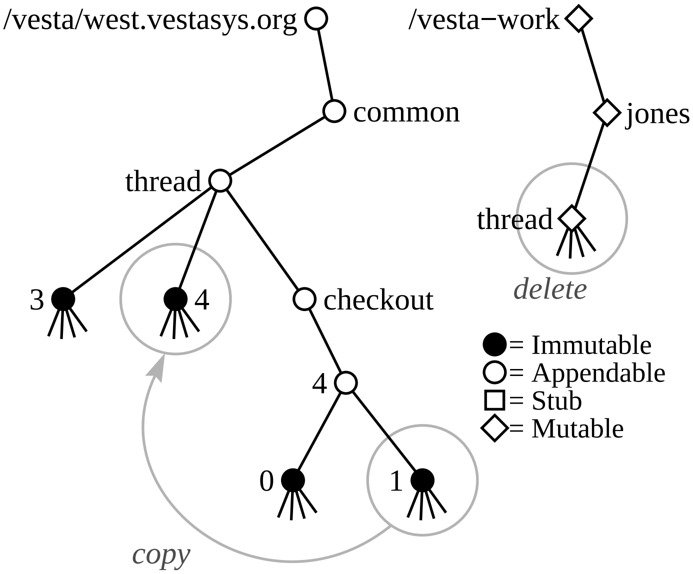


Figure 4.4: Action of the command **vcheckin** in the directory `/vesta-work/jones/thread`.

mentioned earlier. Figure 4.4 illustrates this operation. Note that checkin does not itself snapshot Jones's working copy of the package; instead, it uses the snapshot made by the most recent advance, first checking that the working copy has not been modified since then.

Remarks

The repository server is general enough to support other version control styles. For example, only small changes to the repository tools would be required to support concurrent versioning in the style of CVS. In concurrent versioning, there is no locking at all, and new version numbers are chosen at checkin time rather than checkout time. To support this, we would alter **vcheckout** so that it remembers the version it started from, but does not create a stub. We would alter **vcheckin** so that it tests whether any new version was checked in by another user since the working copy was checked out, prompting the user to merge changes if so. We would also have to change the naming convention for session directories slightly, because the current convention makes the session name a function of the reserved new version name.

We have not written any Vesta-specific tools for merging changes made along different branches, but Vesta makes it easy to merge branches using commonly available tools. Vesta keeps track of the versions created along all the branches, all the way back to the common base versions from which they diverged, and manifests each one as an ordinary file system directory. Thus, one can easily use standard directory-oriented tools such as **diff**, **diff3**, and **patch**, perhaps augmented with some simple shell scripts for convenience. In contrast, RCS and CVS require special variants of these tools that are integrated with their versioning systems.

Additional tools

The **vcreate** tool creates a new package containing no versions. One can then check out the new package and add code to it. When a package has no versions, applying **vcheckout** to it creates an empty directory as the working copy.

The **vsessions** tool provides a simple graphical interface for managing checked-out packages. The tool automatically displays the latest version number in each checkout session belonging to its user. It provides an Advance button for each session, which simply invokes **vadvance** on it.

The **vlatest** tool prints the latest checked-in version number of a given package, or of all the packages and branches in a given directory tree.

The **vwhohas** tool lists the user (if any) who has checked out either a given package or all the packages and branches in a given directory tree.

The **vhistory** tool prints a change log for a package, listing all past versions and their checkin messages.

Our description of the repository tools has touched on a few repository features that we have not yet described in detail. In the next three sections we discuss three of these: mutable files and directories, mutable attributes, and access control.

4.1.5 Mutable Files and Directories

Mutable files and directories in the repository are generally similar to ordinary files and directories in the host file system, but the repository provides additional functionality for efficiently copying data between mutable and immutable directories. The development cycle tools use this functionality to make checkout, advance, and checkin very fast. The added functionality is accessed through a separate RPC interface, not through NFS.

Objects in a mutable directory can be created, deleted (without leaving ghosts), or renamed as desired. Mutable files can be modified freely.

The repository can quickly create a mutable directory whose initial contents are the same as any given immutable directory. No data copying is necessary. We say the new directory is *based on* the old one. It is represented internally as a pointer to the base directory plus a list of changes, initially empty. Thus, each file entry in a mutable directory initially points to an immutable file. If a user tries to modify such a file, however, it is transparently copied to preserve the immutability of the original, and an entry is added to the change list pointing to the copy. A new, empty mutable directory is represented in the same way, but with a null base pointer.

The repository can also quickly create an immutable directory as a snapshot of any given mutable directory. It does this by copying the base pointer and the list of changes. In addition, each file in the mutable directory that was modified or newly created (and hence is mutable) is simply marked as being immutable, without making a copy. If a user later tries to modify the file further through the mutable directory, a new mutable copy is made at that time.

In Figures 4.2–4.4 above, mutable files and directories are shown as open diamonds. As illustrated, the mutable tree under `/vesta-work` is separate from the repository's main tree of appendable and immutable directories under `/vesta`.

Currently, we do not implement symbolic links in mutable directories, and we do not allow multiple hard links to a mutable file. Symbolic links are forbidden because it would not be meaningful to copy a symbolic link into the immutable part of the repository; it is unclear what a symbolic link there should mean if the Vesta evaluator were to encounter one. Multiple hard links are forbidden to simplify the bookkeeping and to guarantee that a mutable file never has to be copied when making an immutable snapshot. In practice, these limitations are inconsequential.

4.1.6 Mutable Attributes

A source control system typically needs to store metadata about sources beyond just their names and contents. For example, the Vesta repository tools need to know whether each appendable directory is a checkout session, a package, or something else, and they need to know the connections between stubs, sessions, and working directories. Users often want to know when and by whom a package was created, checked out, or checked in, and on what previous versions a new version was based.

The Vesta repository provides *mutable attributes* to serve these purposes and others. A source object's attributes are a total function F from string names to sets of string values. If a name n has never been bound to any value, $F(n)$ is the empty set. There are operations to set the value of $F(n)$ to a singleton set or to clear it to the empty set, and operations to add or remove an element from $F(n)$. (Setting $F(n)$ to a singleton is equivalent to atomically clearing it and then adding the value.) There are also several operations to query values of F. This functionality is available through an RPC interface, and the repository tool **vattrib** provides a general purpose command-line interface to it.

Attributes are not visible to the Vesta evaluator, so it is safe for them to be mutable; changing an attribute cannot change the result of an evaluation.

Each source object has attributes unless its parent directory is immutable. Thus in Figure 4.1, the immutable version `common/thread/3` has attributes, but its files and subdirectories `thread.c`, ..., `doc` do not. This limitation is imposed to simplify the implementation. It is not a problem in the uses we have for attributes, because we generally wish to treat each package version as a unit, not as a collection of files with individual properties.

The repository development cycle tools make extensive use of mutable attributes. For example, the **vcheckout** tool puts attributes on the stub, the session, and the working directory so that all three can be found given any one. The **vadvance** and **vcheckin** tools use these attributes and record some of their own. In addition, **vcheckout** and **vcheckin** record the previous version, the user requesting the operation, the time, and other information. Figure 4.5 shows some sample attributes applied by the tools. Note the `message` attribute on the directory `vesta/repos/30`; it is a change log message solicited from the user by **vcheckin**.

We also use mutable attributes to implement a limited form of symbolic link. A symbolic link is implemented as a stub with an attribute called `symlink-to` that gives the link value. The NFS interface manifests any stub with this attribute as a symbolic link, but to the evaluator, such stubs are just stubs, so they cannot be used in models and their mutability cannot compromise the repeatability of builds.

As a convenience feature, each appendable directory that contains versions has a symbolic link named `latest` that points to the latest version. For instance, in the example of Figure 4.1, `/vesta/west.vestasys.org/common/thread/latest` would be a symbolic link to `3`.

We expect to find more uses for attributes in the future. For example, a release-status attribute might be used to mark a particular version as internally released, externally released, or withdrawn from release due to bugs. The **vupdate** tool (Section 5.3.1), which mechanically updates the `import` statements in a Vesta model to newer versions, could be modified to take such an attribute into account.

Attributes are also used to store the access control information described in the next subsection.

4.1.7 Access Control

Designing the access control model for the repository presented some challenges, because replication (Section 4.4) and other forms of remote access are sometimes needed between repositories that are in different *realms*; that is, repositories that are under separate administration, have different spaces of user names, and perhaps

```
% vattrib /vesta/west.vestasys.org/vesta/repos
#owner
        mann@west.vestasys.org
type
        package
creation-time
        Thu Aug 22 17:41:35 PDT 1996
created-by
        heydon@west.vestasys.org

% vattrib /vesta/west.vestasys.org/vesta/repos/30
session-dir
        /vesta/west.vestasys.org/vesta/repos/checkout/30
old-version
        /vesta/west.vestasys.org/vesta/repos/29
message
        Added code to gather usage statistics.
content
        /vesta/west.vestasys.org/vesta/repos/checkout/30/12
checkin-time
        Tue Nov  4 14:10:22 PST 1997
checkin-by
        mann@west.vestasys.org
#owner
        mann@west.vestasys.org

% vattrib /vesta-work/mann/repos
session-ver-arc
        0
session-dir
        /vesta/west.vestasys.org/vesta/repos/checkout/31
old-version
        /vesta/west.vestasys.org/vesta/repos/30
new-version
        /vesta/west.vestasys.org/vesta/repos/31
checkout-time
        Wed Dec  3 09:55:12 PST 1997
checkout-by
        mann@west.vestasys.org
```

Figure 4.5: Sample attributes on some directories.

do not entirely trust one another. Thus we need a practical way of authenticating and access checking cross-realm requests. In addition, for repositories that are cooperating closely, we need it to be meaningful to replicate access control lists, so that each repository does not have to separately administer them. This section outlines how we have addressed these issues.

All access control lists and access checking in the repository are done in terms of global principal names, with the syntax `user@realm` or `ˆgroup@realm`. The realm is an arbitrary name chosen by a system administrator, typically an Internet domain name that makes the realm's user names valid email addresses. Group names are written with a leading caret only for clarity; there is not really any context in which both group names and user names can appear, so the caret is not needed to prevent ambiguity.

We have kept the repository's access control lists close to the Unix style so that they can be manifested fairly accurately through the NFS interface. Each object has an owner ACL and a group ACL, plus a set of nine mode flags that indicate whether the owner, group, and others are granted read, write, and/or directory search access. Unlike the Unix model, the owner and group are sets of global names, not single local names; we have provided this feature mainly so that an object can be given different owners in different realms if desired. Therefore, when choosing which owner and group to manifest through the NFS interface, the repository searches first for one in the local realm. The repository maps between global principal names and the numeric user/group ids used through the NFS interface by examining the local operating system's user and group registries (on Unix, `/etc/passwd` and `/etc/group`) and building up a translation table.

Access control lists and mode flags are stored in mutable attributes named `#owner`, `#group`, and `#mode`. Because not all objects have attributes, and to save space, we use a form of inheritance: if an object does not have a particular access control attribute, it inherits the value from its parent directory. Hence, changing the access control on a directory can effectively change the access control on other directories and files below it in the tree, a departure from conventional Unix semantics. The names of all access control attributes begin with an identifying character ("#"), and the replicator (Section 4.4) can be instructed not to copy them even if ordinary attributes are being copied.

Execute permission is treated specially. In Unix and NFS, a file's execute bits encode two logically distinct items—whether the file is executable, and whether particular principals have permission to execute it. Within a given directory, some files may be executable and some not, so it is not appropriate for files to inherit execute permission from their parent directories. But a file stored in an immutable directory does not have a `#mode` attribute of its own, so we cannot set its permissions individually. To work around this problem, the repository maintains an *executable flag* for every file. When the Unix chmod function is invoked on a mutable file through the repository's NFS interface, the file's executable flag is set to the logical *or* of the three executable bits given; for immutable files, the flag is immutable too, and chmod has no effect. We then use the executable flag and the read permission bits to determine the execute permission bits: if a file is executable, its three execute bits are a copy of its read bits; if not, they are all zero.

We also supply the Unix *setuid* and *setgid* flags. (The setuid flag tells the operating system that if this file is executed as a program, the resulting process should be granted the privileges of the file's owner; the setgid flag that it should be granted the privileges of the file's group.) These flags are of no interest to Vesta itself, but the repository is careful to maintain them with the proper security; for example, a file's setuid flag is automatically turned off if its owner is changed. The flags are encoded in an interesting way, designed to give the right security properties while considering the possibility of multiple and changing owners and groups. Both the #setuid and #setgid attributes are sets of principal names. If the first value of #owner that is in the local realm (the owner that will be manifested through the NFS interface) is also a value of #setuid, the setuid flag is set, otherwise not. Similarly, if the first value of #group that is in the local realm is also a value of #setgid, the setgid bit is set, otherwise not.

Each incoming request to the repository must be authenticated as coming from some particular user. Several authentication methods are supported. The repository administrator fills in a table (similar in style to an NFS export table) that specifies which user names to accept, from which hosts, using which authentication methods, and whether to grant normal or read-only access. Thus far we have implemented only two rather insecure authentication methods: the NFS AUTH_UNIX style, where the user provides his numeric Unix user id and is believed if he comes from a trusted host (needed to support most current NFS clients), and a similar style where the user supplies a global principal name and is believed if he comes from a host that is trusted for names from that realm. We have a design sketch for adding Kerberos authentication and hope to implement it in the future.

The repository recognizes three different administrative principals. The system administrator (Unix user root) has blanket permission to perform any operation, except operations that could cause a violation of the replica agreement invariant (Section 4.4). For convenience, there is also a non-root Vesta administrator (typically Unix user vadmin) that has the same permissions as root except for permission to modify the #setuid and #setgid flags; for the latter, vadmin is treated like an ordinary user. Thus, a designated non-root user can manage the Vesta repository, but cannot gain root access to other system resources or impersonate other users. There is also a special wizard user (typically Unix user name vwizard) that is permitted to do all operations, even those that could potentially violate the agreement invariant; this facility is provided only for emergency repairs and is not needed in normal operation.

Our general model for group access is that the user need only authenticate his user name; the repository determines what groups he is in. This model works well for intra-realm access, because the repository uses the local operating system's facilities to determine the group membership of local users. A user accessing a remote repository, however, by default will not be recognized as a member of any groups, even groups from his own realm. To address this problem, we allow repository administrators to augment the group membership table with additional entries, but this is a bit labor-intensive. A useful future addition might be to allow group membership tables to be replicated.

As an additional feature, one user name or group name can be designated as an alias for another; this is useful when the same person has a login in two different realms or when two cooperating realms each have a group that is working on the same project.

We omit the details of what access permissions are required to perform most operations, which are generally obvious. Operations on attributes are controlled as follows. Changing an attribute whose name begins with # generally requires ownership access, except for a few special cases such as `#owner` itself (which requires administrative access), `#setuid`, and `#setgid`. Changing other attributes requires write access, while reading attributes is unrestricted.

4.2 The Evaluator's View

This section describes some features of the repository that are provided specially for use by the Vesta evaluator and function cache. Most are not visible to ordinary users.

4.2.1 Derived Files and Shortids

As discussed in Chapter 3, running the Vesta evaluator on a model creates *derived files* such as object modules, libraries, and executable programs. Derived files are not entered into the source name space as they are created: by default, they have user-visible names only within the namespace of an evaluation, that is, within a running program in the Vesta system description language. As evaluation proceeds, the evaluator writes function cache entries that refer to deriveds, and when an evaluation concludes, the evaluator may ship (copy or link) some deriveds into persistent, user-visible directories as the evaluation's final output.

The repository provides raw storage for deriveds in the form of a simple abstraction called the *shortid file*. A shortid file is simply an ordinary disk file that is named by a 32-bit integer, or *shortid.* The repository provides an interface for allocating a new shortid and creating a corresponding file, and for opening an existing file given its shortid. Both ongoing evaluations and persistent function cache entries refer to deriveds by shortid.

When the Vesta weeder deletes a function cache entry, some or all of the deriveds referenced in the entry's result value may become garbage, so shortid files are managed using mark and sweep garbage collection. As the weeder walks over the cache, it assembles a complete list of shortids that are referenced by cache entries that are being retained. The weeder passes this list on to the repository, together with the time at which the list is valid. (A time just before the start of weeding is used, to guarantee that deriveds created by evaluations running in parallel with the weeder are not deleted; see Chapter 8.) The repository can delete any shortid file that is not on the list and whose time of last change (Unix *ctime*) is earlier than the list's validity time.

Internally, the repository stores source files in shortid files too. By definition a *source file* is simply a shortid file that has a name in the repository's source name space. As with deriveds, the repository manages sources using mark and sweep garbage collection, not reference counts. Whenever the weeder submits a list of shortids to keep, the repository augments the list by walking its own directory tree to find the sources it needs to keep. It then deletes only files that are not on the augmented list.

Sources and deriveds need to be managed uniformly because it is possible for a source to become a derived, and vice versa. A source becomes a derived if a function returns a source as part of its result; this happens quite often. Later, the original source's name might be deleted (that is, rebound to a ghost), but the shortid file that the source was stored in will be kept as long as a cache entry still refers to it. A derived becomes a source if an RPC client calls the repository and asks for the derived to be inserted (linked) into a source directory. We do not currently use this feature, but we have some future applications in mind. For example, the evaluator currently implements the shipping of final output deriveds from an evaluation by copying them into a conventional file system directory, but it could instead link them into a Vesta directory; this implementation would be faster and use less disk space.

The repository also assigns a shortid to every distinct immutable directory. (Two immutable directories that are identical except for their name and parent directory are not considered distinct; they usually have the same shortid.) This feature lets the evaluator and function cache refer to a whole tree of sources with one shortid, allowing a more compact, coarse-grained representation of dependencies on files in the tree. The use of directory shortids adds another step to the repository's garbage collection algorithm: for each directory shortid that the weeder asks it to keep, the repository must walk the subtree rooted at the corresponding directory and keep the subdirectories and files there as well.

4.2.2 Evaluator Directories and Volatile Directories

The repository provides two classes of directories, called *evaluator directories* and *volatile directories*, for use by the runtool server (Section 5.2.3). The runtool server encapsulates tools by running them in an environment where all their file references are restricted to such directories, enabling the Vesta system to capture and record all the references that the tools make and to supply appropriate, immutable contents for each referenced file.

An evaluator directory is a directory whose contents are defined by a *binding* that exists in the value space of an ongoing Vesta evaluation. A binding (Section 5.2.2) is a data structure in the Vesta language that pairs *names* with *values*, similar to a LISP association list or Perl associative array. An evaluator directory D reflects the contents of a binding B into the file system name space. Each name in the binding appears as a name in the directory. If the name N_1 in the binding B refers to a value V_1 of type Text (a byte string or file), then the name N_1 in the directory D refers to a file with V_1 as its contents. If the name N_2 in the binding B refers to another, subordinate binding V_2, then the name N_2 in the directory D refers to a subdirectory whose contents are defined by V_2, recursively. There is also a way to make I/O devices such as the Unix `/dev/null` appear in an evaluator directory. An evaluator directory is immutable, because it is created and used only while an evaluation is blocked inside a call to the language's `_run_tool` primitive.

When the repository receives an NFS request to list an evaluator directory or look up a name in it, the repository passes the request through to the evaluator via an RPC. When the evaluator receives such an RPC, it records a dependency on the given name and returns its value. The value can be either a shortid, representing a file, or a handle for another binding, representing another evaluator directory. The repository then generates the appropriate NFS reply. The repository keeps a cache for each evaluator directory to avoid repeated RPCs for the same name.

In addition to looking up files, an external tool may create new files or make changes to existing ones. Such changes cannot be made directly to the binding backing the corresponding evaluator directory, since that would amount to a side-effect. Instead, the repository records all such changes in volatile directories. At the end of the `_run_tool` call that launched the tool, the changes recorded in the volatile directory are reported back to the caller as part of the `_run_tool` result.

A volatile directory consists of a pointer to an evaluator directory, called its *base*, and a list of changes. Thus a volatile directory is analogous to a mutable directory. As in a mutable directory, one can create new files or (thanks to copy-on-write) edit existing files.[1] These directories are called “volatile” not for any fundamental reason, but merely because our repository implementation does not record them on disk, so they are lost if the repository crashes and restarts. Any shortid files created are of course recorded on disk; they are later garbage collected, if necessary. This volatility is tolerable because a volatile directory needs to exist only as long as it takes a single tool to run, so the only negative effect is that on the rare occasions when the repository crashes, any ongoing evaluations fail. In compensation, the task of reclaiming resources after a crash is simplified.

Running a tool then works as follows. An evaluation invokes the `_run_tool` primitive with the arguments described in Section 5.2.3, including the binding that is to supply the initial contents of the tool’s root directory. The evaluator assigns a directory handle to this binding and calls the repository to create a new volatile directory based on it. The evaluator then invokes the runtool server, which starts the tool with the root directory name “`/`” rebound to the new volatile directory. (On Unix, this step uses the `chroot` system call.) After the tool finishes running, the evaluator calls the repository to find out what changes the tool made to its volatile directory, and uses this information to construct a result binding. If the tool created or edited any files, their new shortids are returned as part of the change list, and they become deriveds. Finally, the evaluator deletes the volatile directory, freeing the resources it was consuming in the repository.

4.2.3 Fingerprints

Within the function cache, special 128-bit checksums called *fingerprints* are used to provide compact, unique abbreviations for values. Two values can be compared for equality by comparing their fingerprints, with a vanishingly small probability of erroneously considering two different values to be equal. See Section 6.2 for further details.

As a service to the function cache and evaluator, the repository keeps fingerprints for certain kinds of files and directories and makes them available though an RPC interface.

Every immutable directory and immutable file in the tree rooted at `/vesta` has a fingerprint, because an evaluation can refer to such files and directories as sources. Ghosts, stubs, and appendable directories do not have fingerprints, because a (successful) evaluation can never refer to one. The repository also supplies fingerprints for files in volatile and evaluator directories, because an evaluation can

[1] But see Section 4.3.3 for restrictions.

produce and cache such a file as a derived. Volatile and evaluator directories themselves do not require fingerprints; their existence is ephemeral and the function cache never contains a reference to one. For performance reasons, we use several different methods of fingerprinting files and directories.

Each large immutable file and each immutable directory is given a fingerprint based on the full hierarchical name under which it is first placed into the repository. That fingerprint stays with the source as it acquires new names via the checkin-advance-checkout cycle or via renaming. Fingerprinting based on the name provides the required semantics because the repository guarantees that a name is never reused for a different source.

Each large new derived (that is, each large new file in a volatile directory) is given a fingerprint based on an arbitrary unique identifier that is generated when the file is created. That fingerprint is reported back to the evaluator, stored with any function cache entry that points to the file, and supplied again to the repository if the file later appears as an existing file in a new evaluator directory. Fingerprinting based on a unique identifier provides the required semantics because the identifiers are never reused. It gives better performance than fingerprinting the file contents because the identifiers are much shorter than the average large file.

The fingerprint of a small file is computed from its contents. This method of fingerprinting has some interesting consequences. For example, if a user "touches" a source file in a working directory (that is, performs a write to the file that does not actually change the contents), source fingerprinting based on a unique identifier would cause the file to be recompiled and everything that depends on it to be rebuilt. With fingerprinting based on contents, however, the file will be recognized as unchanged and nothing will be rebuilt. Further, suppose a user edits the comments in a source file but does not change the code. This always causes the file to be recompiled, but if the compiler is fully deterministic, the recompilation will generate a derived file with exactly the same contents as the previous version. With derived fingerprints based on contents, the new derived file would be recognized as identical to the old one, and programs that use it would not be relinked.

Because the two types of fingerprinting give semantically equivalent results, we have made the size threshold configurable at runtime. By default, we currently fingerprint both sources and deriveds that are smaller than one megabyte by contents, larger ones by unique identifier.[2] This approach works well for sources, but unfortunately, the C and C++ compilers that we are currently using insert timestamps into the object files they produce, negating the benefits of fingerprinting these files by contents.

[2] On the hardware described in Chapter 9, a file can be fingerprinted at roughly 1 MB/sec (elapsed time).

Note that changing the size threshold between the fingerprinting methods does not cause a problem. The cache and evaluator require only that two files or directories with different contents always have different fingerprints. It is not essential that two files or directories with the same contents always have the same fingerprint, though it is advantageous that they do as often as possible, since that will allow for more function cache hits. (We of course arrange that fingerprints computed by the two methods do not collide, by including a prefix in the fingerprinted string that says which type of fingerprinting was used.)

A late addition to the repository design was an internal table that allows any file or directory to be looked up by its fingerprint. This table has two important uses. First, if a user advances a small file into the repository that is already present with the same contents but a different name, the repository looks up the new file's fingerprint in the table, finds that a copy is already present, and arranges for the two copies to share storage, thus saving disk space. Second, the replicator (Section 4.4.8) uses the table to avoid making redundant copies of both files and directories when copying data from one repository to another, and to avoid copying a directory in full if a copy of its base (Section 4.3.2) is already present at the destination. The table is not kept on disk, as the repository is able to rebuild it from its other data structures upon reboot.

4.3 The Implementor's View

This section describes some interesting aspects of the repository implementation.

4.3.1 Shortids and Files

The repository stores shortid files in an ordinary file system provided by the underlying operating system, under a fixed directory chosen when Vesta is installed. Each file's name is derived from its 32-bit shortid. For example, a file with shortid 0x12345678 would have a name like `/vesta-sid/123/456/78`. Intermediate directories such as `/vesta-sid/123` and `/vesta-sid/123/456` are created only when needed and are deleted when they become empty. Vesta users never see these filenames. The `/vesta-sid` directory and all files and directories beneath it have their access permissions set so as to be directly accessible only to the repository server, the function cache server, and the evaluator. Immutable shortid files (those corresponding to immutable source files or to deriveds that have been completely written and are referenced by cache entries) have their write permission bits turned off.

Why do processes other than the repository server itself have direct access to the /vesta-sid directory tree? We allow this for efficiency. A process that knows a file's shortid can read or write it directly though the underlying file system, thereby avoiding the overhead of passing data through the repository NFS interface and offloading work from the repository server. We also permit processes to create shortid files without contacting the repository for each one. The repository provides an RPC interface that allocates new shortids in blocks of 256. The blocks have leases; that is, ownership of a block times out if it is not periodically renewed, enabling the repository to reclaim blocks allocated to processes that have crashed. A layer of library code provided by the repository implementation hides the complexity of block allocation and lease renewal from client programs.

This efficient route for accessing shortid files unfortunately sees little use. Nearly all accesses to shortid files are through the repository NFS interface. Users of course access sources through NFS. Encapsulated tools read existing sources and deriveds, and write new deriveds, using the repository's NFS interface to volatile directories. Only a few accesses by the evaluator itself and by the weeder bypass the repository NFS server. In hindsight it might have been a better design choice to omit this access path, instead making the /vesta-sid directory tree a private data structure that is hidden from all processes but the repository server itself.

To conserve memory, the repository avoids keeping any sort of in-memory structure indexed by file shortid. It keeps a record only of which 256-shortid blocks are currently leased. When some process holds a lease on a block, that process maintains a bitmap of unused shortids in the block. The initial value for this bitmap is computed when the block is allocated, by looking at the /vesta-sid directory tree and seeing which shortids in the block are currently bound to files. When allocating new blocks, the repository tries to choose empty ones, but does not guarantee to do so every time.

The repository does need to keep an index that maps from immutable directory shortids to the actual directory structure in memory. The index is stored as a hash table in memory. The index is little used, so there should be no serious performance problem if it falls out of the repository's working set and needs to be paged back in from disk when used. It might have been preferable to eliminate this index by computing a directory's shortid from its memory address, but this was impractical because the repository's memory management (discussed next) sometimes moves directories to different addresses. At any rate, the index does not take up much space; see Section 9.3.2.

4.3.2 Directories

The repository keeps all of its directory structure in virtual memory. This includes all five directory types described above (appendable, immutable, mutable, evaluator, and volatile), plus stubs and ghosts. File data is stored in shortid files; directory entries for files point to them by shortid.

While the directory structure externally appears to be a tree, internally it is a DAG (directed acyclic graph). Much structure that appears to be repeated in the tree occurs only once in the DAG, thereby saving a great deal of memory. Internally, directories do not have parent links. Therefore, when an immutable directory with the same children appears at two or more different places in the tree, internally there need be only one copy. This sort of sharing occurs whenever a package has a subdirectory that remains unchanged from one package version to the next. Also, as sketched earlier, every directory is implemented as a list of entries together with a (possibly null) base pointer. Therefore, when one directory is nearly the same as another, the second can be represented compactly as a list of changes relative to the first. This sort of sharing occurs between each version of a package and the previous version. Currently, there is no limit on how long a chain of base pointers can grow, but it would be straightforward to add a heuristic that breaks the chain if it becomes so long that lookups are too slow.

The repository packs its in-memory structure tightly to keep memory consumption down. The goal is to keep the structure small enough to stay resident in physical memory at all times, so that directory access will not be slowed by paging. To this end, references between parts of the structure use 32-bit array indices instead of pointers (which are 64 bits wide on Alpha, the architecture on which Vesta was first implemented), and record fields are arbitrarily byte-aligned. We demonstrate the effectiveness of this compaction in Section 9.3.2.

To keep the directory data stable, we use a simple logging and checkpointing technique [7]. Whenever a repository client requests an update operation, the server appends a record of the operation's name and parameters to a log file and forces it to disk before modifying the in-memory data and returning to the client. If the repository crashes, upon restart it replays the log to restore its state. Operations on volatile directories are not logged.

To make recovery faster, the repository occasionally makes a *checkpoint* by dumping its state to a file. The next recovery then begins from the most recently committed checkpoint. The checkpoint code is actually an application-specific compacting garbage collector, so checkpointing has the useful side-effect of reducing memory fragmentation. The algorithm is designed for minimal memory usage. First, it writes the compacted checkpoint directly to a file, not to a second ("to-space") memory region as an ordinary copying garbage collector would. Second, when it copies an object, it puts the forwarding pointer to the object's new address (which is needed to keep the object from being copied more than once if there are many pointers to it) into the first few bytes of the object's old location. Thus the algorithm is destructive, necessitating an immediate recovery from the checkpoint file when it is complete.[3] This recovery is transparent to the rest of the system; we make it so by including even volatile directory structures in the checkpoint. The volatile structures are written at the end of the checkpoint and are ignored when recovering from a real crash.

Our logging package is sufficiently general-purpose that we use it in the function cache server as well as the repository. It permits an arbitrary number of bytes to be atomically appended to a log, even if the underlying file system and disk controller hardware reorder writes to disk, by including a version number in every disk block. The log packs data tightly into disk blocks, yet ensures that committed data is not lost even if a hardware failure corrupts the block currently being written. It achieves this by alternately writing two different blocks when adding more data at the tail of the log, like the ping-pong algorithm [21] but avoiding the need to sometimes do the last write twice. The package supports fuzzy checkpointing; that is, making a checkpoint in parallel with appending more data. The repository server does not require this feature, but the cache server uses it to enable weeding to proceed in parallel with normal operation.

4.3.3 Longids

To operate as an NFS server, the repository must assign a 32-byte *NFS file handle* to every file and directory it stores, and it must be able to look up a file or directory by its handle. For proper NFS semantics, the meaning of a handle must remain stable across repository crashes and restarts, and a handle for a deleted object must not be reused (at least not soon). To keep from using too much memory, we want to avoid having a large table that maps from handles to memory addresses, yet we cannot use memory addresses directly as handles because checkpointing moves directories to different addresses.

Another problem the repository faces is how to implement the parent links in a Unix-like directory tree when the internal DAG representation does not store such links. Indeed, as mentioned earlier, several externally visible and apparently distinct directories with different parents may share the same internal representation.

We solve both these problems with one novel mechanism, the *longid.* A longid encodes the path through the externally visible directory tree that was used to reach an object. Each component of the pathname is encoded as an index number. The low-order bit of this index number indicates whether the entry is in the mutable change list of the directory or in the immutable base, while the other bits give the entry number within the indicated list. The index numbers are represented in a variable-width encoding (where numbers 0 to $2^7 - 1$ require one byte, 2^7 to $2^{14} - 1$

[3] If the repository crashes while writing a checkpoint, it will recover from the most recent successful checkpoint and the succeeding operation log.

require two, etc.), and the resulting bytes are concatenated to form the longid. We reserve 0 as a terminator. Longids are restricted to 32 bytes in length so that they can be used directly as NFS handles. Restricting the longid length of course restricts the depth of the directory tree, but 32 bytes seems ample for any size source pool we can imagine storing.[4] Longids are repository-specific, not global; replication does not transmit them from one repository to another.

Longids have the major properties we need. Every object in the externally visible directory tree has its own longid, which remains stable across repository restarts. Longids are not reused, because directory entries are not reused or deleted. (In directory types that permit deletion, we keep an invisible placeholder for each deleted entry to prevent its index number from being reused.) The repository looks up a longid by traversing the directory tree much as if it were looking up the corresponding name; this traversal is fast because the entire tree is kept in memory. Given an object's longid, one can find its parent's longid simply by truncating the last component.

Longids do not quite provide a perfect implementation of the expected NFS semantics, but we have been able to paper over the difference effectively. In particular, if an object is renamed, its NFS handle is expected to remain the same. In the repository, an object that is renamed unavoidably gets a new longid, but we replace its old directory entry with a forwarding pointer to the new one so that its old longid can continue to work also. The old longid stops working if the object's old parent directory is deleted, however. Also, the existence of two handles for the same mutable file will cause an NFS cache coherence problem in the extremely unlikely event that the same client has the file open twice, once under the old handle from before it was renamed and once under the new name and new handle. Even though both opens are done by the same client and thus would normally be coherent, in this case the client sees two different handles, so it will cache the file twice and fail to keep the two copies coherent. We have never seen such a case in practice.

Longids as described so far have one major drawback. When the evaluator creates a new volatile directory tree to provide an encapsulated environment for a tool, all the files and directories in the tree acquire new, unique longids. But when the evaluator runs several tools in succession in the course of a build, many shortid files are accessed repeatedly; for example, standard C header files that are read by many compiles, or object files that are written by a compile and read by a subsequent link. For good performance, tools should get NFS client cache hits when they access files that other tools have recently accessed, but this is impossible when the same file is seen as having a different file handle each time a new tool is

[4] In our own usage thus far, we have not exceeded 13 bytes.

run. We had substantial performance problems in large Vesta builds until we found a solution to this problem.

Our solution is as follows. Files in volatile directories are optionally given a *shortid-based longid*, a new kind of longid that encodes the file's shortid and fingerprint instead of its pathname. Thus, the same shortid file has the same file handle every time any tool encounters it, and the tools see good NFS cache performance. All directories continue to use standard pathname-based longids.

Shortid-based longids have two drawbacks that keep them from universally displacing pathname-based longids.

First, a level of indirection is lost, making copy-on-write impossible. That is, with a pathname-based longid, there are two levels of indirection: the longid specifies a path through the directory tree, and the tree specifies a shortid. Therefore we can change the shortid that a longid points to by changing the tree; in particular, we can do copy-on write by changing the tree from specifying an immutable shortid file to specifying a new, mutable copy of the same file with a different shortid. But a shortid-based longid specifies the shortid directly, so its meaning cannot be changed this way. This makes shortid-based longids unsuitable for files in mutable directories, so we do not use them there. It also means that tools cannot be allowed to modify existing files in volatile directories, as the original design permitted (Section 4.2.2), because that also requires copy-on-write. This limitation is no problem for most tools, but to accommodate unusual tools that need to be able to modify existing files, we have made the old behavior selectable by a flag to the evaluator's `_run_tool` primitive (which is in turn passed to the repository's volatile directory creation primitive).

Second, the ability to find a source object's parent directory is lost. This makes shortid-based longids unsuitable for directories. It also means that a file's access control cannot be inherited from its parent directory, which makes the new longids poorly suited for files in immutable and appendable directories. One could imagine living with the access control limitation, by making all immutable files world-readable and relying on directory access controls to protect them when necessary, but this option is unattractive.

Fortunately, pathname-based longids provide adequate NFS cache performance in mutable, immutable, and appendable directories, because such directories are not created and deleted frequently, so retaining them there has caused us no problems.

4.3.4 Copy on Write

As explained earlier, the repository uses copy-on-write to save disk space. When a user creates a mutable directory based on an immutable one, all the file entries in the new directory initially refer to immutable files. If the user tries to write to such a file, the repository copies it to a new file with a new shortid, adds an entry to the “changes” portion of the mutable directory to point to it, and writes to the new file instead.

Pathname-based longids add an extra complication here. In the situation described in the previous paragraph, the new entry has a different index number than the old one, so although the new copy of the file has the same name as the old one, it has a different longid! We fix this problem by setting a flag in the new directory entry to indicate that the old longid should continue to be used, not the new one. When the repository looks up a name to find a longid, if it encounters an entry with this flag set, it looks for another entry containing the same name in the directory’s immutable base and uses that entry’s index number in the longid. When the repository looks up an object by longid, each time it encounters an index number that points into the immutable base of a mutable directory, it extracts the name from the directory entry found and checks to see if there is a flagged entry in the “changes” portion with the same name. This solution uses minimal space but does slow down name and longid lookup a bit.

We also do copy-on-write for directories. A new mutable directory may be based on an immutable directory with immutable subdirectories. If a user tries to edit a file (or make any other change) in such a subdirectory, the repository copies the old immutable subdirectory to a new mutable one, and adds an entry to the mutable parent directory to point to it. In this case, of course, the copying itself is optimized by creating the copy as a new directory based on the old one with an initially empty list of changes. If the user makes his first edit several levels deep in the directory structure, the copying process is applied recursively.

4.3.5 NFS Interface

The repository NFS server runs entirely in user space. It is simply a software layer on top of the basic repository functionality, which is (as we have described) layered on top of an ordinary file system. The layered, user-space approach makes for simpler implementation and debugging than a kernel-resident approach, but it incurs additional overhead in data copying and context switching. We show in Chapter 9 that although the repository provides poorer file system performance than a standard kernel-resident NFS server, it is still fully adequate for our purposes.

The NFS server implementation uses a modified version of Sun’s ONC (Open Network Computing) RPC library. The original library was designed for use only in single-threaded programs; in particular, its server-side duplicate suppression machinery assumes there can be only one outstanding request at a time. But because NFS is built on a simple request/response protocol with no data streaming, NFS implementations generally perform badly unless many NFS reads and/or writes can be in flight simultaneously between the same client and server in separate threads. We rewrote the duplicate suppression machinery to enable multithreading, thereby removing this performance bottleneck.

The repository cheats slightly in its implementation of NFS version 2 semantics. For strict correctness, when a client writes to an NFS2 server, the server must make sure the data is stable (either on disk or in nonvolatile memory) before acknowledging the write to the client. Otherwise, if the server should crash and restart with some data acknowledged but not actually on disk, the client's cache would become incoherent with the server's state and the data would never be written. The repository does not implement these semantics; it passes writes on to the underlying operating system before acknowledging them to the client, but it does not wait for the operating system to force them out to disk. Therefore if the machine that the repository server is running on (not just the repository process itself) crashes while a client is actively writing to it, a write that the client believes has been done may actually be lost.

We have chosen to leave this problem uncorrected because it rarely occurs in practice, typically has little impact, and would be quite difficult to fix. The best long-term fix would be for us to upgrade the repository's NFS implementation to NFS version 3, but this protocol is much more complex than NFS version 2. A simple fix within the NFS2 framework would be to make a Unix `fsync` system call to force each write to disk before acknowledging it back to the client, but this change would significantly harm the repository's NFS performance.

4.3.6 RPC Interfaces

The repository has two additional RPC interfaces, one for access to shortid files and one for access to the directory structure. Most of the interesting features available through these interfaces have been discussed above, so we do not describe them further here. Both interfaces use the SRPC (simple RPC) package described briefly in Section 9.6.

4.4 Replication

Increasingly, large software systems are developed in parallel at geographically distributed sites. The Vesta repository was therefore designed to make it easy to replicate sources at many sites. To enable developers to work independently, the replication design allows each repository to operate mostly autonomously, thereby reducing the overhead of normal repository operations.

Our concept of replication is broad, encompassing everything from source distributions issued on CD-ROM to cooperative development of one source pool across many geographically separated sites. We begin by presenting the overall concept and explaining how it has affected the design of source naming, mutable attributes, and access control. We then go on to describe the replication tools we have written.

4.4.1 Global Namespace

Conceptually, Vesta sources are named in a single namespace that is global across all Vesta repositories. As described earlier, the namespace is organized as a tree. Each repository stores a subtree of the total namespace. Replication exists when two or more repositories store subtrees that overlap. In this case, we require that the overlapping portions *agree*; that is, (1) the same name is not bound to different values in two different repositories, and (2) at most one repository is *master* for each name. We define mastership and agreement precisely in the next section. We often use the term *partial replication* for our style of replication, because each repository can replicate all, part, or none of the data stored in any other repository.

We have chosen the standard Vesta naming convention to make it easy for new sites to adopt Vesta without inadvertently creating source names that clash with those at other sites. Thus, two sites that initially know nothing of each other and share no sources can still be considered participants in the global namespace, and can later decide to cooperate and take replicas of each other's code. In the standard naming convention, the root of the global namespace is called `/vesta`. We chose this name simply to make it easy to mount the repository into a standard Unix file namespace. When a new Vesta site wants to create sources that are initially not shared with any other site, by convention the site administrator puts them in a new directory immediately below `/vesta`, named with an Internet domain name that the site owns. In this way, the new names are made globally unique without the need for any special coordination. This mechanism is not perfect, because Internet domain names can be deregistered and later reregistered to some other owner, but we believe it is adequate for practical use.

Sources that are to be distributed widely should be named carefully, so that the names make sense to the people who will be using them. For example, we plan to make the Vesta sources publicly available under a directory named `/vesta/vestasys.org`, not one named for some particular machine.

4.4.2 Mastership

Every source object in a Vesta repository has a boolean *master* flag, including files, directories, stubs, and ghosts. When an object is replicated in several repositories, its master flag is set in at most one of them. Mastership for an object can be transferred from one repository to another; this must be done using a carefully designed protocol, to ensure that a failed transfer attempt still leaves the object with exactly one master. We describe mastership transfer in Section 4.4.5 below.

Mastership is important chiefly for appendable directories. The master copy of an appendable directory is the synchronization point for adding new names to the directory. Arbitrary new names can be freely added to a master appendable directory, but new names can be added to a nonmaster appendable directory only by copying them from another repository. When an appendable directory is mastered at a particular repository, we do not require that repository to store a complete copy of the subtree rooted at that directory. Nevertheless, the master repository needs to keep a complete list of bound names to prevent clashes when new names are inserted. To support this, we add another use for the stubs and ghosts introduced in Section 4.1.1 above: the master copy may bind some names to placeholder stubs or ghosts while other copies store the real data. Note that an immutable directory cannot contain a stub or ghost. Thus at each repository, every tree rooted at an immutable directory is either completely present or completely absent. In terms of packages and versions, if any file or directory from a package version is replicated in a given repository, then that entire version must be replicated there.

Mastership is important for stubs as well. A master stub can be freely replaced with a newly created source of any type, while a nonmaster stub can be replaced only by copying from another repository. In either case, the new source has the same mastership status as the old stub. The reservation stubs introduced in Section 4.1.4 above are master stubs. Thus, a master stub is a placeholder for data that has yet to be created, while a nonmaster stub is a placeholder for data that may exist in another repository but is not replicated locally.

We also make a distinction between master and nonmaster ghosts. Both types of ghosts indicate that a previously existing source has been deleted; thus they serve a placeholder role similar to that of stubs. We allow either type of ghost to be replaced by a copy of the source taken from another replica, with the new source retaining the mastership status of the old ghost, except that a master ghost cannot be changed to a master appendable directory or master stub. It is impermissible to change a master ghost to a master appendable directory because we cannot guarantee to restore all the names that were bound in the directory at the time it was deleted. It is impermissible to change a master ghost to a master stub because the master stub could in turn be replaced by an arbitrary object different from the name's original, pre-ghost value, thus violating Vesta's immutability guarantee.[5]

For other types of objects, the rule that a name has at most one master remains, but mastership has no other enforced meaning. By convention, however, the master copy of an object can be thought of as the "main" copy, which should not be deleted (or replaced with a ghost) without thinking twice.

4.4.3 Replication Example

Figure 4.6 shows an example of two repositories that partially replicate each other and are in agreement (consistent). The figure illustrates several common patterns that occur in real Vesta usage. On the left is the west coast repository of the imaginary Vesta Systems Organization; on the right is its east coast repository.

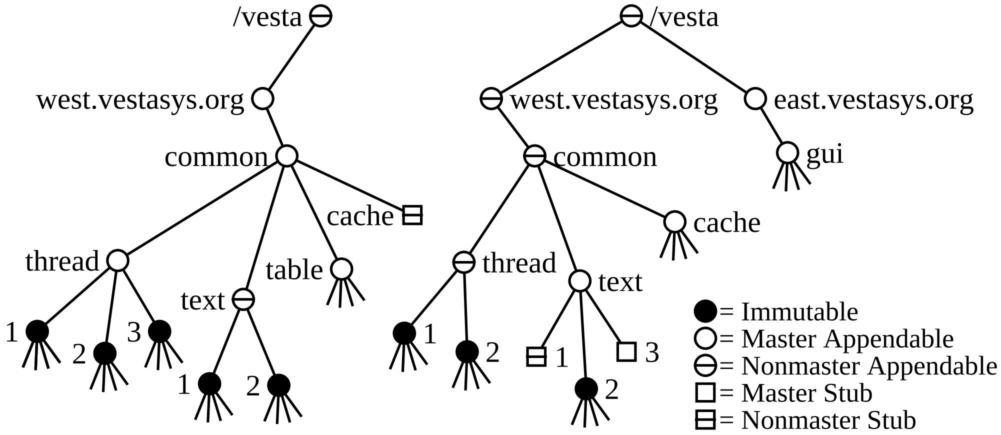


Figure 4.6: Two repositories (western and eastern) that are in agreement.

As discussed above, the root directory `/vesta` is not mastered at either repository, and the names directly under it look like Internet domain names. The western repository has created a subdirectory named `west.vestasys.org` and holds its master copy, and the eastern repository has done the same for `east.vestasys.org`.

Partial replication occurs at several levels of the tree. At the top, part of `west.vestasys.org/common` is replicated in the eastern repository. The western copy has a complete list of names (`thread`, `text`, `table`, and `cache`), while the eastern copy is lacking `table`. The western copy does not have the contents

[5] In hindsight, we could have simplified the design by eliminating both master and nonmaster ghosts, using nonmaster stubs for deleted items instead.

of the cache directory, but does have a stub with that name as a placeholder; thus no one can create a different cache directory that would clash with the copy in the eastern repository.

One level down, in the thread package, the western copy is master and has all three versions that currently exist, while the eastern copy does not currently have version 3. The text package illustrates the point that a directory need not have the same master as its parent; it is mastered at the eastern repository. Perhaps when first created it was mastered at the western repository and later moved to the eastern repository, since version 1 is present in the west but not in the east. Since the eastern copy is master, it must have a complete list of names, so it has a stub for version 1, perhaps inserted at the time it received mastership. In addition, the eastern copy has a master stub for version 3. This master stub is a placeholder for an object whose content has not yet been supplied; the master repository is free to replace it later with a different type of object, but thereafter it cannot be changed back to a master stub. Master stubs are used by **vcheckout** to represent reservations, as described in Section 4.1.4 above.

4.4.4 Agreement

We are now ready to give the precise definition of Vesta *agreement*. Let A and B be Vesta source objects, let $A.master$ denote the master flag of A, let $A.repos$ denote the repository where A is stored, and if A is a directory, let $A.names$ denote the list of names that are bound in it. Then $A \simeq B$ (read "A agrees with B") if the following recursively defined conditions hold:

1. $A.master \wedge B.master \Rightarrow A.repos = B.repos$ and

2. At least one of the following holds:

    (a) A and B are files with identical contents.

    (b) A and B are immutable directories where

    i. $A.names = B.names$, and

    ii. $\forall n : n \in A.names \Rightarrow A/n \simeq B/n$,

    (c) A and B are appendable directories where

    i. $\forall n : n \in A.names \wedge n \in B.names \Rightarrow A/n \simeq B/n$,

    ii. $A.master \Rightarrow B.names \subseteq A.names$, and

    iii. $B.master \Rightarrow A.names \subseteq B.names$.

    (d) A and B are both master stubs.

    (e) A or B is a nonmaster stub.

(f) A or B is a ghost.

In addition, we say two *repositories* agree when their replicas of the root directory `/vesta` agree.

Condition 1 effectively says that the same source is not mastered in more than one repository. It is stated in an odd-sounding way so that agreement can be reflexive (that is, so that a repository agrees with itself). Condition 2d is also needed only for reflexivity. Conditions 2a and 2b require replicas of immutable files and directories to be identical.

Conditions 2c and 2e make partial replication possible. By 2c, two appendable directories can agree even if one or both have only a subset of the complete set of names defined in the directory. But the master replica has a complete list of names; thus, the master can coordinate the creation of new names, assuring that the same name is never bound in different replicas to sources that do not agree. By 2e two appendable directories can agree even if one has a nonmaster stub where the other has some other object.

Conditions 2d–2f reflect the way stubs and ghosts are intended to be used, as described above. A master stub agrees with nothing but itself or a nonmaster stub, because a master stub represents a source that is to be checked in later. If the master stub is still present, the actual source has not yet been checked in, so it cannot exist in a different repository. A nonmaster stub, however, agrees with anything. A ghost also agrees with anything, because an object can be deleted from one repository but remain present in others.

Notice that the agreement relation is not transitive; pairwise agreement between A and B and between B and C is not sufficient to guarantee agreement between A and C. This nontransitivity is an unavoidable property of partial replication. Replicas are considered to agree when their overlapping portions do not clash; but A and C may overlap and clash in a portion that does not overlap with B. For example, suppose that `/vesta/foo` is mastered at repository A, and that `/vesta/foo/bar` is an immutable directory in repository A, absent in repository B, and an immutable file in repository C. Then $A \simeq B$ and $B \simeq C$, but the directory at A clashes with the file at C, so A does not agree with C.

Lest this example create a wrong impression, we must stress that Vesta replication is not intended to operate in a mode where some pairs of repositories agree and others do not. We want to establish and maintain *global agreement*: the invariant that *all pairs* of repositories agree.

We preserve global agreement by allowing a repository to change only through the application of specific operations that are known to be safe. For an operation that modifies only one repository A, it is sufficient to show that for all repositories B, if initially $A \simeq B$ holds, then it still holds after the operation. For an operation that modifies two repositories A and B, it is sufficient to show that for all repositories C, if initially $A \simeq B \wedge B \simeq C \wedge C \simeq A$ holds, then it still holds after the operation. Due to nontransitivity, all three terms in the postcondition must be proved, and all three in the precondition must be given.

4.4.5 Agreement-Preserving Primitives

It is easy to establish initial agreement among repositories, since a new repository that contains only an empty copy of the root directory `/vesta` agrees with every other repository. Thereafter, we need to know how to make changes to agreeing repositories in a safe way, one that is guaranteed not to break the agreement. We want repositories to operate mostly autonomously, so it is important that most operations can be performed without consulting another repository, and that the rest require consulting only one other. Our definition of agreement is designed to make this fairly easy. The following operations are safe, and the repository server provides each one as a primitive.[6] All the primitives are atomic except for number 7, mastership transfer.

1. Create a new master appendable directory in `/vesta`, using a locally owned Internet domain name.

2. Create a new child object of any immutable or appendable type in a master appendable directory.

3. Replace a master stub with a new immutable object.

4. Replace any child of an appendable directory with a ghost that has the same mastership status as the old child.[7]

5. Copy any child into an appendable directory from another repository, possibly replacing an existing ghost or nonmaster stub. If the original is an appendable directory, the copy is an empty nonmaster appendable directory; if desired, its children can be copied by further applications of the primitive. If the original is immutable, however, it is copied in full, including all its descendants.

6. Create a nonmaster stub in an appendable directory, if another repository has that name defined.

[6] We do not present a proof of safety, but the operations are generally simple and it should be easy for the reader to convince himself that each one is safe.

[7] Several variants of this primitive are also safe, but we currently use this one. Safe alternatives include using a nonmaster stub in place of a ghost, and/or completely removing the child if the parent is not master.

7. Transfer mastership on an object from one repository to another, at the same time adding stubs to the new master for any missing children of the old master.

Internally, the repository builds the more complex primitives on top of a set of simpler primitives for adding and replacing single objects, using a built-in feature that allows for short atomic transactions within a single repository. Primitives 1–4 run at a single repository.

Primitives 5 and 6 require consulting another repository, but a multi-site atomic transaction is not required; it is sufficient to read the data from the source repository, then atomically insert a copy into the destination. No lock on the source repository is needed while reading the original, since it cannot change; at worst, it can be replaced with a stub while the read is in progress, but this simply causes the primitive to return an error without changing the destination.

As mentioned at the end of Section 4.2.3, the destination repository optimizes the copying process in primitive 5 to avoid making redundant copies of objects, such as multiple objects that have the same content but different names. Each repository keeps an in-memory table in which each locally stored immutable file and immutable directory tree can be looked up by its fingerprint. When an object is to be copied, the destination repository first looks up its fingerprint in the table to find whether a copy is already present; if so, the repository links the existing copy into its name space instead of making another. In addition, if a directory being copied is encoded in the source repository as a list of changes relative to a base directory (Section 4.3.2), and the destination repository already has a copy of the base directory, then the destination encodes the copy in the same way.

Primitive 7, mastership transfer, is the most complex. Our goals in choosing an implementation were to guarantee that agreement could not be violated, to avoid blocking either repository during the transfer protocol, to minimize the likelihood of a failure resulting in neither repository being master, and to keep a hint on each nonmaster object as to where its master repository is located.

In barest outline, our implementation consists of two separate atomic operations. First, the repository ceding mastership on an object makes a complete list of its children and turns off its master flag; second, the repository acquiring mastership inserts any missing children into its copy as nonmaster stubs and turns on the master flag. To meet our goals, the full implementation also takes care of updating the master location hints, and it keeps a stable record of in-progress transfers at both repositories, persistently retrying them until they are complete. With this implementation, agreement cannot be violated, and the object can be left without a master only if one repository crashes permanently or the network link between them is permanently severed while a transfer is in progress.

In more detail, our implementation works as follows. Steps carried out by the repository trying to acquire mastership are numbered starting with A1; steps carried out by the repository ceding mastership are numbered starting with C1 and are indented. Request/grant identifiers and master location hints are stored in mutable attributes.

A1. Check that the requesting user has the necessary access permissions and that the current master repository can be reached over the network; quit if not.

A2. Choose a unique request identifier and record it on the local copy of the object.

A3. Ask the current master to cede mastership, supplying the request identifier.

> Do steps C1–C4 atomically:
>
> C1. Check that the requesting user has the necessary access permissions; refuse to cede mastership if not.
>
> C2. Form a grant identifier. If the object is an appendable directory, do this by appending a list of its children to the request identifier; otherwise use the request identifier. Record it on the local copy of the object.
>
> C3. Change the object's type from master to nonmaster, and record the new master repository's location on it. This location is of course only a hint, since mastership could move to yet another repository later.
>
> C4. Return the grant identifier to the caller.

A4. If the current master refused to cede, erase the request identifier and quit.

A5. Atomically fill in any missing children listed in the grant identifier (creating them as nonmaster stubs), change the object's type from nonmaster to master, record this repository in the object's master location hint, and record the grant identifier in place of the request identifier.

A6. Ask the old master to erase the grant identifier.

> C5. Erase the grant identifier.

A7. Erase the grant identifier.

The repository that is trying to acquire mastership tries persistently to complete these steps, even if it crashes and restarts during the transfer, until step A7 is finished. Thus mastership will not be lost unless one of the repositories permanently fails (or the network is permanently severed) between steps C4 and A5, and even in this case there will be a record of the incomplete transfer in whichever repository remains up.

4.4.6 Propagating Attributes

The definition of agreement says nothing about mutable attributes; two replicas of a source may agree but have entirely different attributes. Each repository can change the attributes of both master and nonmaster sources, and there is no requirement to propagate attribute changes to other repositories. But in many cases such propagation is desirable, so we have included a feature in the attribute facility to support it.

Section 4.1.6 described attributes as a total function F from names to sets of values, but this is only the user's view. At a lower level of abstraction, an object's attributes are recorded as a history of state changes H, represented as a set of timestamped tuples. Each of the four write operations (set, clear, add, and remove) takes a timestamp argument, which can be any value but defaults to the time at which the operation was requested. Applying an operation inserts a new tuple into H consisting of the name of the operation and its arguments. $F(H)$ is then computed whenever needed by sorting H into timestamp order (with ties broken by taking the operation, name, and value as secondary sort keys), and applying the resulting sequence of operations to the empty function.

In addition to the high-level operations that query F, we also provide a low-level operation to query H. This operation does not necessarily return H itself. Instead, it returns a history K that is *equivalent* to H, in the following sense. Histories H and K are equivalent if for any history L, $F(H \cup L) = F(K \cup L)$. That is, K may as well have been the real history, because one cannot tell the difference by observing either the present state of F or its future states as more operations are applied. The implementation does not store H itself, but stores an equivalent K that is generally smaller. For example, if the same attribute is set twice in succession, the first *set* operation is not retained in K.

The history level lets us make sense of the results when attribute operations are performed independently on two replicas of the same object in different repositories. If the history K_A at repository A is propagated to repository B, we can combine it with the history K_B simply by forming $H = K_A \cup K_B$; the new $F(H)$ then gives a well-defined and reasonable final state for the object. This technique is adapted from Grapevine [6].

In practical terms, then, we propagate attribute changes from one repository to another by sending the timestamped change tuples of the source repository to the destination repository, then unioning them in to the second repository's change history.

4.4.7 Replication Access Control

We needed to add a few special access control lists to the repository to provide proper security for the replication primitives. As with other ACLs, if an object does not have its own replication access control lists, it inherits the access controls from its parent directory. The `#mastership-to` access control list for an object lists the repositories that mastership on the object can be ceded to, `#mastership-from` lists the repositories that mastership can be accepted from, and `#replicate-from` lists the repositories that replicas can be taken from. In addition, if an object has an `#replicate-from-noac` access control list, it will accept replicas of the object's data from the repositories listed, but it will not accept replicas of their access control attributes (that is, those attributes whose names start with `#`. No special ACL is needed to control giving replicas; read access by the requesting user is sufficient for that. Administrative access is required to change these lists.

Why are these lists needed? With mastership, we do not want to accept mastership from a rogue repository that might claim to have mastership on a object that is really mastered somewhere else, since this will cause us to come into disagreement with the repository that really has mastership. We also do not want to give away mastership on an object to an unauthorized repository. With replication, we do not want to copy in data from a rogue repository that might maliciously supply incorrect values.

For convenience, we actually allow any user to replicate data into his local repository, as long as he has read permission for the data in the remote repository, he has search permission on the directories involved in the local repository, and the remote repository is on the proper access control list. Because replicating data does not change it, there would be no sense in requiring the user to have write permission in the local repository. In an earlier version of the repository, we required the same permissions to replicate data in as we would have required to write new data; this caused several annoying problems, such as a user not having permission to replicate in the value of an object's `#owner` attribute unless he was already the owner in the local repository, even though the value to be copied in from the remote repository would make him the owner.

4.4.8 The Replicator

The primitives listed in the previous sections give us a safe way to copy data and transfer mastership between repositories, but they are quite low-level. In this section we briefly describe a higher-level replicator. It is available both as a standalone tool (**vrepl**) that can be invoked from the command line and as a library that can be called by other tools.

Our replicator takes as input a set of pathname patterns and the network addresses of two repositories, a source and a destination. The replicator walks the directory tree of the source to find all names that match the patterns and copies those that are not already present at the destination. It also updates the mutable attributes of every name that matches by merging update tuples from the source into the destination. As a trivial example, the command **vrepl -d east.vestasys.org -e+ /vesta/west.vestasys.org/vesta/repos/LAST** would replicate the highest-numbered version of the Vesta repository source code from the local repository into the repository at `east.vestasys.org`. The full pattern language is similar to Unix shell **glob** patterns with some extensions (such as the pattern "LAST", which matches the highest version number that is not a stub). Prefixing a pattern with "**+**" adds the objects that match it to the set to be copied; prefixing it with "**-**" removes them.

The replicator also has a feature that replicates everything needed to do a particular Vesta build. For example, the command **vrepl -s west.vestasys.org -e@ /vesta/west.vestasys.org/vesta/repos/124/.main.ves** would replicate all the source code needed to recompile version 124 of the repository implementation from the `west.vestasys.org` repository into the local repository, including the entire programming environment (libraries, compilers, etc.) that was available to the build. This feature works by first parsing the build description (written in the Vesta system description language), walking its import tree, and emitting a **+** pattern for every package found; then passing these patterns on to the basic replication algorithm. In practice, it has turned out that **@** is used far more often than **+** and **-** are.

The replicator tool can be used to "push" sources from the local repository to a remote one, to "pull" sources from a remote repository to the local one, or even to copy sources between two different remote repositories. The repository does not currently include support for automatically triggering a replication when new package versions are checked in, though there is some support for automatic replication in **vcheckin**, described below. When two sites are cooperating, it works well to set up a periodic run of the replicator (say, from a Unix cron job) to copy new sources between them. The replicator can also be run manually when needed.

Performance measurements on the replicator are presented in Section 9.3.5.

4.4.9 Cross-Repository Checkout

When two sites running separate repositories are closely cooperating, users at one site may want to check out packages whose master copies are in the other site's repository. In this section we outline how we extended the development cycle tools described in Section 4.1.4 to support this.

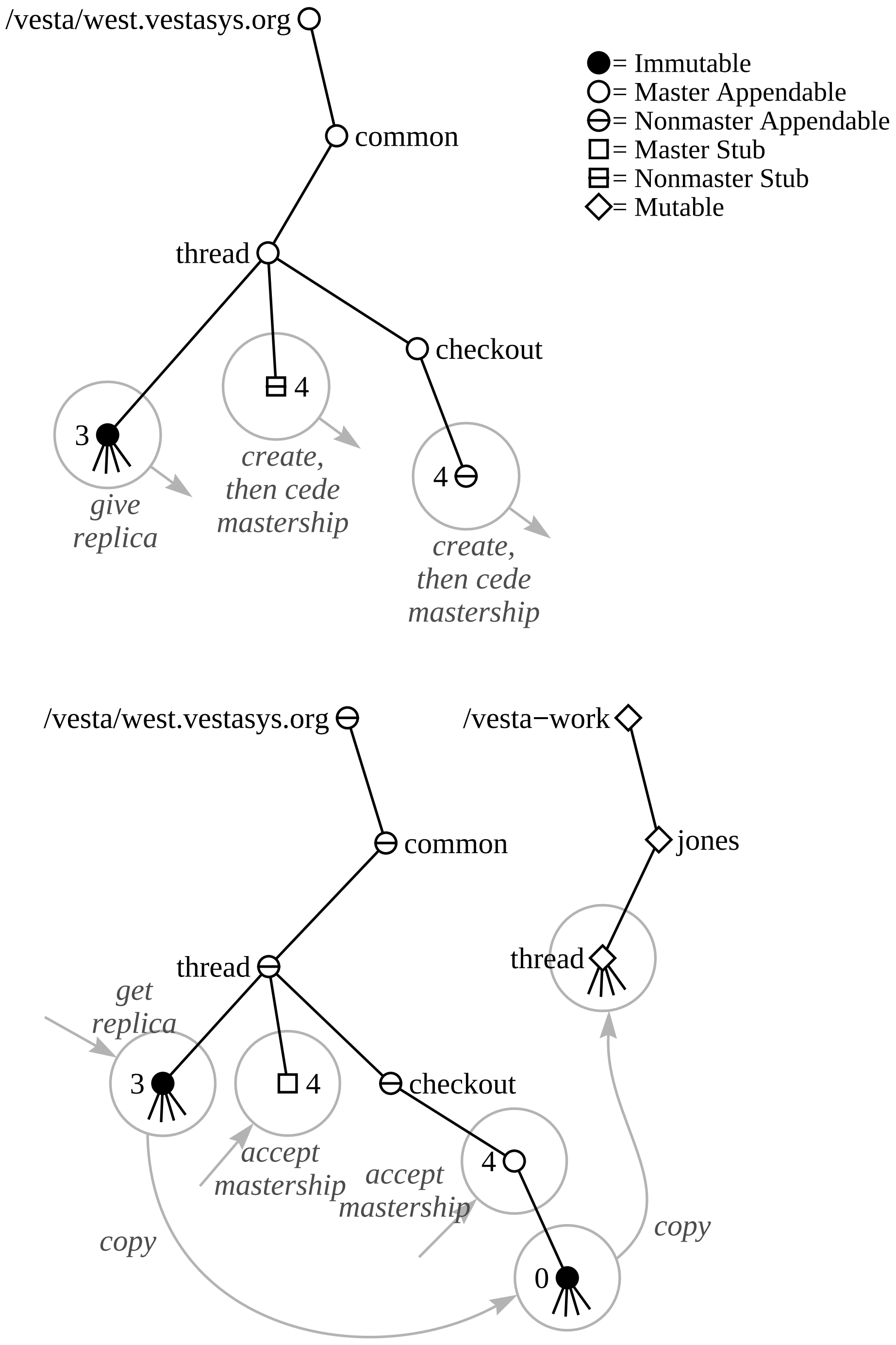


Figure 4.7: Cross-repository checkout.

Almost all of the changes for cross-repository operation were in **vcheckout**, as shown in Figure 4.7. The source repository, where the package being checked out is mastered, is shown at the top; the destination repository, which the user doing the checkout wants to work in, is shown at the bottom. Notice that the actions in the destination repository are similar to the single-repository case in Figure 4.2. The steps in cross-repository checkout are:

1. Examine the master replica in the source repository to find the highest version number.

2. If this version does not exist in the destination, call the replicator to copy it in.

3. Create the reservation stub and empty session directory in the source repository.

4. Call the replicator to copy them to the destination repository.

5. Transfer mastership on them from the source to the destination.

6. Insert version 0 in the session and create the working directory at the destination.

No changes were needed to **vadvance**, since both the working and session directories are in the destination repository.

It would not have been strictly necessary to change **vcheckin** either, since **vcheckout** moves mastership of the reservation stub to the destination repository. However, it is likely that the source repository will want a copy of the new version soon, so we modified **vcheckin** to call the replicator and copy it there after checking it in locally.

The Vesta tools for creating new packages (**vcreate**), branching the version sequence (**vbranch**), finding the latest version (**vlatest**), and finding who has packages checked out (**vbranch**) also required minor modifications to be cross-repository aware. The changes required were similar to what was done to **vcheckout** but considerably simpler.

One problem currently remains with the cross-repository tools. In the single-repository case, each of our tools uses the repository's short atomic transaction support to make its complete action atomic. This support does not work across multiple repositories, so the tools become nonatomic in this case. With **vcheckout**, steps (3) and (6) are individually atomic, but if there is a failure between them, the checkout is left in an incomplete state. We have not yet automated the recovery from this state, but it is not hard to recover manually.

The performance of the tools in the single- and cross-repository cases is compared in Section 9.3.4.

Chapter 5

System Modeling Language

There are two inputs to the construction of a software system: the *sources* and the *instructions* for producing the system from those sources. For small code bases, the form taken by these instructions can be rather simplistic. However, for even moderately large systems, experience shows that a build language with more flexibility and support for abstraction makes it easier to specify the construction of such systems. For example, because Make does not provide any abstraction facilities, writing Make build descriptions for such systems is a bit like doing assembly language programming.

By contrast, Vesta's build language, or *system modeling language*, is a full-fledged programming language that supports complete, hierarchical build descriptions, and is vastly more powerful and flexible than Make's. Moreover, the language's support for functional abstraction makes it possible to encapsulate low-level building tasks, thereby simplifying the system descriptions written by end users.

This section describes the Vesta system modeling language and the structure of Vesta system models. It also describes the standard construction environment we have written for building C, C++, and Modula-3 programs, gives examples of models for different kinds of packages, and describes the mechanisms provided by the construction environment for performing customized builds.

Although this section describes some of the important features of the Vesta system modeling language, it is by no means complete. Rather, its aim is to convey the form taken by typical user-written models, to show that they are simple, and to present one way of organizing a standard construction environment that supports flexible customization mechanisms. Appendix A presents the language's complete syntax and semantics.

5.1 Motivation

In Vesta, the instructions for building a system are contained in *system models*. System models are programs written in the Vesta system modeling language. They describe how to build a software system from sources, i.e., from scratch. While the system models are being evaluated, tools like compilers and linkers are invoked to build the program; the resulting binaries are typically returned as part of the evaluation result.

A few essential requirements dictate the structure and functionality of the system modeling language:

- The builder (i.e., the language interpreter) must be able to construct systems repeatably, incrementally, and consistently.
- The complexity of a software description should be proportional to the conceptual complexity of building the system it describes.
- The language must be practical for developers to use; that is, it must be adaptable to a variety of software development methodologies and organizational processes.

It follows that the modeling language should be organized so that even errors in system models cannot interfere with repeatable and consistent builds. Moreover, the language should support incremental building as the norm, so that good performance is the rule, not the exception. It also follows that the core language facilities should be as basic and "methodology-neutral" as possible, and that support for particular styles of system construction or organization should be programmed in the language, not built in to the language or the builder.

These requirements and immediate consequences establish the major properties of the Vesta language:

- Repeatability and consistency give rise to two properties: all information required to build a system from sources is captured in system models, and all sources are immutable.
- Incrementality leads to the choice of a functional language, because each function invocation then represents a unit of work that can be conveniently cached.
- "Proportional complexity" is achieved by providing a flexible modular structure in which reusable abstractions can be easily defined.

- Methodological neutrality compels a careful choice of basic data types and primitive operations.

It has been rightly said that one can write a bad program in any language, and the Vesta modeling language is no exception. But experience demonstrates that it is possible to write good programs (that is, concise, precise system descriptions) in this language without great difficulty. The facilities of the Vesta language allow development groups to organize their system descriptions to fit their development methodology, while achieving the central benefits of repeatable, consistent, and incremental builds. The remainder of this section covers the important language facilities, and shows how they can be used to model a particular development structure.

5.2 Language Highlights

Unlike software construction languages such as Make's, the Vesta system modeling language is a full-fledged programming language with a well-defined syntax and semantics (see Appendix A).

The Vesta modeling language is functional, modular, dynamically typed, and lexically scoped. Its value space contains booleans, integers, text strings, lists, closures, and bindings. The first five data types are the familiar ones from Algol-like languages and LISP. Bindings are ordered lists of name-value pairs. The language contains about 60 built-in functions for arithmetic and boolean operations, for basic manipulations of texts, lists, and bindings, and for invoking external tools. There is also a built-in function for applying a closure to a list of values in parallel, which we use to achieve coarse-grained parallel compilation. Although all values are typed, and operations are type-checked at execution (interpretation) time, static typing is optional.[1]

There are clear advantages and disadvantages to using a complete functional programming language as the basis for software descriptions. The main advantage is that a functional language forms a tractable basis for caching of function calls, which in turn is the basis for Vesta's incremental building mechanism. The main disadvantage is that new users must learn a new language, different from other scripting languages for program construction. We consider the advantage of reliable incremental building to be overwhelmingly important, and therefore have chosen to design a new language, but we have done all we can to minimize the barrier this poses for users. To reduce initial unfamiliarity, the Vesta language uses

[1] There are simple provisions in the syntax for annotating the types of variables and function results. However, these annotations serve only as comments; they are ignored by the current evaluator.

a C-like syntax and C semantics wherever practical. To reduce conceptual unfamiliarity, Vesta provides simple templates for system models describing common situations, and a “library” of functions for system construction that can be invoked from more customized models. Thus in most cases, writing a model reduces to filling the blanks in a standard template.

Short, simple templates require the ability to decompose the complete description of a system build into component parts. That is, the Vesta language must support organizing the instructions for building a system into a set of modular units. This is accomplished by enabling one system model to reference, or *import* another, yielding a hierarchical structure that can reflect the component structure of the software system itself.

Modular structures are designed to localize information, which is generally a wise methodological principle for organizing software systems. However, the nature of the build process frequently requires broad, systematic alterations of default behavior, and the description language must accommodate these situations gracefully. For example, a customized action may apply to an entire build (“build this program and all the libraries it uses with debugging symbols”) or to a large part of it (“build the symbolic integration library with optimization”). In practice, this means that the functions for building a system must be sufficiently parameterized so that the construction of the individual components can be sensibly customized by the callers of those functions.

How is the potential chaos of customization controlled? By extensive use of parameterization in the system descriptions, it is possible to localize the actual customized values of parameters near the root of the hierarchy of system description modules. To facilitate customized builds, the current construction parameters are collected together in a single composite value called the *environment.* In addition to parameters that control how tools like the compiler and linker are invoked, the environment passed to each such tool contains a complete representation of the file system in which the tool is to run. The Vesta language’s *binding* type is used to represent environments, and the language includes operators for easily merging and overriding particular parts of the environment.

The Vesta notion of environment is central. A precisely specified build is nothing more than a series of tool invocations (e.g., compiles and links) in a controlled naming environment. That environment changes subtly but crucially on each tool invocation, and the process of constructing these many slightly different environments must be both convenient to express and efficient to implement. So, the Vesta language provides:

- a mechanism by which the current environment is easily passed between functions,

- a *binding* data type used to represent naming environments (including build customizations and file directories), as well as language facilities for easily creating and augmenting bindings,
- a language primitive for executing a tool in a particular environment, and
- a *closure* data type for delaying tool invocations until the files they produce are needed.

The next four subsections describe these particular aspects of the Vesta system modeling language in more detail.

5.2.1 The Environment Parameter

As stated previously, Vesta builds are parameterized and customized by amalgamating file systems and build options together in a single composite binding value called the environment. Since the environment plays such an important role, there is special treatment for it in the Vesta language.

In particular, every Vesta function takes an implicit final parameter named "`.`" (pronounced "dot") denoting the current environment. That is, a function declared with n explicit formal arguments actually has $n+1$ arguments, the last being the implicit formal parameter named "`.`". The name "`.`" is a legal Vesta identifier, so it can be used as a normal variable name. A function with n formal arguments can be called with either n or $n+1$ actual arguments. In the former case, the current value of "`.`" is bound to the implicit formal parameter "`.`"; in the latter case, the $n+1$st actual parameter is bound to the implicit formal parameter "`.`". In our experience, the environment is rarely passed explicitly; it is usually "inherited" from caller to callee.

5.2.2 Bindings

Among other things, the current environment embodies both build customizations and the file system in which tools are invoked. A convenient data type for representing both these kinds of values is the *binding*, an ordered map from names (strings) to arbitrary Vesta values. Bindings can be nested.

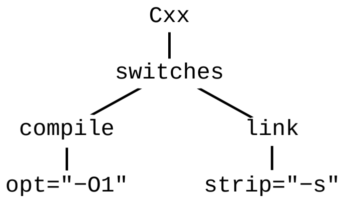
Build customizations are represented by hierarchical arrangements of mappings from customization names to values. For example, Figure 5.1 shows a binding that might be used to specify the options for compiling and linking C++ programs. The code in Figure 5.1a demonstrates the syntax for constructing a binding:

$$[\ name_1 = value_1, \ldots, name_n = value_n\ ]$$

denotes the binding with names $name_i$ and corresponding values $value_i$. Figure 5.1b shows a tree representation of the binding. The hierarchy arises because a binding is itself a value in the Vesta language.

```
[ Cxx= [ switches= [ compile= [opt="−O1"], link= [strip="−s"] ]]]
```

(a)



(b)

Figure 5.1: The code for constructing a binding of build options (a), and the same binding represented as a tree (b).

Binding data structures can also be used to represent file systems. Each directory in such a file system is represented by a binding that maps each name in the directory either to a subdirectory (a nested binding) or to a file (a text value).[2] Hence, a directory d containing files named f_1 and f_2 with corresponding contents c_1 and c_2 would be represented by the binding $d = [\, f_1 = c_1,\ f_2 = c_2]$.

The Vesta evaluator stores bindings in main memory, making the traversal and manipulation of these "directory" structures very efficient compared to conventional file systems. Since bindings are lightweight, new file systems can be created and augmented on the fly for the particular needs of each tool. In traditional development environments, creating such customized file systems would be too expensive because it would require disk operations. Instead, Unix tools use cumbersome mechanisms like search paths. In Vesta, a custom file system is easily constructed for each tool, eliminating the need for long search paths and making the file choices more explicit.

The Vesta language includes syntax and operators for creating bindings, selecting a binding element by name, merging two bindings, and subtracting elements from a binding. Bindings are constructed using the square bracket syntax shown above. The value named n in the binding b is selected by writing b/n. For example, the compiler switches are selected from the binding named `options` in Figure 5.1 by writing `options/Cxx/switches/compile`, which evaluates to the singleton binding `[ opt = "-O1" ]`.

[2] Because file contents are represented abstractly in the language by text values, the implementation must be able to handle large text values that are read and written as files; it does this by representing such values internally as pointers to files stored in the repository (Section 4.2.1).

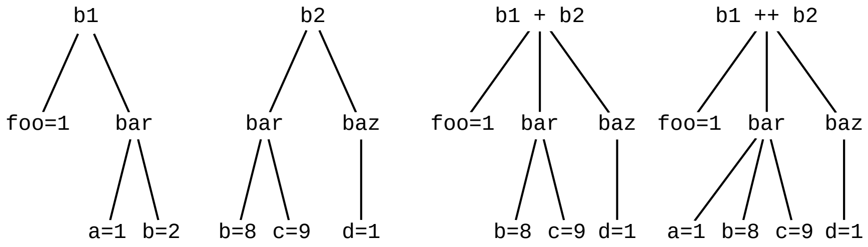


Figure 5.2: The results produced by overlaying (+) and recursively overlaying (++) two bindings, b_1 and b_2. Notice that in $b_1 + b_2$, the *bar* component of b_1 is ignored, while in b_1 ++ b_2, the *bar* components of b_1 and b_2 are overlaid recursively, with the values from b_2 taking precedence.

Bindings are merged using the *overlay* (+) and *recursive overlay* (++) operators. The expression $b_1 + b_2$ is the binding containing the union of the names in b_1 and b_2; for those names that appear in both bindings, the value bound to the name in b_2 takes precedence. The binding b_1 ++ b_2 is like $b_1 + b_2$, except that where both bindings contain the same name, and when that name is bound to a nested binding in each, the nested bindings are overlaid recursively. Figure 5.2 shows an example application of the + and ++ binding operators.

The overlay and recursive overlay operators provide the basis for extending file systems and for applying customizations. For example, if "`.`" is a binding denoting the current environment, and "`.`/root" is the root of a file system, then the environment can be extended to include the extra directories and files contained in the binding `fs` by writing:

```
. ++= [ root = fs ];
```

(As in C, "a *op*= b" is shorthand for "a = a *op* b".)

Compilation options can also be overridden using overlays. For example, to change the build "options" above so as not to use optimization, you could write:

```
options ++=
  [Cxx = [switches = [compile = [opt = "-O0"]]]];
```

Such bindings are so common that the language includes a shorthand for writing them. For example, the same overlay can be written as follows:

```
options ++= [ Cxx/switches/compile/opt = "-O0" ];
```

By definition of the recursive overlay operator, this assignment leaves the linking options unchanged, but it binds the `opt` field of the nested `compile` binding to "-O0". Both this example and the previous one demonstrate how selective portions (i.e., subtrees) of the environment can be changed by a single source statement in a Vesta model.

5.2.3 Tool Encapsulation

Consistent with the functional nature of the language, invocations of external tools like compilers and linkers are expressed as function calls. To call an arbitrary tool in an encapsulated environment, the Vesta language provides a built-in primitive function called `_run_tool`. (To avoid name conflicts with user-defined functions, all primitive functions in the Vesta language begin with an underscore.) Hence, there is a single uniform mechanism for encapsulating new tools with Vesta; no modifications to a tool are required to invoke it. Tool invocations via `_run_tool` are launched by a runtool server running on a machine of the appropriate platform.[3]

The `_run_tool` primitive is parameterized by a command line and several other arguments describing the environment in which the tool should be run. It returns a binding with fields describing the tool's outcome. The complete specification of `_run_tool` is given in Appendix A. Among the `_run_tool` arguments are flags for specifying that the function call result should not be cached in the event of various error conditions, for example, if the tool returns a non-zero status code or if it writes anything to the standard error output.

Of course, the environment parameter "`.`" is also an argument to `_run_tool`; it encapsulates both the tool's environment variables and the file system in which the tool is run. Environment variables are passed in `./envVars`: if it is defined, the value `./envVars` is taken to be a binding that defines the names and values of environment variables to be set for the tool's execution. The file system seen by the encapsulated tool is passed in `./root`. In particular, any absolute pathnames referenced by the tool are looked up in `./root`, while relative pathnames are looked up in the nested binding `./root/.WD`. As described in Chapter 3, file system references are trapped by the repository, looked up in `./root` or `./root/.WD` by the evaluator, and recorded by the evaluator as dependencies of the `_run_tool` function call. All environment variables defined in `./envVars` are also recorded as dependencies.[4]

Although the `_run_tool` function provides a flexible means for invoking external tools, it is rather primitive. We do not expect most user-written system models to call it directly. Instead, we provide reusable libraries of Vesta functions called *bridges* as part of the standard construction environment for interfacing to develop-

[3] To determine which machine to contact for a given platform, the evaluator consults the Vesta site configuration file, which contains a list of suitable machines for each platform type; it then iterates through the list to find an unloaded or lightly loaded machine. If the evaluator is doing a parallel build, it may have several tools running on one or more machines at the same time.

[4] We would have preferred to record dependencies on only those environment variables read by the tool, but there is no mechanism for trapping such references. In any event, we have never needed to set any environment variables before invoking a tool, so we have not experienced any false dependencies on unused environment variables.

ment tools (see Section 5.4 below). We also expect bridge code to hide the details of invoking _run_tool appropriately for multi-platform and cross-platform development. However, our current bridges do not yet provide such support.

5.2.4 Closures

A *closure* is a function paired with a context that supplies values for all of the unbound variables appearing in the function's body. In Vesta, closures are first-class values; that is, they can be passed as parameters or embedded in data structures just as any other value can.

Each Vesta model defines a closure with a single implicit parameter (namely, "."). Hence, a model's body can be invoked using the normal function call syntax. At the start of an evaluation, the Vesta evaluator invokes the model on which it is run by calling it with an empty environment.

Closures have several uses. For example, a collection of callable functions can be represented by a binding that maps function names to closure values. Consider the following model:

```
{
    f() { /* body of closure f here */ };
    g() { /* body of closure g here */ };
    return [ x = f, y = g ];
}
```

When invoked, this model returns a binding containing fields named x and y bound to the closures f and g, respectively. If that result binding was assigned to the variable b, then the closure g could be invoked by writing b/y().

Closures can also be used to effect lazy evaluation: a lazy value can be represented by a closure for computing that value. The work required to compute the value can thus be delayed until it is needed. This way, costly computations (such as compilations) whose results might not be used are not performed. Such lazy evaluation is not generally required, but in the few cases where it is needed, it can be implemented explicitly using closures.

5.3 Model Organization

With the language preliminaries taken care of, we can now focus on the organization of software descriptions in Vesta. There are two parts to a software description: a list of the sources that comprise the software, and the instructions for building those sources to produce derived object files, libraries, and executables. In Vesta, these two pieces are unified in files called system models, or *models* for short.

5.3.1 Properties of Models

Models have two important properties. These two properties, together with the immutability of all source files, are the basis for repeatability in Vesta.

First, like all sources in the Vesta repository, all models are versioned, immutable, and immortal. Hence, the instructions for building a particular version of a software artifact are never lost. In fact, models are regular files that are part of versioned packages just like source files.

Second, models are complete: the result of a model evaluation does not depend on any file, environment variable, or other aspect of the system on which Vesta is being run that is not explicitly specified, either by the model itself or by a model that it imports (directly or indirectly). Put another way, an evaluated model and the models it imports form a complete record of the sources and tools contributing to a build.

This second property implies that in addition to the particular version of each source file and model that contributes to a build, the models must also name the particular versions of all external tools invoked during the build. An in-house tool for which sources are available may be described by a model for building the tool from source. A tool for which no sources are available is described by a model that simply returns the pre-built tool binaries (which are stored as "source" in the repository). Both kinds of tools are encapsulated in the same way by the standard construction environment.

At this point, you may imagine that models will be difficult to maintain, since it would seem tedious to have to name and update a version number for each source file referenced within a model. However, the burden of referring to correct source versions is mitigated by three factors. First, sources are versioned and imported at the granularity of packages (directory trees), not individual files. Second, due to the hierarchical arrangement of packages, updating the version of a single high-level import can indirectly import new versions of many packages lower in the hierarchy. Third, a model directly references only source files that exist in the same package as the model itself, though it may import models either inside and outside the package. References to local source files and local models do not require a version specification because the package is versioned as a whole; these references are implicitly bound to files in the same package version as the model itself.

References to models in other packages do require an explicit version specification. This requirement is sensible: to incorporate a new version of a different package, you have to explicitly update your building instructions. Put another way, each developer has complete control over when to incorporate new versions of other developers' changes into his or her own build. Since this is a fairly common operation, it needs to be easy for users to carry out. We therefore provide a **vup-**

date tool for quickly updating a package's imports from the command line. There is also a **vimports** tool for listing the transitive closure of a model's imports.

5.3.2 Requirements on Model Organization

A system model is the unit of modularity in a Vesta system description. Thus, a complete description is built up by writing a collection of system models that reference each other. But how is this to be done? That is, what organizing principles or methodology should be used to create a collection of understandable and usable models?

Of course, there is no single answer to this question. To demonstrate the Vesta language's feasibility for this purpose, the following sections describe a particular organization of system models, derived from some particular developmental requirements, characterized as follows:

- We expect that the files in a package under development change more frequently than the environment in which that package is built. Consequently, it is sensible and useful to separate the description of the package from the description of the environment for building the package.

- From the perspective of system construction, many packages have a similar form. For example, a package may build either a complete program or a body of code intended to be linked into other programs. (We tend to call the former an *application* and the latter a *library*.) Generally, the models for applications look quite similar to one another, as do the models for libraries. To simplify and standardize user models, the model hierarchy should be organized so that common features can be factored out into system-provided models. In effect, these standard forms become part of the development environment, available to be invoked by short, simple models written by developers. Of course, the standard forms can be modified or entirely bypassed in exceptional cases.

- Occasionally, a developer will need a specialized environment. The specialization might include special versions of libraries, or libraries compiled in a special way or with a special version of a compiler. Consequently, the system description for the construction of the environment must be extensively parameterized to permit an individual developer to create the custom environment needed to develop a package.

- Any constructed system is linked together from a collection of components, typically derived object files and library archives. The dependencies between these components induce an abstract hierarchy; in practice, the components must be given to the linker in a total order consistent with the hierarchy's partial order. The library hierarchy is distinct from the package naming hierarchy, so models must represent the library hierarchy explicitly.

- Systems are built by executing tools (e.g., compilers and linkers), which generally have extensive options that are needed only for specialized purposes. These options must be accessible from Vesta models, but most users of the tools never need to be aware of them. Consequently, the mechanisms for compiling and linking package components must hide these complexities by default, without compromising a developer's ability to invoke the tools in arbitrary ways.

These considerations lead to the organization of a *standard construction environment*, which is discussed in Section 5.4, and a hierarchy of library descriptions, which we consider next.

5.3.3 Model Hierarchies

To accommodate large-scale software, Vesta software descriptions are organized hierarchically; that is, they are trees with models as nodes and model imports as edges.

For example, Figure 5.3 shows the hierarchy of packages comprising the complete release of a hypothetical mail system. Directed arrows in the figure represent model imports across packages. At the lower right of the figure are two library packages named *mail/send_rcv* and *mail/index*, for transporting and indexing mail, respectively. The release consists of two application packages named *mail/inc* and *mail/search*, which provide facilities for incorporating new mail and performing queries on mail messages.

At the root of the tree is the release package itself. The release package does not contain any source code. Instead, its model imports the standard environment, the packages for the two application programs, and an *umbrella library* package named *mail/lib*, which itself imports the two base libraries.

As shown in Figure 5.3, the standard environment imports two kinds of models:

- the bridge models that export interfaces for invoking collections of tools, such as language-specific compilers and linkers, and

- all of the standard libraries required by client applications (in this example, only two standard C libraries are shown).

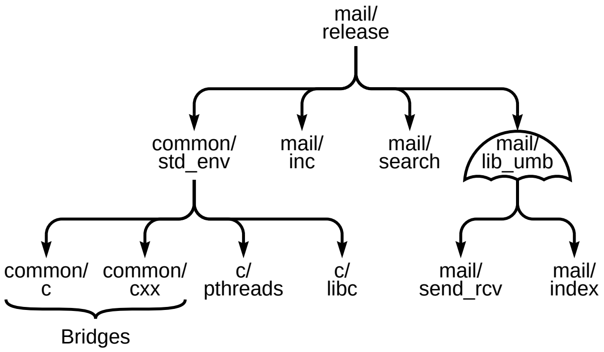


Figure 5.3: The packages of a hypothetical mail system and their imports.

As indicated by the subtree rooted at the *mail/lib_umb* package, library packages themselves may be arranged in hierarchies. In fact, our standard construction environment supports three kinds of libraries: *umbrella libraries*, which are collections of other libraries, *leaf libraries*, which are libraries built from source, and *pre-built libraries*, which are libraries that exist only in binary form. In Figure 5.3, *mail/lib_umb* is an umbrella library, *mail/send_rcv* and *mail/index* are leaf libraries, and *c/pthreads* and *c/libc* are pre-built libraries.

Umbrella libraries are mostly a convenience: they provide a way to package up a collection of related libraries into a single conceptual entity. They also encapsulate the order in which the child libraries must be listed on the final link line to link correctly; the bridge models arrange to link the libraries in a total order consistent with the partial order induced by the library hierarchy. In this example, the release package need only import one library package rather than two. In larger software systems—such as the release model for the Vesta system itself, which contains 9 leaf libraries—the use of umbrella libraries serves to make client models simpler and more succinct.

The particular arrangement of model imports shown in Figure 5.3 is not fixed. The models could have been arranged in other ways, and the current arrangement is partly a consequence of how our standard construction environment is structured, as described below in Section 5.4.

Perhaps the most curious aspect of the current arrangement is that the umbrella library is imported by the release package, rather than by the individual application packages themselves. An alternative arrangement is shown in Figure 5.4. The models certainly could have been structured that way. The advantage to the arrangement shown in Figure 5.3 is that the umbrella package is imported in only one place, so only one import statement must be updated when a new version of

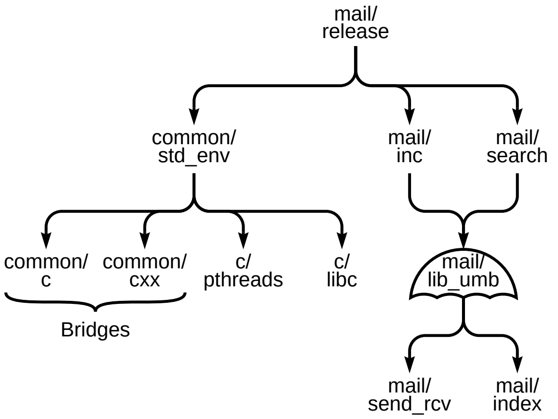


Figure 5.4: An alternative arrangement of package imports of the hypothetical mail system of Figure 5.3.

that package becomes available.

5.3.4 Imports

Recall that a model defines a closure of a single argument (namely, "`.`"). The *import* statement binds a local identifier to the closure corresponding to the model being imported. For example, the mail system's release model might begin:

```
import
  std_env =
    /vesta/west.vestasys.org/common/std_env/9/build.ves;
{
  // build the current environment
  . = std_env()/env_build("AlphaDU4.0");

  // instructions for building the complete release...
}
```

The `import` statement binds the local identifier `std_env` to the closure corresponding to version 9 of the standard environment model. That model is called by the expression `std_env()`. When evaluated, the standard environment model returns a binding representing an abstract interface as described above in Section 5.2.4. That binding maps the name `env_build` to a closure that builds the current environment for the target platform named by its first argument. The result is bound to the distinguished environment identifier "`.`" for use by the rest of the build.

5.3.5 Control Panel Models

Certain aspects of a package build—such as which version of the standard environment model to use and any desired build customizations—are more transient in nature than the instructions for building the package itself. To facilitate the modification of these transient attributes, we envisage a graphical user interface program called the Vesta *control panel* for viewing and modifying the attributes, and for initiating Vesta builds of the package in question.

We have not written a control panel application, but we imagine it would work by reading and writing its own legal Vesta models, called control panel models. Each control panel model would embody the imports and customizations specified by the user in the graphical user interface.

In the absence of a Vesta control panel, we simply create and edit control panel models by hand. By convention, they are named *.main.ves*. They are highly stylized, and usually on the order of 10 to 20 lines long, depending on the number of customizations specified. A control panel model builds a specific instance of its package by setting parameters, importing a suitable version of the standard environment, and then using another model in the package (conventionally named *build.ves*) to compile and link the package's source code in this environment. The *build.ves* model is designed to be independent of the version of the standard environment in use, so it can also be called from elsewhere to build the same package in different environments.

A package may produce several different classes of derived files. For example, a package may produce executables meant to be exported to clients, test programs, libraries, and documentation. We do not necessarily want to produce each of these artifacts on every build. So by convention, a package's *build.ves* model returns a binding that contains a separate closure for building each category of derived file. The control panel model then invokes only those closures in the binding that are currently of interest to the user. In essence, each category is built lazily.

Figure 5.5 shows a typical control panel model. The `import` lines at the beginning import the package's local *build.ves* model and the standard environment model, bound to the names `self` and `std_env`, respectively. Line (1) invokes the `env_build` function of the `std_env` model to build an environment targeted to Alphas running the Digital Unix 4.0 operating system. Line (2) evaluates the local *build.ves* model, binding the result to the local variable `b`. Finally, line (3) selects the `lib` and `progs` fields out of the binding `b`. Each of these fields is a single-argument closure (the all-important implicit argument "`.`"). Each closure is invoked, and the results are bound to fields of a binding, which is returned as the overall result of the package build.

Each package's *build.ves* model is easily implemented by simply importing

```
import
  self = build.ves;
  std_env =
    /vesta/west.vestasys.org/common/std_env/9/build.ves;
{
  // bind the standard environment to ‘.’
  . = std_env()/env_build("AlphaDU4.0");       // (1)

  // build selected components
  b = self();                                  // (2)
  return [ lib = b/lib(), progs = b/progs() ]; // (3)
}
```

Figure 5.5: A typical control panel model.

separate sub-models within the package for building each category of package result, and returning a binding containing the closures corresponding to those imported models. Hence, like control panel models, *build.ves* models are simple and highly stylized. The real work of building the package is deferred to the imported sub-models, whose contents we describe in the next section.

5.4 The Standard Construction Environment

If users were required to construct their programs using the low-level `_run_tool` primitive, their system models would be quite long and complicated. To simplify and standardize user models, Vesta provides a *standard construction environment*, a collection of reusable models for performing common building operations.

This section does not describe the workings of our standard environment directly. Instead, it presents examples of client models written against the standard environment, and it describes the mechanisms available to clients for customizing their builds. Our aim here is not only to convey the general form taken by client models, but also to demonstrate the language’s power, flexibility, and utility.

We designed the standard construction environment to be flexible and to suit our needs, but different development organizations might choose to structure their construction environment differently. Writing a new construction environment is entirely possible; ours is just one example.

5.4.1 Components of the Standard Environment

Our standard construction environment consists of a `std_env` model and a collection of bridge models. Each bridge model exports an interface to some collection of related tools. Vesta-2 bridges are models written in the regular Vesta system modeling language, and they are interpreted by the Vesta evaluator just like client models. They are not hard-wired into the Vesta system in any way.

Bridge models must handle the vagaries of interfacing to legacy tools and of supporting customized builds, so they tend to be much more complicated than client models. The `std_env` model and the seven bridge models comprising our standard construction environment are roughly 1,800 total lines of Vesta code. Approximately two-thirds of that code is in the bridges for C/C++ and Modula-3 programs. These bridges export functions for building programs, pre-built libraries, leaf libraries, and umbrella libraries written in their respective languages.

5.4.2 Building Library Models

The standard construction environment emphasizes the organization of code into a hierarchy of *libraries*. Although libraries may be implemented in different ways, they share a common structure. Each library is modeled as a set of interface files and a set of files that implement the interfaces. The format and arrangement of these files depend on both the language in which the library code is written (and thus which bridge is used to model it) and the library's place in the hierarchy.

For concreteness, we restrict ourselves for the rest of this section to describing the library facilities provided by the C/C++ bridge. The facilities of the Modula-3 bridge are similar.

Recall that the standard construction environment supports the construction of three kinds of libraries: umbrella libraries, leaf libraries, and pre-built libraries. The bridge exports a closure for building each kind of library. Since the client may wish to customize how a library is built, it is necessary to defer the actual construction of the library until it is needed. Hence, the closures for each library type do not actually build anything. Instead, they embed the library source files in a binding that serves as a set of *instructions* for constructing the library. The library is built with the appropriate customizations only when a program requiring it is built.

This approach differs sharply from the conventional one, in which particular versions of standard libraries are placed in standard locations in the file system hierarchy. The Vesta standard environment places *highly parameterized construction recipes* at the developer's disposal rather than particular, previously built versions. The difference in flexibility and development ease is enormous.

```
files
  c_files = [ Lex.C, Scan.C, Index.C ];
  h_files = [ Lex.H, Scan.H, Index.H ];
  priv_h_files = [ IndexRep.H ];
{
  return ./Cxx/leaf("libMailIndex.a",
    c_files, h_files, priv_h_files);
}
```

Figure 5.6: The model for building the leaf library of the hypothetical *mail/index* package.

To see how the Vesta approach works in practice, we first look at the organization of a leaf library. Figure 5.6 shows the model for the hypothetical *mail/index* library of Figure 5.3; it describes how to build a library archive named *libMailIndex.a*.

In general, a model's `files` clause binds local variable names to local files and directories within the package; the paths in a `files` clause are interpreted relative to the directory containing the model itself. In this case, the first four lines of the model introduce three local variables (`c_files`, `h_files`, and `priv_h_files`) whose values are bindings that associate the listed names with the corresponding local file contents. For example, the local variable `priv_h_files` is assigned a binding that associates the name `IndexRep.H` with the contents of that local file. Thus the source files are separated into three groups: 1) C++ source files that must be compiled to produce object files, 2) header files that are provided to clients, and 3) private header files that are required only by the implementation.

The body of the model consists of a single function call. The function being invoked is named `./Cxx/leaf`. It is the function named `leaf` exported by the C++ bridge, which in turn is part of the current environment "`.`". The `./Cxx/leaf` function simply packages up the files into a single binding that will serve as the instructions that later will be used to build the actual library archive (possibly with developer-defined customizations).

Figure 5.7 shows a model for building the hypothetical *mail/lib_umb* package's umbrella library. The first three lines of this model import the two top-level models of the umbrella's component libraries, binding those models to the local variables named `send_rcv` and `index`. The body of the model first assigns the local variable `libs` to a list of the libraries comprising the umbrella. (A list is denoted by a comma-separated sequence of values enclosed in angle brackets.) Note that the list of libraries also includes the standard *pthreads* and *libc* libraries. These libraries do not have to be imported, but instead are accessed from "`.`", where they were

```
import
  send_rcv =
    /vesta/west.vestasys.org/mail/send_rcv/1/build.ves;
  index =
    /vesta/west.vestasys.org/mail/index/3/build.ves;
{
  libs =  < send_rcv(), index(), ./C/libs/c/pthreads,
            ./C/libs/c/libc >;
  return ./Cxx/umbrella("libMailUmb", libs);
}
```

Figure 5.7: The model for building the umbrella library of the hypothetical *mail/lib_umb* package.

installed by the standard environment. Including these standard libraries in the umbrella obviates the need for application models to mention them. The model's final line invokes the C++ bridge's `umbrella` function, which packages up the supplied libraries in an umbrella with the supplied name.

The library hierarchy provided by umbrella models greatly simplifies the developer's job in specifying which libraries are needed to link a complete program, and the order in which they must be listed on the link line. Rather than having to determine and explicitly specify (in order) the set of all necessary library archives, as is generally required in the Unix environment, the developer specifies a small number of higher-level umbrella libraries. As a result, the developer's models are both simpler and more robust. The C++ bridge function that performs program construction handles the complexities of constructing a suitable command line for the Unix linker by "flattening" the umbrella hierarchy.

The C++ bridge also defines a function for packaging pre-built libraries. A model for building a pre-built library simply invokes this bridge function, supplying the library file and its associated header files as flat bindings. These models are typically even shorter than the models for building umbrella libraries.

Now that we've described how to build the three kinds of library models, we turn to the application models that use them.

5.4.3 Building Application Models

The model for an application program specifies the source and header files[5] comprising the program, the library packages to link it against, and any overrides re-

[5] The header files must be specified so they can be added to the virtual file system in which each source file is compiled.

quired to build the program in a customized manner.

For example, Figure 5.8 shows the model for building the hypothetical application package *mail/search*. The model begins with an enumeration of the sources comprising the application and the overrides required to build it. The actual construction occurs when the bridge function `./Cxx/program` is invoked. This function first builds the libraries specified by its `libs` argument, which in this case lists a single umbrella library taken from the environment. It then augments the current environment to include the header files in the `h_files` argument, compiles each of the files in the `c_files` argument, and links everything together. The value returned by the function is a singleton binding that maps the name `mailsearch` to the resulting executable.

Notice that the package expects to get the mail umbrella library *mail/lib_umb* from its environment ("`.`"). This reflects the structure shown in Figure 5.3, where a release model imports particular versions of the standard environment and the mail umbrella, adds them to "`.`", and then imports and builds the two mail applications in this environment. One could also write a control panel model to build just the search application; it too would have to choose and import particular versions of the standard environment and the mail umbrella library.

```
files
  c_files = [ QueryAST.C, ParseQuery.C, Search.C ];
  h_files = [ QueryAST.H, ParseQuery.H, Search.H ];
{
  // override
  ovs = [ Cxx/switches/link/shared_libs = "-non_shared" ];

  // build program
  libs = < ./Cxx/libs/mail/lib_umb >;
  return ./Cxx/program("mailsearch",
    c_files, h_files, libs, ovs);
}
```

Figure 5.8: The model for building the search application of the hypothetical *mail/search* package.

As we describe more fully in the next section, the standard construction environment provides a quite general mechanism for customizing different parts of the build process. It supports different customizations for the compilation of a program's source files and its libraries. For example, the fifth parameter to the `program` function, `ovs`, specifies overrides that apply to the compilation and linking of only the program's sources. In this case, the given value for `ovs` specifies that the program should be statically linked (i.e., linked `non_shared`).

5.4.4 Customizing Builds

The standard construction environment includes several different mechanisms for performing customized builds. The available customizations range from building against a particular version of an entire package to compiling a single file of a library in a customized way. In general, the mechanisms have been designed so that the conceptual size of a customization is proportional to the extra instructions that must be written to produce it.

In general, the overrides supported by our standard environment can be divided into two classes: *general overrides* and *named overrides.* A general override is used to alter the standard environment by adding or replacing bindings in one or more locations. Since the standard environment is a naming hierarchy, a general override is a binding that defines (new) values for selected names in some portion of that hierarchy. A general override is applied by recursively overlaying the override at some point in the standard environment. A named override is a binding (possibly containing other bindings) that is interpreted like a table, with the binding names being the keys. Typically, the names specify the entities to which the override applies, such as a source filename or a library name. We will see examples of both general and named overrides below.

We now consider three forms of overriding supported by our standard construction environment.

Package Overrides. Perhaps the most common form of override involves overriding which version of a package to use in a build. It is not uncommon, for example, for a developer to be simultaneously working on a library and a client application of that library. In that case, the application model must specify that the latest checked out version of the library package should be used, rather than the one installed by the standard construction environment.

For packages imported by the `std_env` model, a *package override* is accomplished by passing a named override to the function for building the standard environment. Figure 5.9 shows an example of a control panel model with an override to use a checkout version of the *c/libc* library package. Compare this version of the model against the one shown in Figure 5.5 (page 83). The only difference is the extra `pkg_ovs` parameter passed to the `env_build` function. This parameter specifies that checkout version 7/1 of the *c/libc* package should be used.

Build-Wide Overrides. As its name implies, a *build-wide override* applies to an entire build. Such overrides are effected in the control panel model by applying

```
import
  self = build.ves;
  std_env =
    /vesta/west.vestasys.org/common/std_env/9/build.ves;
  libc =
    /vesta/west.vestasys.org/c/libc/checkout/7/1/build.ves;
{
  // bind the standard environment to '.'
  pkg_ovs = [ c/libc = libc ];
  . = std_env()/env_build("AlphaDU4.0", pkg_ovs);

  // build selected components
  b = self();
  return [ lib = b/lib(), progs = b/progs() ];
}
```

Figure 5.9: A control panel model that overrides the version of the *mail/index* package used in the build.

a general override to the current environment after the environment has been constructed, but before the package's own *build.ves* model is called.

For example, the following build-wide override might be specified in the control panel model:

```
  // build-wide override
  comp_switches = [ debug = "-g3", opt = "-O1" ];
  . ++= [ Cxx/switches/compile = comp_switches ];
```

This particular override causes all C++ files to be compiled with debugging symbols and optimization. The override applies to all programs built by this model, as well as to all of the libraries they import.

This example shows how parameters can be passed to bridge functions through the environment (i.e., "`.`"). Because the environment is passed implicitly on every function call (see Section 5.2.1 above), parameters stashed inside it are available to every (bridge) function. The interface to the C++ bridge specifies which parts of the environment are accessed by each of its functions.

Library Overrides. The standard construction environment also includes a mechanism for overriding how a particular library is compiled, or even how a particular file within a library is compiled. Both forms use named overrides, since they must name the library archive or source file, respectively, to which the override applies. In the case that the override applies to an umbrella library, the override also affects all descendants of the umbrella.

As in the case of build-wide overrides, library overrides are passed by recursively overlaying part of the current environment, typically in the control panel model. For example, a control-panel model could cause the libVestaBasics.a library to be built with debugging symbols as follows:

```
// library overrides
. ++= [ lib_ovs/libVestaBasics.a =
          [ ovs/Cxx/switches/compile/debug = "-g" ]];
```

Because the construction of each library is delayed until it is needed, each library is built according to the overrides *in force at the time the application using it* is built. Hence, different applications in a release can be built using different customizations of the same library.

5.4.5 Handling Large Scale Software

For Vesta to accommodate the construction of large scale software, the naming used by the standard construction environment must be designed to handle a large number of named artifacts. The standard construction environment addresses this problem by providing various hierarchical name spaces. There are two in particular worth mentioning.

First, the names used to specify packages in a package override are hierarchical, using a name space that parallels the repository's package name space. Notice that in the package override example of Figure 5.9 (page 89), the `pkg_ovs` parameter binds the two-level name `c/libc` to the new version of the package.

Second, the C/C++ bridge of the standard construction environment provides an option for naming libraries in library overrides using hierarchical names. We have already seen that umbrellas can be used to organize libraries hierarchically. So long as the number of libraries is small, a flat name space suffices to name them. But a flat name space is insufficient if the same name is used for two leaf libraries under different umbrellas. The bridge has a mode in which libraries are named hierarchically, thereby accommodating such cases. In this mode, a library is named in a library override by its *path* in the library hierarchy.

Chapter 6

Function Cache

Conceptually, every Vesta build is done from scratch, to ensure consistency. However, to achieve good performance, Vesta remembers the results of previous builds and reuses them whenever possible, thereby making all builds incremental. The Vesta function cache server is the component responsible for storing the results of previous builds and making them available for reuse. Since the same centralized function cache is shared by all developers at a site, developers share the benefits of each other's builds.

Recall that the Vesta system description language is functional, and that invocations of tools like compilers and linkers are represented in the Vesta language by function calls. The partial results of previous builds can thus be saved simply by caching the arguments and results of each function application. The function cache stores the arguments and result of each function call as a *cache entry*.

Although the function cache was designed particularly for Vesta clients (namely, the evaluator and weeder), its interface is general enough to support other clients, as we will see in Section 6.3 below. As one example of this generality, the cache keys stored by the function cache are arbitrary name/value pairs. It is the client's sole responsibility to compute these keys correctly so as to make proper use of the cache. We discuss the Vesta evaluator's caching algorithm in Chapter 7.

For the function cache to be of any use to a client, one obvious requirement is that the time required by the client to compute the necessary result in the event of a cache miss must far exceed the time required by the cache to produce a cache hit. This is certainly the case in Vesta, since some functions (such as external tool invocations) are relatively long running.

The rest of this chapter motivates the Vesta caching problem, describes the interface provided by the function cache server, lists some requirements that the server must satisfy, and sketches aspects of the server's implementation that help it to meet the requirements.

6.1 Dynamic, Fine-Grained Dependencies

The Vesta function cache is different from a normal memory cache in two important respects. First, the entries stored in the cache are immutable, so the cache does not have to support invalidation, a considerable simplification. Second, the complete cache key used to look up a cache entry is not directly computable from the function call site. This property substantially complicates the design, but it is inherent in any caching scheme in which the cache key is formed partly from dependencies that are computed dynamically, a topic we discuss below.

By using a cached function result whenever it is safe to do so, unnecessary recompilations and other work can be avoided. But when is it safe to reuse a cached result? Only when the evaluation context at a candidate call site agrees with those parts of the context on which some previous call to the same function depended.

When detecting dependencies, it is of course essential not to omit any, or the cache would be unsound, sometimes returning incorrect results. In particular, if the cache did not record sufficient dependency information for a particular function call, a subsequent function call might incorrectly return a hit on the previously cached result. We call this situation a *false hit*. It is an intolerable outcome that is indicative of a bug in the client's caching algorithm.

However, it is also important not to err by introducing overly *coarse-grained* dependencies, or the cache will be ineffective, sometimes failing to return a result when it should. We call this situation a *false miss*. False misses arise because the cache has recorded an unnecessary dependency on the calling context. If that aspect of the calling context is different on a subsequent call, the cache will report a miss, even though it would have been safe to use the cached result.

False misses do not lead to incorrect results, but they do introduce unnecessary inefficiency. In fact, false cache misses can be quite costly because they can trigger a cascade of subsequent cache misses. For example, if one source file is recompiled unnecessarily, then all subsequent commands that use the resulting object file (e.g., building a library that contains the file and programs that use the library) are likely to be unnecessarily repeated as well, as the object file will appear to have changed. Hence, we go to great lengths to avoid false cache misses.

For the cache to be most effective, each function call's dependencies must be recorded as precisely as possible. For example, consider the following simple function:

```
f(x, y, z) {
  return (if x > 0 then y else z);
}
```

Because of the conditional expression, the arguments on which this function depends vary *dynamically* from call to call. For example, in the call $f(1, 2, 3)$, the result depends only on the values of x and y; the value of z is irrelevant. Hence, the subsequent invocation $f(1, 2, 7)$ in which the values of x and y are identical should produce a cache hit on the first call.

In this particular example, the observant reader will notice that the exact *value* of x is also unimportant. What matters is simply whether or not x is positive. Hence, to get the most accurate caching, dependencies should take the form of *predicates* on values, not the exact values themselves. For now, we will assume that all dependencies are recorded on values, but we describe how to represent more general dependencies in the next chapter.

The caching problem is further complicated by composite value types, such as lists and bindings. For example, consider a slight modification to our previous example, in which the y argument is a binding:

```
f(x, y, z) {
  return (if x > 0 then y/a else z);
}
```

In this case, the result of the call $f(1, [a = 2, b = 5], 3)$ depends only on x and y/a. The subsequent call $f(1, [a = 2, b = 9], 7)$ should produce a cache hit. Recording a dependency on the entire binding y would cause the second call to get a false cache miss. This example demonstrates that the dependencies calculated with respect to composite values should be as *fine-grained* as possible.

From these two examples, we conclude that the Vesta evaluator must compute fine-grained dependencies dynamically (that is, during an evaluation). The challenges in accurately implementing a caching scheme based on dynamic, fine-grained dependencies are two-fold. First, algorithms must be developed for representing, computing, and propagating dynamic, fine-grained dependencies. We sketch the technique used by the Vesta evaluator in Chapter 7. Second, due to the dynamic nature of the dependencies, it is impossible to compute a single cache key at a function call site before the function has been evaluated. We discuss this problem next.

6.2 The Caching Problem

To understand the problem introduced by dynamic dependencies, consider the prototypical case of caching a compilation. Assume that a compilation is invoked by a call to the following function:

`compile(`*filename, options,* `.);`

Here, *filename* is the name of the file to compile, *options* is a binding denoting the command-line options to be passed to the compiler, and "." is the current environment.[1]

The current environment "." is a binding value that includes a representation of a file system directory tree. This directory tree contains all the files necessary to perform the compilation. As described in Chapter 5, the Vesta language makes it easy to construct and extend bindings, so a custom file system can be constructed quite cheaply for each build.

Figure 6.1 shows a sample environment. Two paths in "." are special: `root` and `root/.WD`. As described in Chapter 3, external tools such as compilers and linkers are run in an encapsulated environment in which all references to files are trapped by the repository and serviced by the evaluator. To service a file request from the repository, the evaluator looks up the file in the current environment: absolute pathnames are looked up in the `root` subtree, while relative pathnames are looked up in the `root/.WD` subtree.

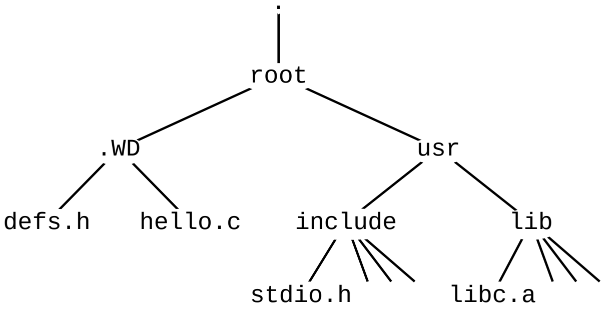


Figure 6.1: The file system directory trees of a sample environment.

Now consider the following invocation of the *compile* function:

```
compile("hello.c", [debug = "-g"], .);
```

Let "." be the binding shown in Figure 6.1. When invoked, this function returns the singleton binding that maps the name "`hello.o`" to the derived object file produced by compiling the file bound to `./root/.WD/hello.c`.

To effectively cache this call, the evaluator will compute dynamic, fine-grained dependencies. This function invocation might be found to depend on the definition of the *compile* function itself, the values of the first two arguments, and the following parts of the environment:

[1] As described in Section 5.2.1, the final "." parameter can be omitted, in which case it is passed implicitly.

```
./root/usr/lib/cmplrs/cc
./root/.WD/hello.c
./root/usr/include/stdio.h
```

Notice that even the reference to the compiler (cc) is trapped and recorded as a dependency. The cache entry for the compilation combines all these values into a cache key, and associates that key with the resulting value.

It is worth noting here what we mean by "combining" values together to form a cache key. Since the values used to form the cache key can be quite large, the evaluator and function cache use *fingerprints* of the values instead of values themselves. A fingerprint is a small, fixed-size hash of an arbitrary byte sequence [10, 42]. Fingerprints come with a mathematical guarantee bounding the probability of a collision; by choosing long enough fingerprints, the probability of a collision can be made vanishingly small.[2] As a result, fingerprints can be used as a basis for equality tests, since we can safely assume that $FP(a) = FP(b) \iff a = b$. Two operations supported on fingerprints are extending a fingerprint by more bytes and extending a fingerprint by another fingerprint. In the latter case, we write $fp_1 \oplus fp_2$ to denote the result of extending fp_1 by fp_2. The $\oplus$ operation is non-commutative.

Fingerprints have the advantage that they are small and can be easily combined without forfeiting their probabilistic guarantee. Storing just the fingerprints of values contributing to the cache key is sufficient because the only operations the function cache needs to perform on such values are combining them and comparing them for equality, both of which can be done just as well on fingerprints as on the full values. Result values, of course, must be stored in full so that they can be supplied to the evaluator in the event of a cache hit.

We now come to the main caching problem in Vesta: how can a lookup be performed on the cache when all of the dependencies are not known until after the function has been evaluated? Given the use of dynamic dependencies, it would seem that when the evaluator reaches a new function call site, it must evaluate the function before it will have the necessary key to look it up in the cache! Obviously the cache would then be useless. Alternatively, the evaluator could search linearly through the entire cache, looking at each entry's dependencies and checking whether their values at the call site match the entry. A cache designed this way would be too slow to be useful.

The solution to this chicken-and-egg problem lies in separating the cache key into two parts, *primary* and *secondary*. Lookup then becomes a two-step process. First, the evaluator computes the primary key in a fixed way solely from informa-

[2] For safety, Vesta uses 128-bit fingerprints. Using the numbers in Section 3.3 and other conservative estimates, we compute that the probability of a collision occurring over the expected lifetime of the Vesta system is much less than 2^{-42}.

tion available at the call site. A cache lookup using this primary key yields a small number of candidate cache entries. Next, the secondary key of each candidate entry is checked to see if it matches the values at the call site.

In more detail, we divide the dependencies into two groups: those that are known statically at the function call site, which we call *primary dependencies*, and those that are only known dynamically when the function is evaluated, which we call *secondary dependencies*.[3] Since primary dependencies are known at the call site, they can be used to compute the primary key and reduce the number of cache entries that must be considered during a cache lookup operation.

The primary key (PK) is formed by combining the fingerprints of the primary dependency values. Each secondary dependency consists of a name and the fingerprint of the corresponding value. Together, these secondary dependency names and fingerprints form the cache entry's secondary key (SK). Overall, then, a cache entry is a triple of the following form:

$$\langle \text{ primary-key, secondary-key, result-value } \rangle$$

Here, the primary key is a single fingerprint, the secondary key is a set of (name, fingerprint) pairs, and the result value is the function's full result value, suitable for use by the evaluator in the event of a cache hit.

Figure 6.2 shows the primary and secondary keys computed for the example compilation above. First, the fingerprint Q of the *compile* function itself (a closure value) and the fingerprints R and S of the first two function arguments are computed. These fingerprints are then combined to form a new fingerprint A, the primary key. When the function is evaluated, references to "." are trapped and the names and fingerprints B, C, and D of the corresponding values are recorded as secondary dependencies. The cache entry formed for this function evaluation is a triple consisting of the primary key, secondary dependency names and fingerprints, and the result value of the evaluation.

It is common for multiple cache entries to have the same primary key. In particular, this occurs whenever a source file is edited and recompiled. Figure 6.3 shows an example. In that figure, the two columns of fingerprints on the right denote two different cache entries. Both cache entries correspond to the compilation of a file named "`hello.c`" with the same compilation switches, so both entries have the same primary key A. However, between the two compilations, the source file "`hello.c`" has been edited, so the fingerprint for the corresponding secondary dependency has changed from C to E. Since the secondary dependencies for the two evaluations are different, two different entries are stored in the cache.

[3] As described in Section 7.4.1, the Vesta evaluator uses heuristics and user-supplied pragmas to distinguish primary from secondary dependencies when caching calls to user-defined functions.

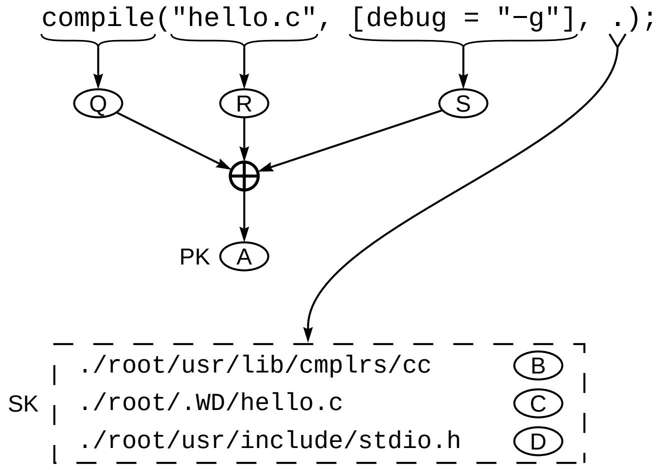

compile("hello.c", [debug = "-g"], .);
Q
R
S
PK
A
./root/usr/lib/cmplrs/cc
B
SK
./root/.WD/hello.c
C
./root/usr/include/stdio.h
D

Figure 6.2: The primary key (PK) and secondary key (SK) of a single cache entry.

		Entries	
		0	1
	Primary Key (PK)	A	A
SK	./root/usr/lib/cmplrs/cc	B	B
	./root/.WD/hello.c	C	E
	./root/usr/include/stdio.h	D	D

Figure 6.3: Two cache entries with the same primary key (PK).

An important consequence of using dynamic fine-grained dependencies is that the set of secondary dependency names may differ from one cache entry to the next, even among entries with the same primary key. An example is shown in Figure 6.4, where cache entries 1 and 2 have different sets of secondary dependency names because the file "`hello.c`" was edited to include a file named "`defs.h`" instead of "`stdio.h`". The difference between entries 2 and 3 is that the file "`defs.h`" was changed.

Figure 6.4 also demonstrates an important property of the Vesta caching strategy. For Vesta caching to be correct, all user-defined functions and external tools must be functional and deterministic: the same (possibly dynamic) inputs should always produce the same output[4]. As a result, cache entries like the one labeled

[4] This requirement is stated more strictly than is actually necessary. For example, some compilers embed a timestamp in the object files they generate, so strictly speaking, the contents of generated

		Entries				
		0	1	2	3	X
	Primary Key (PK)	A	A	A	A	A
SK	./root/usr/lib/cmplrs/cc	B	B	B	B	B
	./root/.WD/hello.c	C	E	F	F	F
	./root/usr/include/stdio.h	D	D			J
	./root/.WD/defs.h			G	H	H

Figure 6.4: Multiple cache entries with the same primary key (PK), but with differing sets of secondary dependency names.

“X” in the figure should not be created. The fingerprints of cache entry X agree with those of cache entry 3 for those secondary dependencies shared by the two entries, but entry X has an additional secondary dependency on the file “`stdio.h`”. Functional tools like compilers never produce such entries, and neither do user-defined functions written in the Vesta language.

6.3 Interface

Having described primary and secondary dependencies, we are now prepared to present the two-step cache lookup process in detail. Figure 6.4 helps illustrate how lookups in the cache are performed. Consider the function call from our running example:

```
    compile("hello.c", [debug = "-g"], .);
```

To test whether this function call produces a hit in the cache, we first compute the primary key from the call site as shown in Figure 6.2. We then consider all cache entries with the computed primary key. For each such entry, we compare the secondary dependency fingerprints against the fingerprints of the corresponding values in the current evaluation context. If the secondary dependency fingerprints of any cache entry match those of the evaluation context, we have found a hit. Otherwise, we report a miss.

The protocol for doing a cache lookup is thus a two-step process. One round-trip to the function cache is required for each step.

1. In the first step, the evaluator computes the primary key from the function call site, and communicates it to the function cache. In response, the cache

object files depend on when the compiler is run. However, such embedded timestamps are semantically irrelevant to the use of the object files, so it is safe to treat the compiler as a functional tool.

returns the *union* of all secondary dependency names associated with cache entries having the designated primary key. In our running example, the secondary dependency names are the four pathnames shown in Figure 6.4.

2. In the second step, the evaluator computes (the fingerprints of) the values associated with those secondary dependency names in the current evaluation context, and sends them to the function cache. The function cache then compares those fingerprints to each of the entries with the designated primary key, looking for a match between the fingerprints from the evaluation context and those in some cache entry. In our running example, the received fingerprints would be compared to the columns of Figure 6.4; if any column matches, a cache hit is reported.

Between these two steps, another client may have added a new entry that includes some new secondary dependency names. To avoid reporting a false miss on the new entry, the cache uses an optimistic concurrency control scheme: the second step may return a result indicating that the names returned from the first step are stale, in which case the client is expected to restart the lookup process from step one. This scheme eliminates the need to lock the cache against updates during a lookup operation.

In the event of a cache miss, the evaluator will proceed to execute the function, recording dynamic fine-grained dependencies along the way. It will then form the primary key, secondary key, and result value for a new cache entry, and call a method of the function cache interface for adding this new entry to the cache.

In addition to the main interface that supports performing cache lookups and adding new entries to the cache, the function cache also exports an interface used solely by the Vesta weeder. This interface allows the weeder to indicate when it is beginning its work, and when it has determined the full set of cache entries to keep. Between these two calls, the cache disables the expiration of leases on cache entries (described below) to prevent newer entries from being inadvertently deleted.

6.4 System Requirements

In addition to implementing the interfaces described in the previous section, there are several other requirements on the function cache. This section highlights some of the function cache's main system requirements.

- It must store cache entries persistently; properly shutting down and restarting the server must not cause any cache entries to be lost. Persistence is achieved by storing older cache entries in disk files and using a combination of logging and checkpointing for newer cache entries. For efficiency, log entries are kept in memory; the interface to the function cache includes a method for synchronously flushing all pending log entries to disk.

- It must be fault-tolerant. In the event of a crash or other failure, the function cache must be able to recover in a consistent state. In particular, the various log files and cache entry files must be flushed in an order that allows the recovery algorithm to tolerate failure at any time. Although some recently created cache entries may be lost in a crash, only a relatively small amount of work has to be repeated to recreate them.

- It must support fast lookup. Since disk reads dominate the time required to do a lookup, the format of cache entries on disk has been designed so most lookups can be performed with at most two disk reads. Many lookups result in hits on in-memory cache entries, which can be serviced without any disk reads.

- It must service multiple clients concurrently. The server is thread-safe, and it includes fine-grained locks to avoid excessive lock contention.

- It must support concurrent weeding (the deletion of unwanted derived files and cache entries) without affecting clients adversely. Supporting concurrent weeding is essential because the weeder can take a long time to run, and because deleting cache entries is also time consuming. Once we allow for concurrent weeding, an immediate problem arises: the set of cache entries to keep will be computed by the weeder based on some “snapshot” of the cache, so an extra mechanism is necessary to protect all cache entries created since the snapshot was taken.

- It must be tolerant of client errors and client failures. For the former, the function cache interface includes run-time checks of its arguments with provisions for an error return code. For the latter, the function cache protects newly created and recently looked-up cache entries from weeding by *leases*. A lease is a “fresh” bit that protects a cache entry from weeding, but leases expire after a fixed amount of time. Hence, a cache entry created by a client that crashes can eventually be weeded, since the lease protecting the cache entry will eventually time out.

- It must scale. In particular, the function cache has been designed to store up to tens of millions of cache entries. To accommodate that number of cache entries, a two-level memory hierarchy is used. New and recently looked-up cache entries are cached in memory, while others are stored only on disk.

Within each disk file, cache entries are stored in a three-level hierarchy described below.

These requirements substantially complicate the function cache's implementation. It would be tedious to go into complete detail here on everything the implementation does to meet the requirements. Instead, the next section describes only the most important aspects of the implementation, focusing on information that will be needed as background to understand the performance measurements given in Chapter 9.

6.5 Implementation

Because step 1 of cache lookup narrows the search down to entries that match a given primary key, it is most efficient for the cache server to organize its cache entry storage by primary key. Both in memory and on disk, the server stores entries in groups called *PKFiles.* Each PKFile holds all the entries that have a particular primary key.

If the cache lookup algorithm worked exactly as outlined in Section 6.3, its second step would have to search through all the cache entries in the relevant PKFile. For each such entry, it would have to perform a fingerprint comparison for each of the entry's secondary dependencies. Both the number of cache entries in a PKFile and the number of secondary dependencies in a cache entry can be quite large, on the order of hundreds. Hence, if this naive lookup algorithm were used, lookups would require tens of thousands of fingerprint comparisons in the worst case, and hence, would be quite expensive.

To avoid this problem, cache entries are organized in a multi-level hierarchy. First, all entries with the same primary key are grouped together. Then the entries in each group are partitioned in such a way that only a subset of the entries in each group need be examined on any lookup.

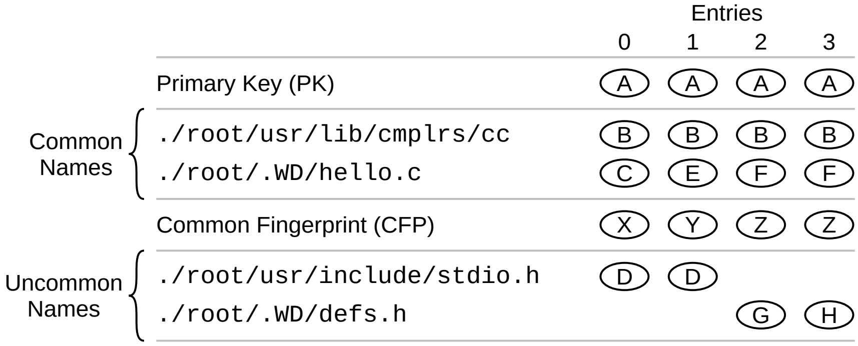
To explain how this partitioning is done, we introduce some notation. For any cache entry e, let $e.pk$ denote e's primary key, let $e.names$ denote the set of names in e's secondary key, and for any name $n \in e.names$, let $e.val(n)$ denote the fingerprint value associated with n in e's secondary key. Now define the following:

$$\begin{aligned}
\mathit{Entries}(pk) &= \{e \mid e.pk = pk\} \\
\mathit{AllNames}(pk) &= \bigcup_{e \in \mathit{Entries}(pk)} e.names \\
\mathit{CommonNames}(pk) &= \bigcap_{e \in \mathit{Entries}(pk)} e.names
\end{aligned}$$

$$CFP(e) = \bigoplus_{n \in CommonNames(e.pk)} e.val(n)$$

The set $CommonNames(pk)$ thus consists of those names that occur in every cache entry with the given primary key. The names in $AllNames(pk) \setminus CommonNames(pk)$ occur in some cache entries with the given primary key but not others; we call them *uncommon.* The value $CFP(e)$, e's *common fingerprint*, is the result of combining the fingerprints of all secondary values of e corresponding to $e.pk$'s common names. Due to the non-commutative nature of the $\oplus$ operation, it is important to enumerate the names n in a well-defined order. To this end, the function cache maintains a canonical ordering for each pk of the names in $AllNames(pk)$; it uses that ordering when computing $CFP(e)$.

Given these definitions, we can now describe how the cache implements both steps of the lookup algorithm. For each primary key pk, the cache maintains $Entries(pk)$, $AllNames(pk)$, and $CommonNames(pk)$ (the last of which is represented by a bit vector with respect to $AllNames(pk)$). In step 1 of the lookup algorithm, the cache simply returns $AllNames(pk)$ for the supplied primary key. To efficiently perform step 2 of the lookup algorithm, the cache maintains $CFP(e)$ for every entry e. It then groups the entries $Entries(pk)$ into equivalence classes according to their common fingerprints. For example, Figure 6.5 shows the common names and common fingerprints for the cache entries of Figure 6.4.



		Entries			
		0	1	2	3
	Primary Key (PK)	A	A	A	A
Common Names	./root/usr/lib/cmplrs/cc	B	B	B	B
	./root/.WD/hello.c	C	E	F	F
	Common Fingerprint (CFP)	X	Y	Z	Z
Uncommon Names	./root/usr/include/stdio.h	D	D		
	./root/.WD/defs.h			G	H

Figure 6.5: The common names and common fingerprints for the cache entries of Figure 6.4.

The PKFile is then partitioned by CFP value; two entries are in the same partition if and only if they have the same CFP. Figure 6.6 shows the hierarchy that results from the cache entries and common fingerprints shown in Figure 6.5. As shown in this figure, multiple entries appear in the same CFP group only when one of the uncommon secondary dependencies has changed.

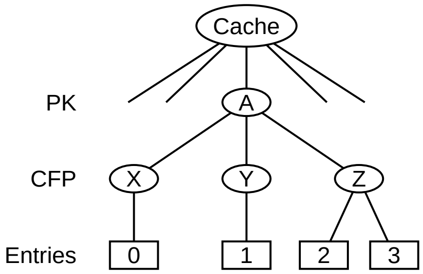


Figure 6.6: The hierarchical division of cache entries first by primary key (PK) and then by common fingerprint (CFP) for the cache entries of Figure 6.5.

In step 2 of the lookup operation, the cache is given a *pk* and the fingerprints $f_1, f_2, \ldots, f_k$ of the values corresponding to the names *AllNames*(pk) at the call site. Call these fingerprints f_i the *call site fingerprints.* To perform the lookup, the cache first combines those call site fingerprints associated with *CommonNames*(pk), thereby producing a common fingerprint *cfp*. Next, the cache does a hash table lookup to see if *pk* has *cfp* as one of its associated common fingerprints. If not, the cache reports a miss. Otherwise, the cache examines the entries in the identified CFP group. In testing for a hit, only the fingerprint values associated with the uncommon names need be examined, since by virtue of being in the correct CFP group, all entries being considered are known to have matching values for the common names.

The performance of this lookup algorithm depends on the distribution of CFPs within a PKFile and on the PKFile's relative number of common and uncommon names. In the case of a compilation, the file being compiled will be one of the common names. Hence, each time a new version of a file is compiled, a new common fingerprint is produced. The expectation, then, is that within a PKFile, there will be many CFPs, but each CFP grouping will contain relatively few entries. This means that the number of cache entries considered during a lookup is expected to be small.

In addition, the lookup algorithm requires fewer fingerprint comparisons, and therefore performs better, the smaller the number of uncommon names per cache entry. In the style of usage we envisaged when designing the cache, we expected nearly all PKFiles to have few uncommon names compared to the number of common names. In the compilation example we have been considering, this expectation is not unrealistic, since the set of files read during the compilation of a particular source file does not tend to change much from version to version. This has proven out so far in practice, as shown by the distribution of PKs, CFPs, and common and uncommon names in an actual cache reported in Section 9.4.2.

In summary then, this lookup algorithm results in two major cost savings compared to the brute-force algorithm. First, only those entries in the identified CFP group need be examined. Second, when examining the entries in a CFP group, only the values associated with the uncommon names need be examined. These two effects thus drastically reduce the number of fingerprint comparisons required to perform a cache lookup.

The function cache was designed to store on the order of tens of millions of cache entries. Hence, storing all of the cache entries in memory is impractical. Instead, a two-level memory hierarchy is used. Most of the entries are stored in stable PKFiles on disk, and the newly created cache entries and those cache entries on which a hit has recently occurred are stored in volatile PKFiles in memory. Stable PKFiles are stored in a normal file system provided by the underlying operating system.

To avoid the disk fragmentation that would otherwise occur from creating one disk file per stable PKFile, the stable PKFiles are grouped together into so-called *MultiPKFiles* on disk. Two PKFiles are stored in the same MultiPKFile if and only if their corresponding primary keys have the same 16-bit prefix.[5] Since the primary keys are essentially random, the PKFiles tend to be evenly distributed among the MultiPKFiles.

To look up a cache entry, the cache locates the appropriate volatile PKFile, and first looks for the entry in memory. If a hit is not found there, it opens the appropriate stable MultiPKFile, and seeks to the start of the appropriate PKFile. The stable PKFile header contains a table mapping common fingerprint values to positions in the PKFile where the entries with each CFP are stored. The CFP table is stored in sorted order by CFP so interpolated binary search can be used for CFP lookup.

Most cache lookups that go to disk require two disk reads: one to read the CFP table and one to read the entry in the event of a hit. In the typical case in which no matching CFP is found, only one disk read is required. Depending on the filesystem block size, the filesystem read-ahead algorithm, and the number of CFPs in the PKFile, some cache hits can actually be serviced in a single disk read. However, more than two disk reads may be required to service a cache hit if the CFP table or the cache entry being read is larger than the filesystem block size. Section 9.4.1 reports the time required to perform cache lookup operations with various outcomes.

Maintaining this cache organization is complicated by the addition or removal of cache entries. When a new entry is added to or removed from the set *Entries*(pk),

[5] This prefix size is a compile-time constant, and can be easily changed.

the sets *AllNames*(*pk*) and *CommonNames*(*pk*) can change. Since changes to the latter would require common fingerprints to be recomputed and cache entries to be reorganized, newly added entries are kept in two side buffers (one for entries that have all of the common names, and one for those that do not). The lookup algorithm must consult these side buffers in addition to checking for a hit in *Entries*(*pk*). Once a large enough number of new entries are collected, they are merged into *Entries*(*pk*) together, thereby amortizing the high cost of recomputing all of the common fingerprints and shuffling cache entries among CFP groups. Deletions triggered by the weeder are handled similarly; again, new common names and common fingerprints are not computed for every deletion, but only once for each PKFile as a batch.

The function cache uses mutexes to protect its shared data structures. One centralized mutex protects the main cache variables. To reduce lock contention, a separate mutex protects access to the cache entries and other state of each volatile PKFile. The use of a separate mutex for each volatile PKFile allows lookups on different primary keys to proceed in parallel. When a PKFile is rewritten, its mutex is held only as long as necessary; for example, the mutex is not held while the stable PKFile is read and while its common names and the common fingerprints of its entries are recomputed. This locking strategy allows most of the considerable work required in rewriting a PKFile to occur in parallel with lookups and the addition of new entries, thereby improving the cache's overall performance.

Chapter 7

Evaluator

The Vesta evaluator interprets system models written in the Vesta system description language. It interacts with the function cache described in Chapter 6 to achieve higher performance through incremental building. Writing an interpreter for the Vesta language is straightforward, but exploiting the function cache effectively is not. This section describes the evaluator's caching strategy, as well as its solutions to some practical problems that arise due to caching.

7.1 Overview

In a typical Vesta build, the expensive operations are applications of the language's `_run_tool` primitive, which cause external tools to be invoked. Hence, effectively caching `_run_tool` invocations is crucial to achieving good incremental build performance. The relatively high cost of invoking external tools means that the evaluator can afford to spend time computing fine-grained dependencies in order to increase the cache hit rate on such calls.

Although caching `_run_tool` calls is a good start, such caching alone does not scale well. As discussed in Chapter 3, it is important for the running time of an incremental build to be proportional to the size of the change, not to the size of the system being built. Building a large software system from scratch might entail thousands or tens of thousands of tool invocations. If we assume that a cache lookup takes 10–20 milliseconds, an incremental build using only `_run_tool` call caching would take tens of seconds to rebuild such a system, even if it were completely up to date. In fact, this is precisely the behavior exhibited by Make, whose "cache lookups" are file timestamp comparisons on the input and output files of tool executions. The time required for these lookups increases linearly with the number of files involved in a build, which is the primary reason Make does not scale well.

Therefore, it is important to cache parts of the build larger than individual tool invocations. Vesta's functional modeling language provides a natural basis for such caching. Each function call in a build's call graph can be cached. During a build, whenever the evaluator encounters a function call, it consults the function cache to see if the same function has previously been evaluated on sufficiently similar arguments. If so, the evaluation of that function *and all of its descendants in the call graph* is skipped. Because the language is functional, there are no side effects to be reproduced; the evaluator simply takes the previous call's result value from the cache and uses it as the result of the new call.[1]

As described in Section 6.1, using dependencies that are too coarse-grained can lead to costly false cache misses. To avoid false cache misses, the evaluator attempts to compute dependencies on only those parts of the evaluation context that are relevant to the function being evaluated. These dependencies are computed *dynamically*, that is, as the evaluator is interpreting a function call.

A dependency can be thought of as a predicate on the evaluation state that is required to produce a cache hit. When viewed in this way, a dependency that is too coarse-grained corresponds to a predicate that is unnecessarily strong. The goal of the evaluator is to record dependencies that are as weak as possible, but strong enough to capture the relevant aspects of the state so as to produce correct cache hits in the future. As described earlier, we call such dependencies *fine-grained.*

The dependencies recorded by the evaluator are not always the weakest possible predicates. Sometimes, unnecessarily strong dependencies are used intentionally because they are more compact and therefore easier to manipulate and check. There is a tradeoff between the strength of the dependency and the cost of getting a cache miss. In practice, we have balanced the tradeoff so that Vesta's performance rarely suffers due to unnecessarily strong dependencies. In any event, the evaluator never records dependencies that are too weak. If it did, false cache hits would be possible, and incorrect builds would result.

When caching `_run_tool` calls, the evaluator records the tool's file system references as fine-grained dependencies. Several different *types* of dependencies are recorded, each corresponding to a different kind of file system operation, and each representing a different predicate on the evaluation state. For example, it is essential for correctness that all attempts to open non-existent files are recorded. The operation of listing a directory is another kind of dependency that must be recorded. Each of these differs from the dependency created by successfully open-

[1] The language's primitive functions other than `_run_tool` are inexpensive, and therefore are not cached. By default, all other calls are cached. However, the language also includes a stylized comment, or *pragma*, to suppress caching of particular functions, which is occasionally useful in build performance tuning.

ing and reading a file.

Caching invocations of user-defined functions (i.e., closures) is more complex than caching invocations of tools. More kinds of operations are available on data structures in the language than are available on an encapsulated file system, making it desirable to use more precise dependency predicates. In fact, the main challenge in implementing the Vesta system description language lies in computing fine-grained dependencies for the language's many operations, in order to make effective use of the function cache.

Although caching user-defined functions leads to significant performance gains, it does not solve the scaling problem completely. When one function calls another, most dependencies of the callee typically become dependencies of the caller. Hence, if no special measures were taken, the root function of an evaluation would have an extremely large number of dependencies. Obviously, that situation could prevent Vesta from scaling well. To prevent too many dependencies from being propagated up to the root of an evaluation, we exploit the special properties of models (as distinct from general closures) to allow for an efficient, coarse-grained representation of the dependencies associated with model evaluations. Models thus serve as cutoff points in the call graph, beyond which overly fine-grained and bulky dependencies do not propagate. We explain this matter fully in Section 7.5 below.

The rest of this chapter is organized as follows. The next section briefly describes how the evaluator represents dependencies. The three sections after that describe the evaluator's algorithm for creating cache entries for the three classes of functions enumerated above, namely, the `_run_tool` primitive, user-defined functions, and models. Section 7.6 then shows the effect of the evaluator's caching techniques on real builds. The final section describes how the evaluator handles errors.

7.2 Representing Dependencies

As mentioned earlier, dependencies in general are predicates on the evaluation context. In a practical implementation, however, allowing for arbitrary predicates would be costly in both space and time. We therefore use a small, fixed collection of predicates, encoded as *dependency paths*. It is these dependency paths that are passed to the function cache as the names in a cache entry's secondary key.

The syntax of a dependency path is given by the following grammar:

$$\begin{array}{rcl} \mathit{dpath} & ::= & t : \mathit{path} \\ t & ::= & V \mid X \mid D \mid T \mid L \mid E \\ \mathit{path} & ::= & \epsilon \mid \mathit{id}/\mathit{path} \end{array}$$

A dependency path takes the form $t : path$, where t denotes the dependency type, and *path* specifies the component of the evaluation context on which the evaluation depends. We use ϵ to denote an empty path; a path of the form id/ϵ or ϵ/id is equivalent to the path *id*.

The *type* is a single-character code that selects a predicate on the path's value that must be satisfied for the dependency to match; we describe the meanings of the six dependency types below. The *path* is a pathname that can be looked up in the context of the function call being evaluated. It is not necessarily a path expression that would be valid to write at that point in the model. For example, if *a* is a binding, the path *a*/*b* would represent the field *b* in *a*, but if *a* is a closure, *a*/*b* would represent the value of *b* in *a*'s context.

Associated with each dependency path is a value whose fingerprint is passed to the cache on a lookup operation or when a new cache entry is created. The rules for computing the value associated with a dependency path vary depending on the dependency type and the kind of function being evaluated. These rules are described in more detail below.

7.3 Caching External Tool Invocations

Caching `_run_tool` calls is fairly straightforward. The primary key for the cache entry is formed by combining a fixed fingerprint for the `_run_tool` primitive with the fingerprints of all of the function's arguments except the implicit "`.`" argument. As described in Section 5.2.3, the tool's file system is supplied via the binding `./root`. Obviously, recording a dependency on the complete file system represented by `./root` would be too coarse-grained, since it is very unlikely that the tool will access all the files in that directory tree. Hence, the cache entry's secondary key is formed from the file system references made by the tool during its execution. As described in Chapter 3, the tool's file system references are trapped by the repository, sent to the evaluator, and satisfied by the evaluator by doing lookups in `./root`.

The evaluator must handle two different kinds of requests from the repository, and it records three different types of secondary dependencies based on the kind of request and its outcome. The two types of requests are *lookup* (looking up a name in a binding that represents a filesystem directory) and *list* (listing the entries in such a binding). Table 7.1 lists the three types of dependencies that the evaluator creates in response to repository requests during a tool execution. We give the details in the following paragraphs.

When a name lookup succeeds and returns a file, the evaluator passes the file's shortid back to the repository and records a *value* (V) dependency. The depen-

Operation	Dependency Path	Value
lookup(*dir*, *name*) $\Rightarrow$ success	$V : dir/name$	Value(*dir*/*name*)
lookup(*dir*, *name*) $\Rightarrow$ failure	$X : dir/name$	FALSE
list(*dir*)	$D : dir$	domain(*dir*)

Table 7.1: The three dependency types recorded by the evaluator during external tool invocations.

dency's associated value is the contents of the file; however, recall that the evaluator and cache actually use the fingerprint of the value in the dependency, not the value itself. In this case, the fingerprint for the file is supplied by the repository (see Section 4.2.3).

When a name lookup succeeds and returns a directory, the evaluator passes a handle for the directory back to the repository. The tool now knows that the directory exists, so in general the evaluator must record a dependency reflecting this fact; namely, an *existence* (X) dependency on the directory's pathname, with TRUE as its value. However, in the most common case, these X dependencies can be optimized out: if the tool subsequently attempts to look up a name in the directory, the evaluator will generate a dependency whose path has the directory's path as a prefix. Such a dependency subsumes the X dependency for the directory itself, so the evaluator avoids recording the latter.

When a name lookup fails, the evaluator returns a "not found" result to the repository and again records an existence (X) dependency on the path. In this case the path is not defined, so the dependency's associated value is FALSE.

Finally, if the evaluator receives a request to list a directory, it records a *domain* (D) dependency. To obtain the dependency's value fingerprint, the evaluator combines the fingerprints of the names defined by the corresponding binding, in order. The values bound to those names are not considered in the fingerprint. This procedure reflects the semantics of listing a directory: the result depends only on what names are defined (and their order), not on what the names are bound to.

Table 7.2 shows some of the secondary dependencies that might be recorded for a simple compilation of a file named "hello.c". The fingerprints in the Fingerprint field are those supplied by the repository for the corresponding files, except of course for FP(FALSE).

As mentioned previously, it is important for correctness to record a FALSE existence (X) dependency when a lookup request fails. For example, consider the following common situation. A compiler typically searches several directories for header files. Assume that on one compilation, a header file was found in the second such directory but not the first. If on a subsequent compilation a file of

Type	Path	Fingerprint
V	`./root/usr/lib/cmplrs/cc`	FP(`cc`)
V	`./root/.WD/hello.c`	FP(`hello.c`)
X	`./root/.WD/stdio.h`	FP(FALSE)
V	`./root/usr/include/stdio.h`	FP(`stdio.h`)
$\vdots$	$\vdots$	$\vdots$

Table 7.2: Some of the secondary dependencies for a simple compilation of the file “hello.c”. *FP* denotes the fingerprint function.

the same name existed in the first directory, a false cache hit would result on the first compilation unless a `FALSE` existence dependency had been recorded for the filename in the first directory during the first build.[2] Such false cache hits are intolerable, since they can lead to inconsistent builds.

7.4 Caching User-Defined Function Evaluations

Computing fine-grained dependencies for calls to `_run_tool` is relatively straightforward, since they are limited to file system accesses. But handling the general case of caching a call to a user-defined function written in the Vesta language is quite challenging, and proved to be the most difficult part of the evaluator design and implementation. This section gives the mathematical rules we have developed for computing dependencies during the evaluation of user-defined functions. It also states (but does not prove) a correctness theorem.

To describe the key ideas used in our dependency calculation, we define a subset of the Vesta modeling language. Table 7.3 gives the subset language’s syntax. Here, *Literal* is the set of possible literals (constants), and *Id* is the set of possible identifiers (variables). This subset has been chosen to include the core parts of the Vesta language that present the most significant challenges to effective dependency analysis. It omits the language’s primitive functions (of which there are approximately 60), its iteration construct, its provisions for importing one system model from another, and its support for binding program variables to versioned directories and files in the Vesta repository.

Every expression is evaluated in some *evaluation context*, which is a mapping from variable names to values. Let $Eval(e, c)$ denote the result of evaluating the

[2] Some build systems do not handle this issue correctly. For example, ClearCASE records dependencies only on existing files. Makefiles generated by the Unix makedepend tool have this problem as well. In both cases, this deficiency can lead to inconsistent builds.

e ::=	
$\mid a$	literal
$\mid x$	variable
$\mid \lambda x.e$	lambda
$\mid \underline{\text{if}}\ e_1\ \underline{\text{then}}\ e_2\ \underline{\text{else}}\ e_3$	conditional
$\mid [n_1 = e_1, n_2 = e_2, ..., n_k = e_k]$	binding constructor
$\mid e/n$	binding selection
$\mid e!n$	binding domain test
$\mid e_1 + e_2$	binding overlay
$\mid \underline{\text{let}}\ x = e_1\ \underline{\text{in}}\ e_2$	let construct
$\mid e_1(e_2)$	function application

Table 7.3: The syntax for a subset of our system modeling language. Here e represents an expression, a a literal, x an identifier, and n a name.

expression e in the context c, and let $Dpnd(e, c)$ denote the dependency information resulting from evaluating e in c. Since we use standard call-by-value evaluation, the rules for $Eval(e, c)$ are straightforward, so we will not describe them here.

7.4.1 Computing Primary and Secondary Cache Keys

When caching the evaluation of a user-defined function, how are the primary and secondary cache keys computed? Clearly, the primary key should contain some representation of the function itself, since no cache hits should occur on old entries if the function definition has changed. In fact, the evaluator uses the fingerprint of the function's parse tree as the basis for the primary key.

What about the function's arguments? As we have seen, folding all of the arguments into the primary key would produce cache entries that are too coarse-grained. However, not including any of the arguments in the primary key would produce many cache entries with the same primary key. That would cause cache lookups to take longer, since there would be more entries to search through for any given primary key — one for each time the function was invoked.

The evaluator chooses a course between these two extremes. It uses a heuristic that folds the fingerprints of simple arguments (i.e., booleans, integers, and texts) into the primary key, but performs fine-grained dependency analysis on the composite arguments (i.e., bindings, lists, and closures). That is, all dependencies on values not captured by the primary key are included in the secondary key. The Vesta language includes pragmas for overriding this heuristic, but they are rarely needed. They are used mainly by writers of bridges to tune the system for better cache performance; in any event, their presence or absence cannot produce false cache hits.

7.4.2 Relation to Caching

As described in the previous section, when faced with a function invocation, the evaluator first computes a primary key from the function's body and zero or more of the argument values. In response to this primary key, the function cache returns a set of secondary names. The cache treats the secondary names as meaningless strings, but to the evaluator they represent dependencies. The evaluator computes the values of each of the secondary names in the current context, fingerprints them, and sends the resulting list of value fingerprints to the cache. The cache then tests for a hit as previously discussed. In the event of a cache hit, the evaluator uses the cached result value.

In the event of a cache miss, the evaluator proceeds to evaluate the function on the given arguments. As it does so, it represents each runtime value by a pair containing the true value and the dependencies detected while computing that value. Value-dependency pairs are also stored for any sub-values nested inside composite values such as lists or bindings. Once it has finished evaluating the function, the evaluator collects up the dependencies in the function's result value. Any dependencies that are not already part of the primary key are considered part of the secondary key. The evaluator then calls the function cache to create a new cache entry with the computed primary key, secondary key, and result value. It is important to note that the cached result value, like all of the evaluator's runtime values, itself is a pair that includes dependencies. Whenever a later evaluation gets a hit on this entry, both the value and its dependencies will be needed so that the dependency analysis for subsequent uses of the value can proceed correctly.

For caching to be correct, the cache entries created by the evaluator must satisfy a theorem, a formal statement of which is given in Section 7.4.6 below. To understand the intuition behind the theorem, imagine that the cache contains an entry with primary key *pk* that resulted from evaluating the expression *e* in a context c_1. The secondary names associated with this cache entry will be the set of dependencies $\mathit{Dpnd}(e, c_1)$.

When attempting to evaluate *e* in another context c_2, the cache will produce a hit on the cached entry only if the values computed in c_2 for the dependencies $\mathit{Dpnd}(e, c_1)$ match the values stored in the cache. Note that the values associated with the dependencies in the cache are precisely the values computed in c_1 for $\mathit{Dpnd}(e, c_1)$.

We therefore define *equivalence* between two contexts with respect to a set of

	Dependency on	
Type	the component's . . .	Computed Value
V	. . .complete value	$Value(V : path, c) = Eval(path, c)$
X	. . .existence	$Value(X : path/id, c) = Eval(path!id, c)$
D	. . .domain	$Value(D : path, c) = \{n \mid Eval(path!n, c)\}$
T	. . .type	$Value(T : path, c) = Eval(typeof(path), c)$
L	. . .length	$Value(L : path, c) = Eval(length(path), c)$
E	. . .expression (closures)	$Value(E : path, c) = Eval(path, c).body$

Table 7.4: The meanings of the six dependency types and the rules used by the evaluator to evaluate each type of path in a context c.

dependencies d as follows:

$$Equiv(c_1, c_2, d) = (\forall p \in d : Value(p, c_1) = Value(p, c_2)).$$

In this definition, $Value(p, c)$ denotes the result of "evaluating" the dependency p in the context c. It corresponds to the fingerprint associated with each secondary name in the cache. (We define $Value(p, c)$ for the particular dependencies created by the evaluator in the next section.)

Clearly, the cache hit will be correct if and only if performing the evaluation of e in c_2 would produce the same result as the one stored in the cache, that is, $Eval(e, c_2) = Eval(e, c_1)$. Therefore, to prove our caching correct, we must show that

$$Equiv(c_1, c_2, d) \Longrightarrow Eval(e, c_2) = Eval(e, c_1).$$

After describing our dependency calculation algorithm, we give a formal statement of this requirement in Section 7.4.6 below.

7.4.3 Dependency Types

Table 7.4 presents the meanings of each of the evaluator's six dependency types during a user-defined function evaluation, along with the rules it uses to compute $Value(t : path, c)$, the value of the dependency path $t : path$ in the context c. In this table, the language's primitive *typeof* and *length* functions compute the dynamic type of a value and the length of a list or binding, respectively. The notation *cl.body* denotes the closure *cl*'s body component.

The V (value) dependency type records a dependency on the entire run-time value. It is the strongest dependency type, and hence subsumes the other five. The X type denotes the existence (or non-existence) of a name in a binding or closure context; the D type denotes the domain of a binding (i.e., the sequence of names in the binding's domain); the T type denotes the run-time type of a value; the L type denotes the length of a list or binding; and the E type denotes the value of a closure's expression (i.e., function body). Some of these dependency types, such as the T and L types, arise solely from the existence of certain language primitives.

Although some dependencies are subsumed by others, the evaluator does not prune such redundant dependencies when it forms cache entries for user-defined functions. It is unclear whether the cost of communicating and storing the extras outweighs the cost of detecting and removing them.

The dependency types that the evaluator uses were chosen based on practical considerations about the expected use of the Vesta language. The current dependency types are appropriate for the standard construction environment (Section 5.4) and many similar environments. It is possible, of course, that a radically different style of use of the Vesta language might reveal a need for additional dependency types.

7.4.4 Dependency Calculation Rules

We now give the mathematical rules for calculating dependencies. We first define $D(e, c, p)$ where p is a dependency path; $D(e, c, p)$ evaluates to a set of dependency paths. We then define $Dpnd(e, c) = D(e, c, V\!:\!\epsilon)$. Intuitively, $D(e, c, p)$ is the dependency for just the portion of e's value that is selected by p. The definition of $D(e, c, p)$ now proceeds by cases on the program structure:

- If $e = a$ ($a \in Literal$), then $D(e, c, t\!:\!p) = \emptyset$. Evaluating a constant has no dependency on the context.

- If $e = x$ ($x \in Id$), then $D(e, c, t\!:\!p) = \{t : x/p\}$. Evaluating a variable x depends only on the dependency path extended on the left by x.

- If $e = \lambda x.e_1$, only the following two cases arise:

$$D(e, c, V\!:\!\epsilon) = FVs(e)$$
$$D(e, c, E\!:\!\epsilon) = \{\}$$

  where $FVs(e)$ denotes the set of e's free variables. In both cases, the path must be empty. If the type of the path is V, the lambda expression depends on the whole closure value, that is, the set of e's free variables. If the type of the path is E, it depends on only the lambda expression of the closure value, namely, the expression e. Since e is incorporated into the primary key of every function call whose body contains e, it is correct not to record any dependencies in this case.

- If e = $\underline{\text{if}}$ e_1 $\underline{\text{then}}$ e_2 $\underline{\text{else}}$ e_3, then we have the following rule:

$$\frac{\begin{array}{c} d_1 = D(e_1, c, V\!:\!\epsilon) \\ v_1 = \mathit{Eval}(e_1, c) \\ d_2 = \text{ if } v_1 \text{ then } D(e_2, c, t:p) \text{ else } D(e_3, c, t:p) \end{array}}{D(e, c, t:p) = d_1 \cup d_2}$$

  This rule states that the dependency of a conditional is the union of the dependencies of the guard e_1 and the dependency of either e_2 or e_3, depending on the value of e_1. We use an empty path in computing the guard's dependencies because the guard evaluates to a boolean value that has no components.

- If $e = [n_1 = e_1, n_2 = e_2, ..., n_k = e_k]$, there are two cases to consider. If p is empty, it means that we depend on the entire binding. If $p = t\!:\!n_i/p_1$, it means that we depend only on the value of field n_i. The following rules cover the two cases:

$$D(e, c, t\!:\!\epsilon) = \cup_{i=1}^{k} D(e_i, c, V\!:\!\epsilon)$$
$$D(e, c, t\!:\!n_i/p_1) = D(e_i, c, t:p_1)$$

- If $e = e_1/n$, then $D(e, c, t:p) = D(e_1, c, t\!:\!n/p)$. For binding selection, we recursively call D with the path extended on the left by n.

- If $e = e_1!n$, then $D(e, c, t:p) = D(e_1, c, X\!:\!n/p)$. For the binding domain test, we recursively call D with the path extended on the left by n. Note that the new dependency path has type X, regardless of the type t.

- If $e = e_1 + e_2$, then there are two cases to consider: If p is empty, it means that we depend on the entire binding. If $p = t : n/p_1$, it means that we depend only on the binding that supplies the n field. In the case that the n field comes from e_1, we must add the dependency that n is not defined in e_2. The following rules cover the two cases:

$$D(e, c, t\!:\!\epsilon) = \tag{7.1}$$
$$D(e_1, c, V\!:\!\epsilon) \cup D(e_2, c, V\!:\!\epsilon) \tag{7.2}$$
$$D(e, c, t\!:\!n/p_1) = \tag{7.3}$$
$$\text{if } \mathit{Eval}(e_2!n, c) \tag{7.4}$$
$$\text{then } D(e_2, c, t\!:\!n/p_1) \tag{7.5}$$
$$\text{else } D(e_2, c, X\!:\!n) \cup D(e_1, c, t\!:\!n/p_1) \tag{7.6}$$

- If $e = \underline{\text{let}}\ x = e_1\ \underline{\text{in}}\ e_2$, then

$$\frac{\begin{array}{c} c_1 = c \circ \{x \to Eval(e_1, c)\} \\ d_2 = D(e_2, c_1, t:p) \\ d_{2a} = \{p' \mid p' \in d_2 \land head(p') \neq x\} \\ d_{2b} = \{t':p' \mid t':x/p' \in d_2\} \end{array}}{D(e, c, t:p) = d_{2a} \bigcup \cup_{p' \in d_{2b}} D(e_1, c,\ p')}$$

  where $\circ$ denotes the operation for extending a context, and $head(p)$ denotes the first element of p's path. We first augment the evaluation context with x mapped to $Eval(e_1, c)$ and compute d_2 as the dependency of e_2 in the augmented context. We then divide the dependency paths in d_2 into two sets d_{2a} and d_{2b}. The set d_{2a} contains paths unrelated to x. So, d_{2a} must be included in the result. The set d_{2b} contains paths starting with x. So, we need to recursively compute $D(e_1, c,\ p')$ for each path p' in d_{2b}.

- If $e = e_1(e_2)$, then

$$\frac{\begin{array}{c} Eval(e_1, c) =< \lambda x.e_3, c_3 > \\ d_1 = D(e_1, c, E:\epsilon) \\ c_1 = c_3 \circ \{x \to Eval(e_2, c)\} \\ d_3 = D(e_3, c_1, t:p) \\ d_{3a} = \{p' \mid p' \in d_3 \land head(p') \neq x\} \\ d_{3b} = \{t':p' \mid t':x/p' \in d_3\} \end{array}}{\begin{array}{l} D(e, c, t:p) = d_1 \bigcup \\ \quad (\cup_{p' \in d_{3a}} D(e_1, c,\ p')) \bigcup \\ \quad (\cup_{p' \in d_{3b}} D(e_2, c,\ p')) \end{array}}$$

  This rule is similar to the rule for the let construct, where e_1 in the let expression is like the argument e_2 here, and e_2 in the let expression is like the closure body e_3 here.

7.4.5 Example

We now present a simple example to demonstrate the above dependency rules. We compute the dependencies for the expression:

$$e = \underline{\text{let}}\ x = [r = [s = y], t = z]\ \underline{\text{in}}\ x/r/s$$

in the context c. Obviously, $Eval(e, c) = c(y)$. Here is the start of the dependency calculation:

$$\begin{array}{ll} & Dpnd(e,\,c) \\ \equiv & \{ \text{ definition of } Dpnd \text{ } \} \\ & D(e,\,c,\,V\!:\!\epsilon) \end{array}$$

Since the expression e is a let construct, the let rule applies. The main step in calculating the dependencies for the let construct involves calculating the dependency set named d_2 in that rule, where $e_1 = [r = [s = y], t = z]$ and $e_2 = x/r/s$. Here we derive the value for d_2, using c_1 to denote the augmented context $c \circ \{x \rightarrow Eval(e_1, c)\}$:

$$\begin{array}{ll} & D(x/r/s,\,c_1,\,V\!:\!\epsilon) \\ \equiv & \{ \text{ binding selection rule } \} \\ & D(x/r,\,c_1,\,V\!:\!s) \\ \equiv & \{ \text{ binding selection rule } \} \\ & D(x,\,c_1,\,V\!:\!r/s) \\ \equiv & \{ \text{ variable rule } \} \\ & \{\ V\!:\!x/r/s\ \} \end{array}$$

From the dependency set d_2, we compute the partitioned sets d_{2a} and d_{2b}:

$$\begin{array}{l} d_{2a} = \emptyset \\ d_{2b} = \{\ V\!:\!r/s\ \} \end{array}$$

We can now continue computing $Dpnd(e, c)$:

$$\begin{array}{ll} & D(e,\,c,\,V\!:\!\epsilon) \\ \equiv & \{ \text{ let rule } \} \\ & \emptyset \cup D([r=[s=y],\,t=z],\,c,\,V\!:\!r/s) \\ \equiv & \{ \text{ binding constructor rule } \} \\ & D([s=y],\,c,\,V\!:\!s) \\ \equiv & \{ \text{ binding constructor rule } \} \\ & D(y,\,c,\,V\!:\!\epsilon) \\ \equiv & \{ \text{ variable rule } \} \\ & \{\ V\!:\!y\ \} \end{array}$$

Hence, the evaluation of e in c depends only on the value of y, as we would expect.

7.4.6 Correctness

The following theorem states the correctness of the dependency calculation rules.

Theorem 1 (Caching Correctness) *If the expression e evaluates to a value v in the context c_1, then we can compute $Dpnd(e, c_1)$, and, if every path in $Dpnd(e, c_1)$*

evaluates to the same value in contexts c_1 and c_2, then e also evaluates to v in c_2. Formally,

$$
\begin{aligned}
(\exists v : \mathit{Eval}(e, c_1) = v) \Longrightarrow & \qquad (7.7)\\
(\exists d : \mathit{Dpnd}(e, c_1) = d) & \qquad (7.8)\\
\wedge\ (\mathit{Equiv}(c_1, c_2, \mathit{Dpnd}(e, c_1)) \Longrightarrow & \qquad (7.9)\\
\mathit{Eval}(e, c_2) = \mathit{Eval}(e, c_1))). & \qquad (7.10)
\end{aligned}
$$

Before implementing the dependency algorithm, we formalized the above evaluation and dependency rules for this subset of the Vesta language in the Nqthm theorem prover [9], and mechanically checked the correctness theorem. The proof is beyond the scope of this paper. It took several iterations of running the prover and correcting our rules before the mechanical proof succeeded. This proof effort revealed a couple of subtle errors in earlier versions of the rules. We then implemented the rules in the Vesta evaluator. Although we did not mechanically check the rules for the complete Vesta language, the subset we did verify covers the most complex aspects of the language, and so the mechanical verification was quite useful.

Our dependency calculation is related to earlier work by Abadi, Lampson, and Lévy, which uses a labeled λ-calculus to compute both dynamic and fine-grained dependencies [1]. However, their approach is quite different from ours. It associates labels with all expressions in a function body, and then develops rules for keeping the labels of only those expressions that are evaluated during a call. As a result, it records only one kind of dependency, analogous to our value dependencies. Also, their calculus supports only the selection operation on records. Computing dependencies for the binding operators ! and + complicates the problem significantly.

7.5 Caching Model Evaluations and Model Values

We treat two closely related topics in this section: caching evaluations of models, and recording dependencies on model values in other cache entries.

7.5.1 Model Evaluations

The evaluator creates two cache entries for each model evaluation: a *special* entry that takes advantage of the special semantics of models, and a *normal* entry that caches the model evaluation in the same manner as the evaluation of any other function call.[3]

The main difference between special and normal model entries is in the distribution of dependencies between the primary and secondary cache keys. The primary key of a special model entry combines two items:

1. the fingerprint of the immutable directory containing the model, which (as explained in Section 4.2.3) captures the state of the entire immutable tree rooted at that directory, and

2. the model's name relative to that directory.

Computing the primary key in this way implies that the evaluation depends on the model's entire directory tree, including the complete text of the model itself. All locally referenced files are thus captured in the primary key and need not be recorded in the secondary key. All non-local model imports are also captured, because the absolute pathname of each such import appears in the text of the model file and thus contributes to the fingerprint. (Unfortunately, any files in the model's directory tree that the model does *not* reference are also captured, as well as irrelevant elements in the model text such as imports that are not used, comments, and whitespace.) Because so much is captured by the primary key, the secondary key of a special model entry includes only fine-grained dependencies on the model's environment ("`.`") parameter.

In contrast, the dependencies of a normal model entry are calculated in the standard way, by treating the model as a closure of one argument whose context is defined by its `files` and `import` clauses. Thus a normal model entry's primary key captures only the parse tree of the model body, while its secondary key includes both references to the environment ("`.`") parameter and references to files and models made through the body's free variables.

Why two kinds of entries? Special model entries have fewer secondary dependencies than normal model entries, making them faster to look up in the cache. More importantly, as mentioned at the end of Section 7.1, special model entries serve as cut-off points to prevent too many dependencies from propagating up the function call graph: individual dependencies on files or imports of a model are not propagated beyond calls or references to that model. Without these cut-offs, the root node of the function call graph of a build would contain a dependency on every source file contributing to the build.

Moreover, cache hits on special model entries are common. For example, when an application is built against several frequently-used libraries, chances are that most of the libraries will be unchanged, so the evaluator will get fast cache hits on

[3] Recall from Section 5.3.4 that models are semantically equivalent to closures of one argument, so a model evaluation is semantically identical to a function call.

the libraries' special model entries. In fact, the performance benefits of the special cache entries created for model evaluations are so appealing that we sometimes create a separate model for a piece of Vesta code, rather than simply wrapping that code in a function definition within an existing model. This technique leads to a slightly larger number of model files in exchange for better caching performance.

False cache misses on special model entries are possible because the entries are intentionally coarse-grained. When looking up a model evaluation in the cache, the evaluator always checks for a special model entry first; if that lookup misses, it checks for a normal model entry. A new special model entry is created whenever the first lookup misses, even in the event of a cache hit on the normal model entry. Although it is uncommon for a cache lookup to miss on a special model entry but hit on the corresponding normal entry, this does happen occasionally. Such hits save enough work to amply justify the cost of creating the normal entry.

7.5.2 Model Values

Recall from Section 5.2.4 that every imported model has a corresponding closure value that can be returned as (part of) a function result just like any other value. Whenever any function evaluation depends on an unevaluated model value, we record this dependency in a coarse-grained manner, using the same idea described above for constructing the primary key of a model evaluation.

For example, suppose function f_1 calls model m_1, while function f_2 returns an unevaluated model m_2 as part of its result. (Both these cases are common in our own usage of Vesta.) When f_1 calls m_1, the evaluator will of course either find or create a special model entry for m_1 as just described, and f_1 will inherit the limited secondary dependencies from this coarse-grained cache entry as part of its own dependencies. In the case of f_2, although m_2 is unevaluated and hence no cache entry is involved, the evaluator still constructs f_2's dependency on m_2 using the same model fingerprinting technique. That is, the evaluator creates a "V" dependency whose fingerprint field combines (1) the fingerprint of the immutable directory from which m_2 was imported, and (2) m_2's name relative to that directory. As before, this technique is correct because m_2's value as an (unevaluated) closure is fully specified by the immutable text of the model file that defines it and the immutable contents of the directory tree where the files and import clauses in that model are resolved.

7.6 Example Evaluation Call Graphs

We now consider the effects of the evaluator's function caching in practice. We show the call graphs that result from the evaluation of various packages.

7.6.1 Scratch Build of the Standard Environment

Figure 7.1 shows the call graph that results from evaluating the model for the standard construction environment against an empty cache. In this figure and the next two, the nodes denote function invocations, and an edge from a node f to a node g indicates that f calls g. The edge is solid black if f and g are defined in the same package, and gray otherwise.

The call graph in this example can be divided roughly in half. The left half of the figure shows the evaluation of a stripped down "backstop" standard environment. This environment contains bridges and libraries necessary to build the full standard environment, which is shown in the right half of the figure.

In both cases, building the standard environment entails evaluating bridge models and library models. The library models are divided into three groups: C libraries, Vesta libraries, and Modula-3 libraries. The full standard environment contains three more bridges than the backstop environment. One of these, the *lim* bridge, requires more work, since the lim tool exported by the bridge must be built from source.

Note that although this evaluation was performed against a cache that was initially empty, some cache hits still occurred. In particular, cache hits occurred on five of the six bridges in the full environment because identical versions of those bridges had already been evaluated in the backstop environment. Cache hits also occurred on the complete sub-trees of the C and Modula-3 libraries, again because those libraries had been evaluated in the backstop. In contrast, hits did not occur on all of the Vesta libraries because the version of those libraries referenced by the full standard environment model (i.e., version 30) differs from the one referenced by the backstop (version 28). Note that none of the library evaluations resulted in a tool invocation, since in general the actual *construction* of a library is delayed until a program is built against it.

7.6.2 Scratch Build of the Vesta Umbrella Library

Figure 7.2 shows the call graph that results from building a "hello world" C program against the complete Vesta umbrella library, assuming that the construction of the standard environment model shown in Figure 7.1 is already cached. The only reason for building such a simple program against the Vesta umbrella is to

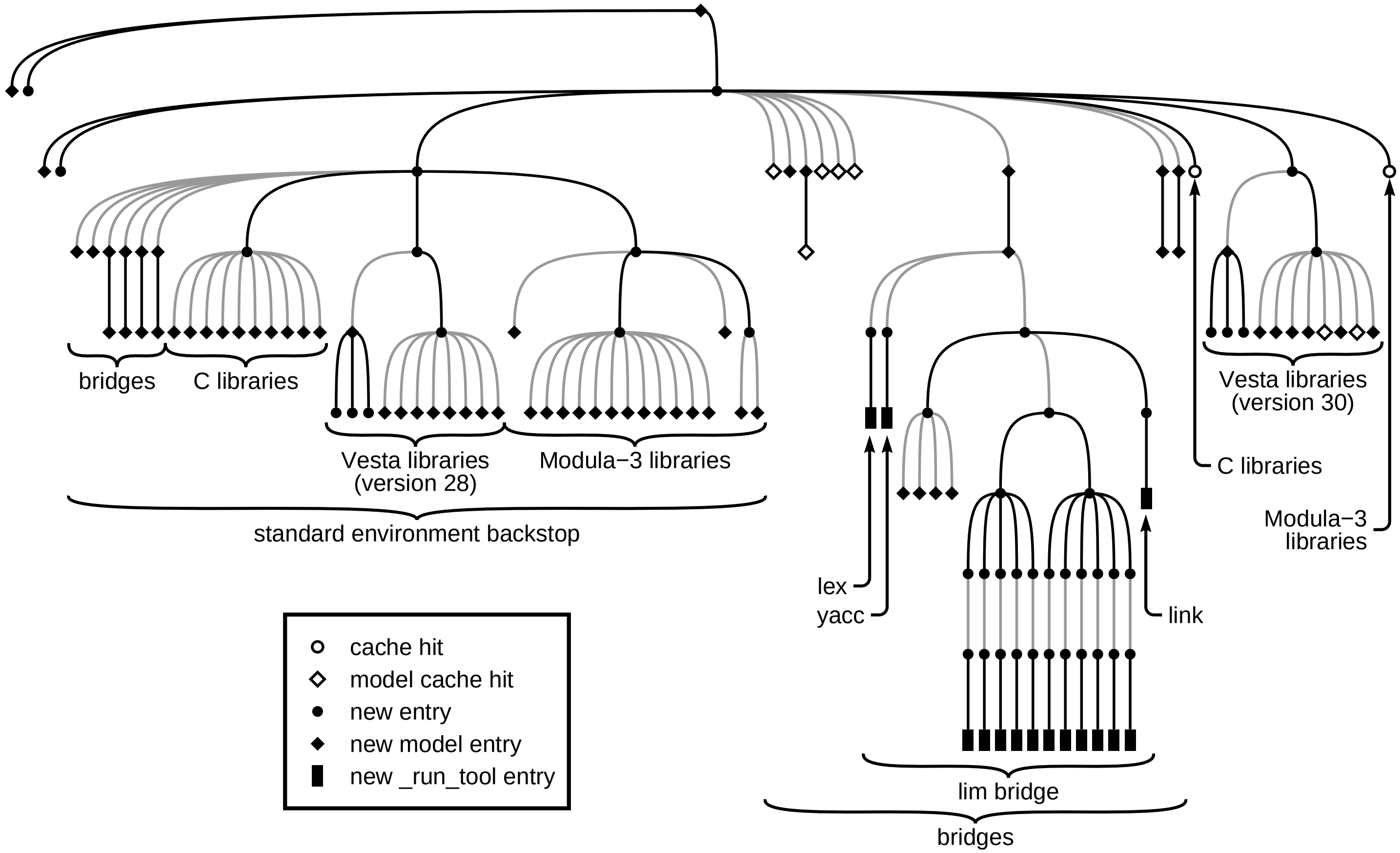


Figure 7.1: The call graph for the construction of the standard environment model. To build the standard environment, a stripped down ‘‘backstop’’ environment is first constructed as shown in the left half of the figure. The environment proper is then built, which includes the construction of three additional bridges. One of these, the lim bridge, requires building the lim tool from source.

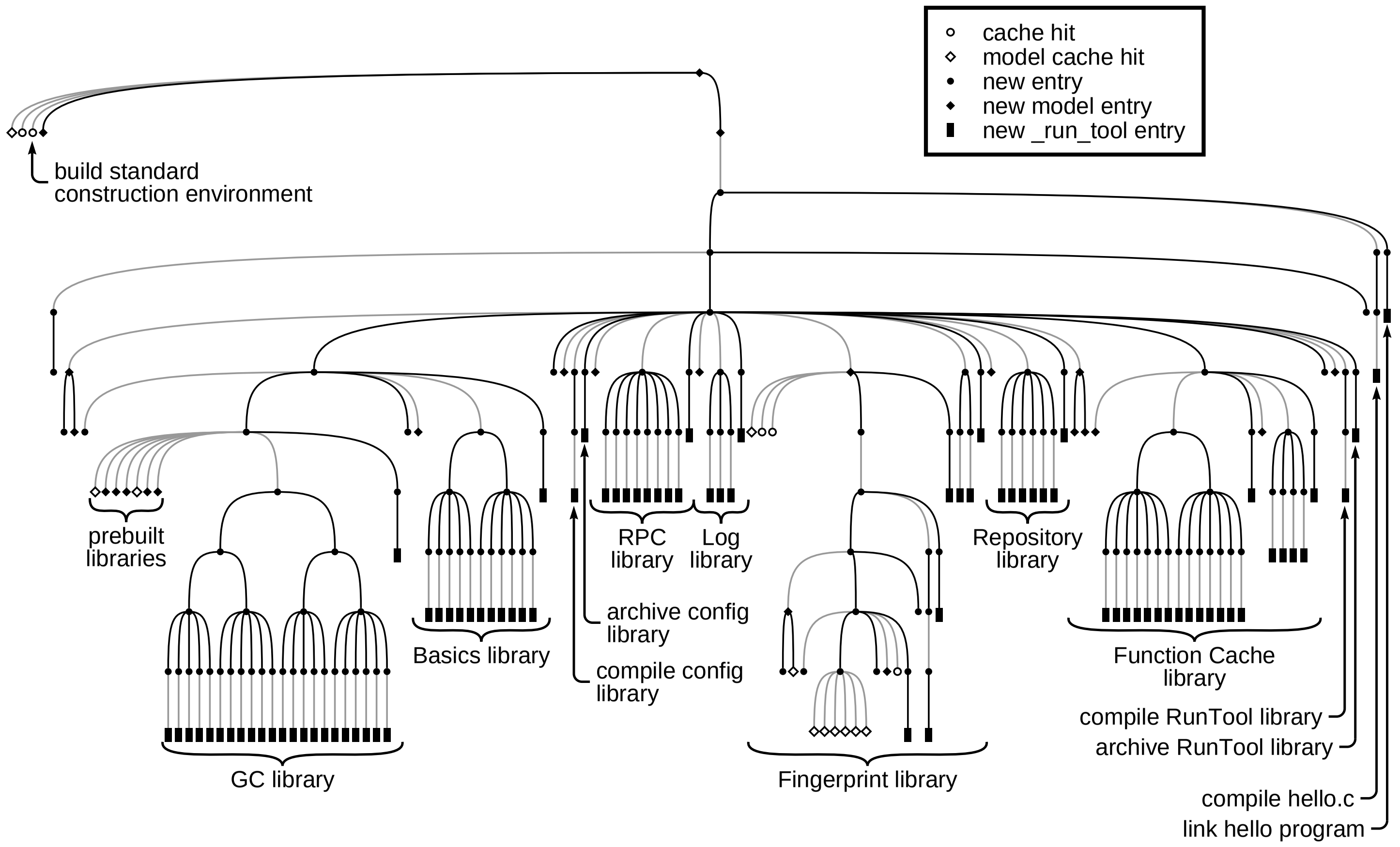


Figure 7.2: The call graph for the construction of ‘‘hello world’’ linked against the complete Vesta umbrella library, assuming the evaluation of the standard construction environment is cached. The bulk of this figure depicts the construction of the Vesta umbrella library, which consists of seven prebuilt libraries and nine other libraries built from source.

illustrate the construction of the umbrella, which constitutes the bulk of this figure. The actual construction of the hello world program itself consists only of the two rightmost tool invocations.

Although the function cache is relatively empty in this example, the importance of good caching is evident. The cache hit on the standard construction environment in the upper left of the figure is critically important, as it saves the evaluator from having to evaluate the entire call graph of Figure 7.1!

Perhaps the main thing to notice about this call graph is its sheer size. Building the Vesta umbrella library from scratch requires 86 tool invocations. Except for the fingerprint library, each sub-library is constructed by compiling the library's sources one at a time, and then collecting the resulting objects together into a library archive.

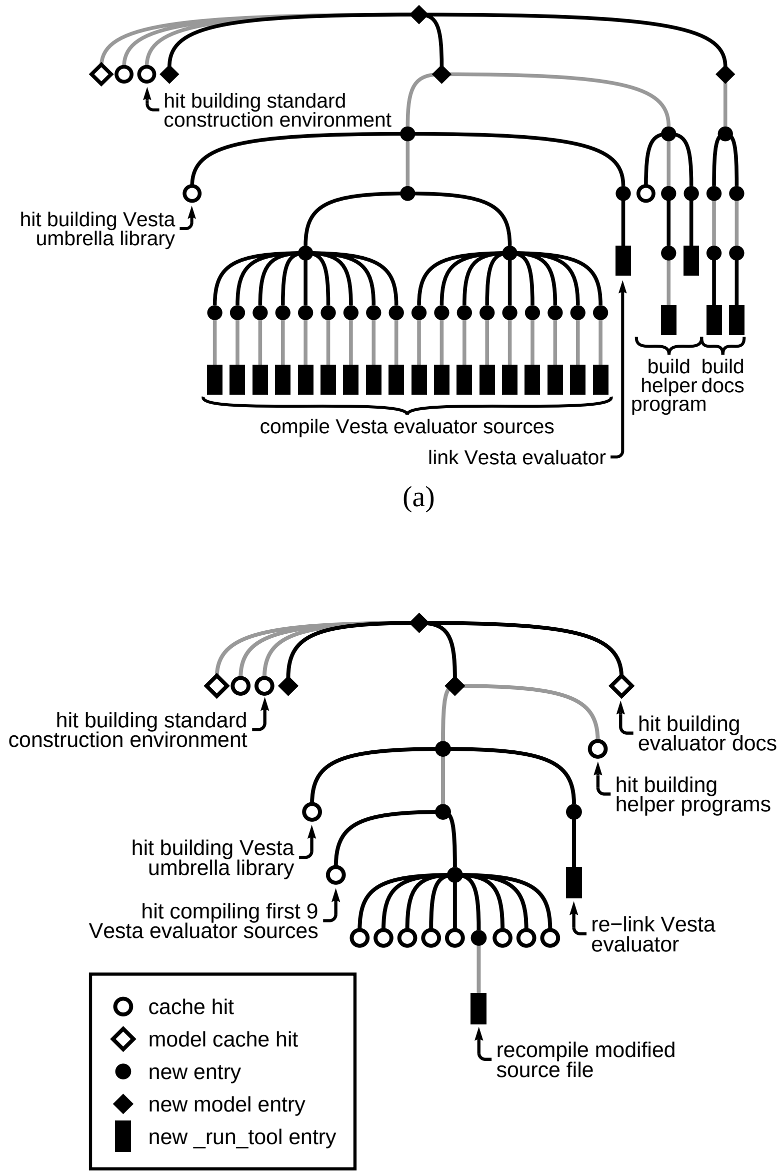
The construction of the fingerprint library requires a bit more work, since a program for generating a header file of computed fingerprint constants must first be built and run. This example shows how the intermediate results of a build (namely, the program for generating the header file) can be invoked later in the build. It also illustrates that Vesta's distinction between source and derived files is not the same as the C compiler's distinction between source and object files; here, a C source file is mechanically generated, not stored directly in the repository, so it is a derived from Vesta's point of view.

7.6.3 Scratch and Incremental Builds of the Evaluator

Figure 7.3(a) shows the call graph that results from building the Vesta evaluator package from scratch, assuming that the standard environment and Vesta umbrella library are cached. In addition to the Vesta evaluator executable, building the package also causes a helper program and some derived documentation files to be constructed.

This figure is an even better example of the importance of effective caching. In this case, there is a hit not only on the evaluation of the standard construction environment, but also on the entire Vesta umbrella library shown in Figure 7.2. Hence, the 86 tool invocations shown in that previous figure are all avoided by virtue of a single cache hit. The construction of the evaluator itself is straightforward: its 18 sources are compiled into objects, and those objects are then linked together against the Vesta umbrella library. The construction of the helper program and documentation files is also straightforward.

Figure 7.3(b) shows the call graph that results from an incremental build of the Vesta evaluator. This example illustrates the typical case that occurs during the inner edit-build-test loop of the development cycle. This figure was produced by starting with the cache that resulted from the evaluation of Figure 7.3(a), modify-


hit building standard construction environment
hit building Vesta umbrella library
compile Vesta evaluator sources
link Vesta evaluator
build helper program
build docs
hit building standard construction environment
hit building evaluator docs
hit building helper programs
hit building Vesta umbrella library
hit compiling first 9 Vesta evaluator sources
re−link Vesta evaluator
recompile modified source file
○ cache hit
◇ model cache hit
● new entry
◆ new model entry
▮ new _run_tool entry

(a)

(b)

Figure 7.3: The call graphs that result from building the Vesta evaluator package from scratch (a) and again after one of the evaluator's source files has been modified (b). In both cases, hits occur on the standard construction environment and the Vesta umbrella libraries.

ing one of the evaluator's source files, and rebuilding. In this case, there are many cache hits and only two tool invocations (one to recompile the modified source and one to relink the evaluator executable). Because the helper program and documentation sources were not changed, high-level cache hits occur on those parts of the evaluation.

This example also illustrates an interesting feature of the C and C++ bridges. In particular, notice that the compilation of 10 or more sources in a library or application produces a balanced subtree in the call graph so that no more than 9 files are compiled together directly under a single node. This "weeping willow" effect is especially evident in the construction of the GC library in Figure 7.2. When the evaluator was built from scratch, the compilation of its 18 sources was automatically divided by the bridge into two groups of 9. As a result, the work required to recompile the files in the first group can be avoided by a single cache hit. In the general incremental case, this divide-and-conquer technique results in a number of cache lookups proportional to the logarithm of the number of sources in the library or application, rather than the linear number that would result if the naive flat organization had been used.

7.7 Error Handling

Some care is required to handle evaluation errors correctly in the face of caching. Two kinds of errors are possible: runtime errors that occur while evaluating an expression in the Vesta language and failed `_run_tool` calls. The evaluator handles the two kinds of errors somewhat differently.

When an evaluation error occurs, such as attempting to select an undefined component of a binding, the evaluation is aborted. An error message is printed to the user, along with the evaluation call stack to indicate the error's source. In this case, naturally, no cache entries are created for the function evaluations that were still in progress (i.e., on the call stack) at the time of the error, since they had not yet returned a value that could be cached. Earlier function calls in the evaluation that did complete are cached, however, so if the evaluation is performed again, the same error message is regenerated quickly.

In contrast, when a `_run_tool` call fails, it may be undesirable to abort the entire evaluation immediately. For example, when the evaluator is compiling a long list of sources, the user will often go off and do something else. If a compilation early in the list fails, the user will be much happier if the evaluator goes on to compile the rest of the files before he returns than if it aborts immediately. Therefore, when a `_run_tool` call fails, if an appropriate command line switch has been given, the evaluator prints an error message and continues, indicating the error as part of the function result. (If this switch is not given, the evaluator prints an error message and aborts the evaluation.) Bridges are typically written to detect any such errors and pass the special value ERR up to their callers, causing any attempt to use the result of the compilation to fail and stop the evaluation.

When a _run_tool call fails, although it would be theoretically correct to cache the failure and return it again if the build is repeated, it is preferable not to do so, for two reasons. First, occasionally a tool invocation may fail because of some transient condition that is outside Vesta's control, such a full disk or a network timeout. Capturing such a failure in the cache would make it impossible to correct the problem by clearing the transient condition and retrying. Second, Vesta bridges are usually written to print such messages as a side-effect, rather than including them as part of the _run_tool result. When a Vesta user needs to reproduce the error messages from a failed compile, it is important for him to be able to run the compiler again to regenerate the messages.

Therefore, failed calls to _run_tool must not be cached. Caching any function higher up in the call graph would be equally bad, since a hit on such an entry would prevent the failed call to _run_tool from reoccurring, again suppressing the error messages. To handle this situation, the evaluator records a *cachable* flag for each runtime value. The evaluator computes the cachable flag of a new value based on the cachable flags of the values from which the new value is produced. A cache entry for a function evaluation is created only when its result value is cachable.

Good error reporting has its costs. When the evaluator encounters a runtime error in a Vesta model, it is important for it to report the location (file name, line number, and column number) of each entry on the function call stack; if this information were not provided, failing models would be very difficult to debug. Therefore, when the evaluator caches a function result value that includes a closure, it must store not only the parse tree of the closure's function body, but also the original model name from which the body was read and the original line and column numbers of its tokens. These extra fields increase the function cache's communication, memory, and storage costs. However, our experience debugging system models that use closures demonstrates that the extra caching costs are well worthwhile.

Chapter 8

Weeder

Vesta manages the creation and naming of derived files automatically. The deletion of derived files is left to a separate quasi-automatic management process called the Vesta *weeder*. The weeder is also responsible for deleting unwanted cache entries, which consume system resources as well.

8.1 Motivation

Ideally, the file deletion process would be completely automatic, perhaps operating according to some heuristic. For example, it might delete files older than some threshold age, or those that had not been recently used. However, there are two problems with such heuristic approaches.

First, a derived file cannot simply be deleted irrespective of those cache entries that refer to it. A serious failure would result if a client could get a hit on a cache entry that referenced a nonexistent derived file! Hence, the deletion of derived files must be coordinated with the deletion of unwanted cache entries.

Second, the frequency of derived file obsolescence tends to vary dramatically from package to package, depending on how actively a package is undergoing development. As a result, any fixed heuristic is likely to delete too much in some packages, and not enough in others. Since the developers know the most about which package versions are worth preserving, it makes sense to give the developers control over what is kept, rather than relying on a heuristic.

These points suggest that the deletion of derived files and cache entries should be managed by a common process, and that the input to that process should be specified by people. The Vesta weeder is responsible for managing the deletion process. It determines which derived files and cache entries to keep using a technique akin to garbage collection, and then communicates with the repository and function cache, which do the actual work of deleting everything else.

Apart from the obvious requirement that the weeder not delete anything that it was meant to keep, the main consideration in the design of the weeder is that it be as unobtrusive as possible. By this we mean two things. First, weeding should be transparent to users. While a weed is in progress, it should be possible to do normal builds, and the performance impact of weeding on the rest of the system should be minimal. Second, the administrative overhead of running a weed should be negligible. In particular, weeding should be an infrequent activity, the weeder should not require gross amounts of disk space or memory, and the instructions telling the weeder what to keep should be simple to specify.

In this chapter, we limit our discussion to the weeder's design, implementation, and usability. We evaluate the weeder's performance in Section 9.5 of the next chapter, covering the measured performance of weeding and the frequency with which weeding is required for a medium-sized development project.

To simplify the task of writing weeding instructions, the set of cache entries and derived files to keep is specified at a coarse granularity; namely, at the level of package builds. If the weeder is instructed to keep a particular package build, it automatically keeps all cache entries and deriveds generated by that build. The package builds to keep are specified using a simple but powerful pattern language (described below), so instruction files can be kept short and easy to understand. As we shall see, one of the advantages to specifying package builds is that they are quite straightforward to name.

What happens if the weeder is run with instructions that tell it to delete too little or too much? The cost of deleting too few derived files is that not enough disk space is freed up. In that case, the weeder can simply be re-run with a smaller list of packages to keep. The cost of deleting too much is that some builds may have to be repeated. But because all sources in Vesta are immutable and immortal, and all builds are repeatable, there is never a correctness problem associated with deleting too much. Hence, there is a time-space tradeoff inherent in the weeding process. Weeding more derived files frees up more disk space, but it may require time to recreate those weeded files (and cache entries) if they are needed later.

Why is Vesta's clean-up process called weeding instead of garbage collecting? In a normal garbage collector, there is a clear distinction between objects that are reachable from a set of known roots and those that are unreachable garbage. In Vesta, there is no real “garbage”: any cache entry is potentially useful in a future build. Hence, a person must decide which builds are worth keeping. These builds are taken as the “roots” of a typical mark and sweep garbage collection; all other builds are weeded from the system. Deciding which builds to keep is subjective: one person's flower is another's weed.

8.2 Weeding Instructions

The input to the weeder names the exact versions of control-panel models on which the evaluator was invoked and whose corresponding builds are deemed worth keeping. It would be too tedious to have to specify the exact name and version of every Vesta package build to be kept, so the weeder's input is a pattern language. Here is the input for one of our recent weeds:

```
+ /vesta/west.vestasys.org/vesta/release/[LAST-1,LAST]/.main.ves
+ /vesta/west.vestasys.org/vesta/*/LAST/.main.ves
+ /vesta/west.vestasys.org/vesta/*/checkout/LAST/*/.main.ves
```

The first line causes the two latest versions of the Vesta release to be kept. The second line keeps the last checked-in version of every Vesta package build, and the last line keeps all checkout session builds of the last version of every Vesta package. Each pattern ends in "`.main.ves`", indicating that each pattern names a root control-panel model on which the Vesta evaluator was actually invoked. The weeder keeps all derived files and cache entries created by such root invocations. In addition, it is possible to specify as a command-line option to the weeder that any builds performed recently (i.e, in the last *n* hours for some given *n*) should also be kept.

Note that each line is preceded by a plus sign, indicating that these versions are to be kept. The weeder's input language also allows lines beginning with a minus sign, which removes the specified package builds from the set of builds to keep. The lines are processed in order; plus and minus signs may be alternated to successively include and exclude different sets of package builds. The exclusion feature makes the language slightly more complex, and in hindsight, perhaps it was not necessary; thus far we have not had occasion to use it.

8.3 Implementation

Before describing the weeder's implementation, we first describe some preliminary information about the function cache and evaluator.

First, we describe the cache from the weeder's point of view. Running a set of Vesta evaluations against the cache can be viewed as generating a directed acyclic graph (DAG) in which the nodes are cache entries (i.e., function calls whose results are cached) and in which there is an edge from entry f to entry g if and only if the call g was made during the evaluation of f's body. This DAG is in general not a tree, because a cache entry acquires an additional parent whenever a new cache hit is produced on it. The roots of the DAG correspond to the evaluations of the top-level (control panel) models on which the Vesta evaluator has been invoked. The function cache in fact maintains this graph as an explicit data structure; the nodes are cache entries that are stored as described in Chapter 6, while the edges are kept on disk as a separate file called the *graph log*. Each cache entry is identified uniquely by a small integer called its *cache index*, and the graph log edges are represented in terms of cache indices. Each entry in the graph log contains the index of a cache entry and the indices of all its children. The graph log is a "log" in the sense that except during weeding, new edges are only appended to it; existing edges are never deleted or changed.

Second, a point about derived files. To avoid large runtime values, the evaluator represents the contents of a source or derived file by the shortid of the corresponding file in the repository. For example, the result value of a compiler invocation might be a singleton binding that maps the name of a derived file to the file's shortid. The shortids of all files referenced in a function's result value are also stored in the graph log entry corresponding to that function evaluation.[1] Hence, all deriveds referenced by an evaluation can be reached by traversing the function call graph corresponding to that evaluation.

So much for preliminaries. The weeder itself works like a mark-and-sweep garbage collector. It reads the user's specification of which model evaluations to keep, and treats those as the roots of its mark phase. It then reads the graph log written by the function cache, and marks all cache entries reachable from any of the roots. It also writes to a separate file the shortids of all derived files referenced from any marked entry. Once it has marked the derived files and cache entries to keep, the weeder notifies the repository and the function cache to do the actual work of deleting the unmarked derived files and cache entries (in parallel).

This description glosses over two important issues in the weeder's implementation.

The first issue arises from the fact that Vesta's design targets are large enough that the entire function call graph is not expected to fit entirely in memory (see Section 3.3). However, the graph is small enough that the weeder can afford to keep a single *mark* bit in memory for each cache entry; it collects these bits together into a large bit vector. The weeder's mark phase begins by marking each of the roots. It then works by iteratively scanning the graph log. On each scan, any graph log entry whose corresponding cache index is unmarked is written out to a new version of the log to be read on the next iteration. If, on the other hand, the graph log entry's corresponding cache index is marked, the mark bits corresponding to its children are set, and the graph log entry is then dropped. This process repeats until no more

[1] For the most part, these are derived files, but it is quite possible for a function to return a source as part of its result. Strictly speaking, such files are not derived, but they must still be protected from weeding by virtue of being referenced in a function result. In this chapter, we will simply refer to all files referenced by a function result as derived files.

entries get marked.

In general, this algorithm requires *d* iterations, where *d* is the depth of the call graph. Although *d* is not expected to be large, the weeder implements an optimization to reduce the number of iterations further: instead of writing out all unmarked graph log entries immediately, it holds as many as possible in a fixed-size memory buffer, from which they are retrieved and processed if they become marked. Unmarked entries are then written out only when the buffer overflows. How big to make the memory buffer is strictly a time-space trade-off: the larger the buffer, the fewer iterations are required. We report on our experience for a particular buffer size in Section 9.5.

The second, more subtle issue involves the synchronization between the repository, function cache, and weeder. Since weeding may require a noticeable amount of time, the interfaces between the weeder and the other two processes have been designed so that weeding can proceed concurrently with normal builds. There are several aspects to this problem, but here we will focus on the most important one, namely, the protection of recently created derived files and cache entries from the weeder. This description omits several important details, but it describes the essence of the design.

The weeder's first action is to checkpoint the graph log; its remaining processing uses the resulting file. Hence, the weeder operates on a snapshot of the function call graph. Care is required because some evaluations may have been in progress at the time the snapshot was taken.

According to the weeder's marking algorithm, a derived file or cache entry is protected from weeding by virtue of a path in the graph log from one of the user-specified weeder roots to the derived file or cache entry. But if an evaluation was in progress when the snapshot was taken, any new entries created by that evaluation are not yet reachable from a root, since the root entry is not created until the evaluation completes. Hence, if no special action is taken, these newly created entries would all be weeded.

To prevent this problem, all newly created cache entries are automatically leased [20, 36]. We use the term *lease* in a broad sense: a lease is an agreement between two or more parties that remains valid up to a prearranged time even if the parties are not in active communication. In this case, the evaluator acquires a lease on newly created cache entries that is also recognized by the cache server and the weeder. Leases can be renewed; the lease on a cache entry is automatically renewed whenever a client gets a cache hit on that entry.

A lease protects a cache entry from weeding. A lease on a cache entry also protects all cache entries and derived files reachable from that cache entry. At weeding time, the cache informs the weeder which entries are currently leased, and the weeder additionally uses all such entries as roots of its mark phase.

Once the weeder has communicated to the function cache which cache entries to delete, any lookup on an entry pending for deletion is treated as a miss, even if the entry still exists in the cache. Hence, there is a small possibility that the evaluator may have to re-evaluate a function whose value is still in the cache but was about to be deleted. There would be no point in adding machinery to keep this rare and harmless event from occurring.

All leases eventually expire. If leases did not expire, then no cache entries or derived files could ever be deleted! We have used lease durations as long as three days during Vesta development, to allow for situations where the evaluator is stopped in the debugger for a day or two, and as short as three hours in a production environment where disk space was in short supply. For efficiency, our lease implementation may actually protect entries for up to twice as long as the nominal lease duration. When choosing a lease duration, it is best to err on the conservative (long) side; an unduly long lease causes no problems except for preventing some files and cache entries from being deleted as soon as they could have been.

Chapter 9

Performance

This section describes Vesta's performance. We first compare the overall system performance to that of Make, focusing on the resource requirements of the Vesta components that typically run on user machines, namely the evaluator and runtool server. We then present more detailed performance measurements of the Vesta components that typically run on a central server machine, namely the repository, function cache, and weeder. We also describe the performance of our remote procedure call (RPC) library, which is used for communication among the components.

One of Vesta's main design goals is that it should scale to the construction of very large software (see Section 3.3). Unfortunately, at the time the measurements in this chapter were taken, the loads we had placed on Vesta were quite small compared to those for which it was designed. For example, we had never stored more than about 10,000 entries in the function cache at any one time, two orders of magnitude less than the design target. In addition, the system had had at most four simultaneous users, one to two orders of magnitude less than the design target.[1] Hence, we do not have any performance measurements directly attesting to Vesta's scalability.

However, all of the Vesta components have been designed and implemented with an eye toward good scalability. Hence, we have reason to hope that as the system is put under increased load, any unforeseen performance bottlenecks that arise will not be show-stoppers.[2] In this section, we argue for Vesta's scalability by extrapolating from our current performance measurements and by describing the measures we have taken to avoid scalability bottlenecks.

[1] Later, a group of about 130 developers within Compaq adopted Vesta and used it to develop a code base of about 700,000 lines over a 2.5 years period. Unfortunately, we do not have performance measurements for their installation.

[2] This has proven true so far with our product development users.

9.1 Hardware Configuration

The measurements described in this chapter were performed with the standard Vesta system configuration. In this configuration, the evaluator and runtool server run on a single client machine, and the repository and function cache run on a single server machine. For the Make experiments described below, we ran Make on the client machine, using files served by a standard in-kernel NFS version 3 implementation running on the server machine. In both sets of experiments, the underlying files being served by the repository and NFS3 servers were mounted on a local Digital AdvFS file system using two directly attached Digital RZ28 SCSI disks.

The client machine used in our tests was a Digital AlphaStation 500 5/333, with a 333 MHz CPU and 192MB of memory. The server machine was a Digital AlphaStation 400 4/233, with a 233 MHz CPU and 192MB of memory. The two machines were connected by SRC's local AN2 network, a prototype of Digital's GIGAswitch/ATM high-bandwidth ATM network. More details about Vesta's communication costs are described in Section 9.6 below.

These machines are now two generations old, so we expect the absolute performance in both the Vesta and Make cases would be substantially better on modern hardware.

9.2 Overall System Performance

In this section, we report our measurements of the overall performance of the Vesta system from a user's point of view.

First, we compare the performance of Vesta against that of Make [17]. Our measurements show that Vesta outperforms Make on both scratch and incremental builds. In the most common case, an incremental build in which a small number of files are recompiled, Vesta substantially outperforms Make. This improved performance comes despite the fact that Vesta provides considerably stronger consistency guarantees than Make does.

Second, we present more detailed measurements of Vesta's client performance. Our measurements show that the Vesta caching machinery performs quite well. The cost of incremental builds is indeed proportional to the magnitude of the change, not to the total size of the software being built. Our measurements also confirm that Vesta spends most of its time in runtool calls, indicating that the overhead induced by the evaluator and function cache is low.

Finally, we characterize the memory usage of the evaluator and the runtool server—the two Vesta processes that are expected to run on client machines. The measurements show that the run-time memory consumption of these processes is reasonable: Vesta does not compromise the performance of the other applications on the client machine because of its memory consumption.

To compare Make and Vesta, we ran three tests. The *Hello* test requires compiling and linking a toy "hello world" program. The *Evaluator* test requires building the Vesta evaluator, which in turn requires building all of the Vesta libraries used by the evaluator. Finally, the *Release* test entails building the entire Vesta release, which includes building all of the Vesta libraries, server programs, utilities, and test programs.

Table 9.1 summarizes various attributes of the three tests. When counting the total source lines, we treated all the C, C++, and header files in the Vesta implementation as source files. When counting the number of modules, we included only the Ċ files, not the header files. The number of runtool calls reported in the table is the number of times an external tool invocation is required during the test; it includes invocations of compilers, linkers, and archivers. Finally, the number of packages reported is the number of separate packages around which the sources are organized.

	Total	Number of	Number of	Number of
Test	Source Lines	Modules	Runtools	Packages
Hello	10	1	2	1
Evaluator	53,304	103	117	11
Release	119,602	255	333	16

Table 9.1: Sizes of the Three Build Tests

Even the largest test's size is well below that of the software systems for which Vesta was designed. In interpreting the results of our measurements, we make arguments about Vesta's scalability when it is possible to extrapolate from the data presented.

9.2.1 Performance Comparison with Make

In this subsection, we compare the performance of Vesta and Make for both scratch and incremental builds. As explained in Chapter 2, Make does not always build consistent software systems as Vesta does. The combination of Make and Makedepend, though a better approximation to Vesta, still falls short of providing the same consistency guarantee. Figures 9.1 and 9.2 compare the performances of Vesta and Make on scratch and incremental builds, respectively. Using Makedepend with Make would add a nontrivial amount of time to the Make times in the tables, which

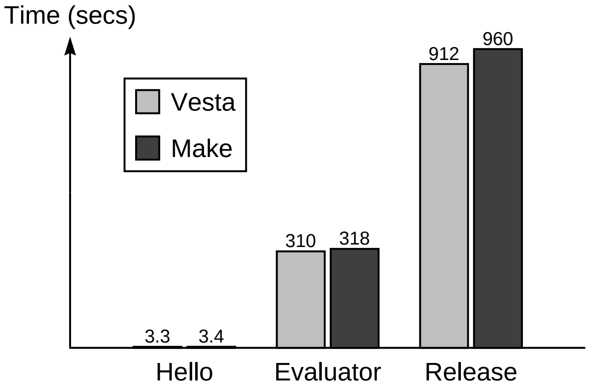


Figure 9.1: Elapsed time in seconds for scratch builds of the three tests using Vesta and Make.

would be quite significant in the case of incremental builds.

As shown in Figure 9.1, Vesta was slightly faster than Make on scratch builds of all three build tests. Even though Vesta outperforms Make on scratch builds, it is worth noting that scratch builds occur less often under Vesta. Make users often initiate scratch builds because building a system from scratch is the only way to guarantee consistency with Make. The need for such scratch builds is eliminated under Vesta, because Vesta always produces consistent incremental builds. However, if a user changes a low-level header file that is used in many places, a near-scratch build will result in which many files are recompiled. Figure 9.1 indicates that Vesta performs quite well on such builds.

As shown in Figure 9.2, Vesta clearly outperforms Make on incremental builds. To test incremental build performance, we touched a single source file and rebuilt. In each test, this required one invocation of the compiler and one invocation of the linker, so we estimate that the total time spent running tools was roughly the same under Vesta and Make. Figure 9.2 shows the elapsed incremental build times for the three tests.

When a program is composed of sources spanning multiple packages (as occurs in the Evaluator and Release tests), developers using Make typically run Make only in the package on which they are working; these are the times reported under the “Make One” column. However, such incremental builds can lead to inconsistent results if files in any of the other packages have changed. To get a (more) consistent build, Make can be run in all of the contributing packages; these are the times reported under the “Make All” column. In this incremental test, however, we did not modify any of the other packages, so the difference between the “Make One” and “Make All” columns is simply the time required by Make to determine that the

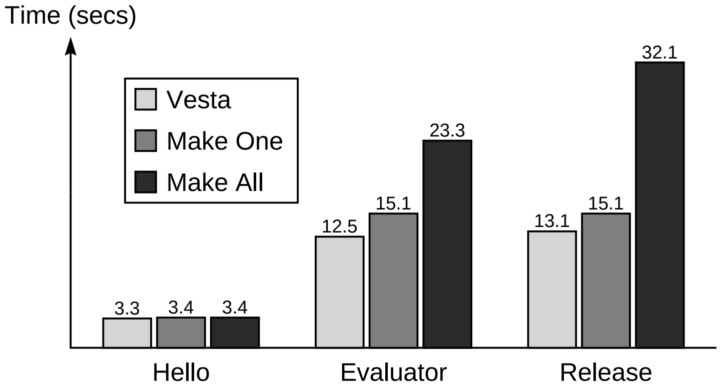


Figure 9.2: Elapsed times in seconds for one-module, incremental builds under Vesta and Make. The "Make One" column is the time required to run Make in the single package in which the change was made, while the "Make All" column also includes the time required to run Make in each of the program's other packages.

other packages were all up-to-date.

We consider the "Make All" scenario to be much more of an apples-to-apples comparison with Vesta than the "Make One" scenario. However, neither Make scenario comes close to matching Vesta's consistency guarantees, particularly because they do *not* include the time required to run Makedepend in each of the relevant packages.

Except for the trivial Hello program, Vesta ran 86% to 145% faster than Make alone. We expect that Vesta's performance advantage will tend only to increase when building larger software, since incremental builds under Make and Makedepend require time proportional to the size of the software being built, while incremental builds under Vesta require time proportional to the magnitude of the change. This claim is confirmed by the fact that Vesta ran 86% faster than Make alone on the Evaluator test, but 145% faster on the larger Release test.

9.2.2 Performance Breakdown

In this subsection, we examine how time was spent by Vesta in both scratch and incremental builds.

We divide the total elapsed time into three parts: the time spent in the Vesta evaluator itself, the time spent by the evaluator making remote procedure calls on the Vesta function cache, and the time spent in external runtool invocations. Note that the overheads due to the repository and the tool encapsulation machinery are

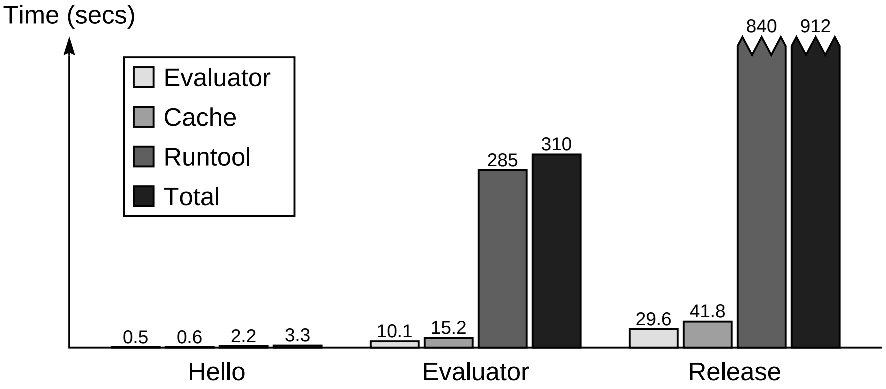


Figure 9.3: Breakdown of scratch build elapsed times in seconds for each of Vesta's components.

included in the runtool time. We have not analyzed these overheads in any greater detail.

For scratch builds, Figure 9.3 shows that the time spent doing evaluation and caching for non-trivial builds is no more than 8% of the total running time. Moreover, the time spent running the evaluator and cache accounts for a smaller fraction of the total elapsed time the larger the software being built. As yet larger software is built with Vesta, we expect this fraction to continue to decline somewhat, and certainly never to increase.

The breakdown of the elapsed time for incremental builds is shown in Figure 9.4. It demonstrates that Vesta's performance scales well, at least for medium-sized programs. Even though the Release sources are more than twice the size of the Evaluator sources, the absolute times spent in the Vesta evaluator and function cache are small and nearly identical across the two builds. This result supports our claim that the time spent in the Vesta system proper is proportional to the magnitude of the change, not to the total size of the software being built.

9.2.3 Caching Analysis

In Chapter 7, we argued that caching user-defined function calls is necessary to achieve good overall system performance. Caching user-defined function calls has its costs (most notably, the cost of computing each function's fine-grain dependencies). In this section we show that the benefits outweigh the costs in practice.

To measure the effectiveness of caching user-defined functions, we ran our tests with three different levels of caching. At the lowest level, we cached only runtool calls. This level of caching is essentially what Make provides, but with Vesta's

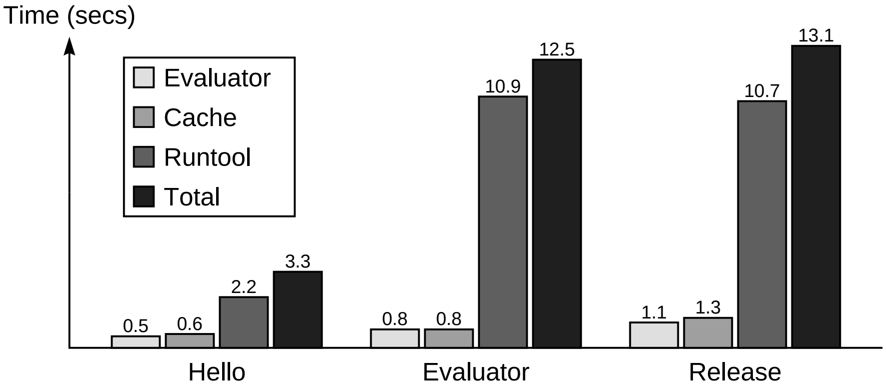


Figure 9.4: Breakdown of one-module, incremental build elapsed times in seconds for each of Vesta's components.

strong consistency guarantee. At the next level, we cached both runtool calls and model calls.[3] Models are a natural caching boundary because every package build is performed by evaluating the package's root model. Finally, we report on Vesta's default behavior, which is to cache runtool calls, model calls, and calls of user-defined functions. Figure 9.5 shows the results of doing incremental builds of the three tests with these three caching levels; the final column shows the "Make All" elapsed time from Figure 9.2 for comparison.

We can draw three main conclusions from this data:

- Caching all function calls is substantially faster than caching only runtool calls or runtool and model calls together. We expect that this performance difference will become even bigger for larger software.

- Caching only tool invocations is clearly *not* sufficient. When only tool invocations are cached, the cost of doing incremental builds is proportional to the size of the software being built, not to the number of tool executions required. This is especially evident in the case of the Release test, where caching only runtool calls led to so many requests on the function cache that the build time was an order of magnitude larger than when all function calls were cached.

- Although caching model calls in addition to runtool calls does help, substantial performance benefits occur when calls to user-defined functions are also cached. Again, the Release test demonstrates this point well: caching

[3] Recall that a Vesta model represents a one-argument closure that can be called from other functions and models.

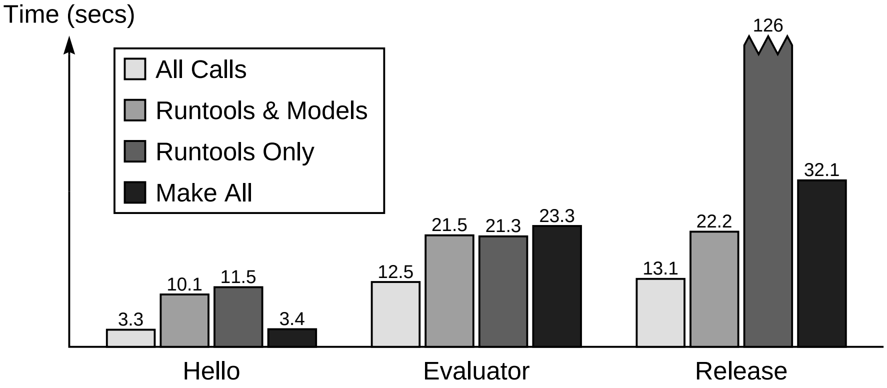


Figure 9.5: Elapsed times of one-module, incremental builds in seconds with different Vesta caching levels. For comparison, the final column shows the incremental build time under Make.

models in addition to runtool calls led to a factor of 5.5 speedup, but caching user-defined function calls as well yielded another speedup factor of nearly 2.

Together, these points show that the benefits of performing fine-grained dependency analysis clearly outweigh its cost.

Recall from Section 7.5 that in addition to the normal cache entry created for each model evaluation, the evaluator also creates a special cache entry. These special cache entries yield faster cache hits, and the overhead required to create them is quite small. In the tests above, such special cache entries accounted for all the cache hits for model evaluations. This result suggests the usefulness of these special cache entries, although we have not measured the system's overall performance without them.

9.2.4 Resource Usage

This subsection reports the CPU usage of the main components of the Vesta system, as well as the memory usage of client processes. More detailed measurements about the function cache and repository are presented in Sections 9.3 and 9.4 below.

CPU Usage

To measure the CPU usage, we wrote a simple script to invoke ps(1) once every second. We used this script to monitor the evaluator, the runtool server, the function cache, and the repository during both scratch and incremental builds. During the

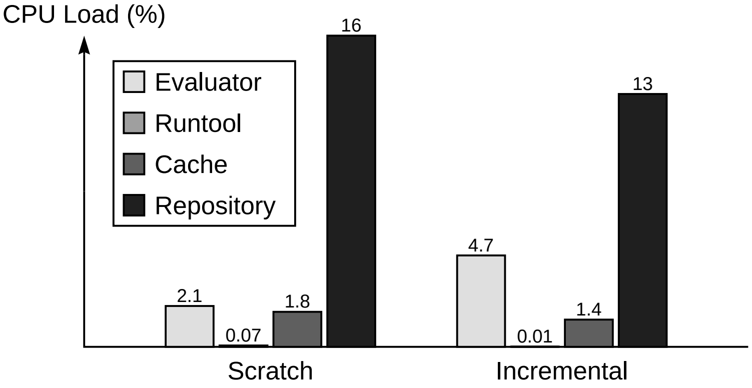


Figure 9.6: The mean CPU loads of the Vesta evaluator, runtool server, function cache server, and repository server during scratch and incremental builds.

builds, there was no other build in progress that was using the same function cache or repository, i.e., the Vesta system was being used by a single client.

The evaluator and the runtool server are run on the client machine. So, the average CPU overhead on the client machine is below 5%. We believe that the CPU load of these two programs should remain about the same when building larger software.

The function cache and repository servers were run on a single dedicated server machine, although they could be run on separate server machines as well. If multiple builds were performed simultaneously, the CPU loads would be higher. Based on the data in Figure 9.6, Vesta running on the relatively old hardware used in these experiments could adequately support 50-100 developers, assuming the function cache and repository were run on separate server machines and that not all developers would be doing builds simultaneously. More powerful server machines could be used to support a larger number of developers. In fact, as we report in the next chapter, Vesta has been used by a team of 130 engineers developing a code base of approximately 700,000 lines, and it held up well under that load.

Client Process Memory Usage

In the typical Vesta system configuration, the evaluator and the runtool server are both run on the client machine. The memory usage of the runtool server is always well below 2MB. The evaluator, on the other hand, can consume somewhat more memory during a large scratch build. To measure the memory usage of the evaluator, the evaluator calls ps(1) right before it exits. Figure 9.7 gives the memory

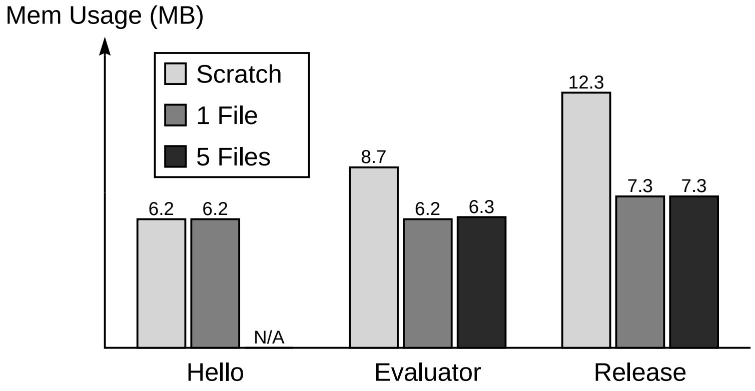


Figure 9.7: The run-time memory usage of the Vesta evaluator on scratch and incremental builds. The last two columns in each case show the incremental results in which 1 file and 5 files were changed, respectively. (The 5-file change test is not applicable in the Hello case because that program consists of only a single file.)

consumption of the evaluator for both scratch and incremental builds of the test programs.

How can these memory requirements be expected to scale for larger builds? For incremental builds, we expect to see similar memory consumption, even for much larger software. As explained in Section 9.2.2, the cost of an incremental build is determined by the magnitude of the change. Since the evaluator's memory consumption is roughly proportional to the size of the call graph it must evaluate, the memory required by the evaluator to do an incremental build is almost completely independent of the size of the software being constructed. For scratch builds, however, it still remains to be seen whether the evaluator will perform well when Vesta is used to build multi-million line software systems.

The memory requirements reported above may be artificially high due to the fact that the evaluator uses a garbage collector for its memory management. The collector uses heuristics to decide when it should grow its heap, and we have observed the current collector growing the heap unnecessarily. We believe we could decrease the evaluator's memory usage for large scratch builds by tuning the collector's heuristics. However, doing so would probably lead to an increased number of collections, which in turn might lead to somewhat slower evaluations.

9.3 Repository Performance

We describe several aspects of the repository performance in this section.

First, we measure the speed of basic file operations through the repository's NFS server interface. This aspect of repository performance is most important during builds, when encapsulated tools make intensive use of the repository to provide file service. If the repository is fast enough to service builds adequately, it will easily meet the lighter demands placed on it by users browsing through the append-only source tree or editing files in mutable source directories. Our measurements show that repository performance is adequate. In comparisons on the same hardware, the repository shows a write data rate of about 96% of a standard NFS server, a read data rate of about 55%, and comparable performance on other operations.

Second, we examine the repository's memory and disk space consumption. We present measurements of the current amount of source code stored in the repository (number of packages, versions, files, etc.), together with a breakdown of the actual memory and disk space used by the server for various purposes. Our approach of keeping directory structures in memory appears successful: a nontrivial but reasonable amount of memory is used, and memory will not grow unreasonably as the amount of source code increases. Our approach to file versioning appears successful too: even though we do not use deltas or other forms of compression, deriveds still take up considerably more disk space than sources, and the space consumed by old versions of sources is inexpensive at today's disk prices.

Finally, we measure the performance of the repository's development cycle tools: **vcheckout**, **vadvance**, **vcheckin**, etc. These tools all run very fast (roughly 0.5 to 1.5 elapsed seconds for the cases tested), making the system pleasant to use.

9.3.1 Speed of File Operations

We measured the speed of the repository as a file server on two commonly used file system benchmarks: the Connectathon '97 Basic Benchmark (CBB) [15] and the Modified Andrew Benchmark (MAB) [28, 40]. CBB is a microbenchmark that tests small groups of related operations. MAB is a higher-level benchmark that measures performance on a software development task. Neither benchmark provides a measurement of file server performance in isolation; both measure overall file system performance, including the buffer cache of the client operating system. Both (especially MAB) can at times be CPU-bound, not I/O-bound. Nonetheless, the benchmarks provide a rough basis for comparison between the repository and an ordinary NFS file server.

For these measurements, we used the standard Vesta configuration described in Section 9.1. The benchmarks either accessed a mutable directory in the user-space repository server via NFS version 2, or (for comparison) accessed a directory in the underlying AdvFS file system through the standard kernel-space NFS version

3 server. In both cases, accesses were over SRC's high-speed ATM network.

Test	Description	AdvFS+NFS3		Vesta+NFS2	
1	file and directory creation: creates 155 files and 62 directories.	6.13	(0.19)	7.32	(0.38)
2	file and directory removal: removes 155 files and 62 directories.	5.56	(0.19)	6.53	(0.36)
3	lookup across mount point: 500 getwd and stat calls.	1.35	(0.07)	1.37	(0.04)
4	setattr, getattr, and lookup: 1000 chmods and stats on 10 files.	11.38	(0.17)	3.04	(0.42)
4a	getattr and lookup: 1000 stats on 10 files.	0.11	(0.02)	0.01	(0.03)
5a	write: writes a 1048576-byte file 10 times.	5.88	(0.57)	6.26	(0.82)
5b	read: reads a 1048576-byte file 10 times.	1.40	(0.02)	2.54	(0.08)
6	readdir: reads 20500 directory entries, 200 files.	5.28	(0.18)	7.27	(0.24)
7a	rename: 200 renames on 10 files.	3.51	(0.13)	6.75	(0.34)
9	statfs: 1500 statfs calls.	1.16	(0.04)	1.18	(0.17)

Table 9.2: Connectathon Basic Benchmark, run on a standard file system through NFS version 3, and on the Vesta repository through NFS version 2. Each table entry is an average elapsed time in seconds; smaller numbers are better. All values are averaged over 20 runs of the benchmark. Standard deviations are included in parentheses.

On the Connectathon Basic Benchmark (Table 9.2), the repository is slightly slower than the kernel NFS server on most operations, but there are substantial differences on a few. Data writes and reads (tests 5a and 5b) are the most interesting. Converting the elapsed times to bytes per second and comparing, we find that the repository's write data rate is 96% of kernel NFS, while its read data rate is 55%.

We have not investigated the reasons for these differences in detail, but it seems likely that writes appear artificially fast under Vesta because the repository cheats, as explained in Section 4.3.5. The repository server currently violates the expected NFS2 semantics by not forcing writes all the way through to disk before returning to the client, so the benchmark does not get the expected write-through on close semantics at user level. The kernel NFS server, on the other hand, implements the NFS3 protocol correctly and thus does provide write-through on close.

Reads likely are slow because the data is really being read from disk, so the extra overhead of going through the user-space repository server is fully visible. The Connectathon read test flushes the client machine's buffer cache before each read.

Curiously, the repository is a great deal faster than kernel NFS on tests 4 and 4a. This may be due to the repository's in-memory directory structure, but in any case the difference is probably unimportant for overall system performance.

Tests 7b and 8 of the benchmark were omitted because they test hard links and symbolic links, neither of which are supported in repository mutable directories.

Phase	Description	AdvFS+NFS3		Vesta+NFS2	
1	Create Directories	947	(175)	756	(116)
2	Copy Files	9286	(366)	5776	(280)
3	Directory Status	3609	(56)	3750	(90)
4	Scan Files	4393	(189)	4414	(112)
5	Compile	22627	(536)	18913	(630)
	Total	40862	(654)	33609	(840)

Table 9.3: Modified Andrew Benchmark, run on a standard file system through NFS version 3, and on the Vesta repository through NFS version 2. Each table entry is an average elapsed time in seconds; smaller numbers are better. All values are averaged over 20 runs of the benchmark. Standard deviations are included in parentheses.

The first phase of the Modified Andrew Benchmark (Table 9.3) creates a tree of directories. The second phase copies a 350 KB collection of C source files into the tree. The third phase traverses the new tree and examines the status of each file and directory. The fourth phase reads every file in the new tree. The fifth phase compiles and links the files. This benchmark does not use the Vesta evaluator or tool encapsulation; here we are comparing only the performance of the repository as a file server, not the overall Vesta system performance on software building.

On this benchmark, surprisingly, the repository is actually somewhat faster than the kernel NFS server in some phases and in the overall time. We have not done the detailed measurements that would be needed to find out for certain why this happens. It seems likely that some of the difference (especially in Phase 1) is due to the repository's in-memory directory structure, and some (especially in Phases 2 and 5) is due to the lack of write-though on close discussed above.

Our purpose in making these measurements was not to understand the repository's NFS performance in detail; it was simply to establish that the performance is adequate for our purpose. The measurements confirm this claim. Even though reading from the repository server takes almost twice as long as reading from an in-kernel NFS server, the actual elapsed time to do builds is slightly less. Both the small, non-encapsulated build measured in the Andrew benchmark and the Vesta builds measured in Section 9.2.1 show this effect.

In the future, we may investigate the possibility of speeding up repository NFS reads and writes by moving them into the kernel. The repository server does no useful work on these operations other than mapping from its file handles (longids) to the correct files in the underlying file system. It would be straightforward to move this functionality into the kernel, getting the user-space server out of the loop for these time-critical operations. All other repository NFS operations would continue to be handled in user space, minimizing the amount of code added to the kernel. However, one big advantage to the current design is that it is far more portable.

9.3.2 Disk and Memory Consumption

For this section, we took a snapshot of our working repository as of October 27, 1997, and made a variety of measurements on its contents. We were using this repository to develop the Vesta system itself. It also contained a few other software packages that we had converted to be built under Vesta as test cases for the repository and evaluator. In particular, it included the Juno-2 constraint-based drawing editor [27, 29] and the many Modula-3 [38, 39] libraries required by Juno-2.

Our measurements do more than indicate how much disk and memory space is required to store the current snapshot; by extrapolation they also give a useful estimate of how space consumption will grow as more packages and versions are added to the repository. We discuss this extrapolation after presenting the basic figures.

The snapshot contains 69 top-level packages and 26 branches. Within these, there are 648 checked-in versions and 10 reservation stubs for versions under development. There are 631 checkout sessions; this includes both active sessions associated with the reservation stubs (10), and old sessions associated with checked-in versions (621). Within these checkout sessions there are 4,981 package versions, for a grand total of 5,629 versions.

Summing over the entire snapshot, including both checked-in versions and checkout sessions, there are 24,556 directories and 407,662 files. That is, the snapshot's externally visible hierarchical namespace includes that many distinctly named directories and files. In the repository's internal DAG structure, many of these directories and files share storage. If all 407,662 files were stored without sharing, they would occupy 12,076,631,120 bytes, or (considering fragmentation) 12,018,130 1-KB blocks (about 11.7 GB).

The mutable portion of the repository was also included in the snapshot. It contains 29 directories and 449 files; again, some of these files may share storage with each other or with immutable files. If stored without sharing, the files would occupy 2,986 1-KB blocks (about 2.9 MB).

The repository's internal sharing of storage between identical files shrinks these numbers greatly. In reality, only 10,856 distinct source files exist in the snapshot's shortid pool, occupying only 269,030 1-KB blocks (about 263 MB).

How much disk space is being spent on old versions and branches? To get an approximate answer to this question, we look at only the latest version of each package and each branch. There are 66 latest top-level versions (not 69, because a few packages have no versions), below which are 445 directories and 4,367 immutable files; if stored without sharing, these files would occupy 93,391 1-KB blocks (about 93 MB). Since only one version from each package is counted here, there is in fact little or no sharing, so we can take these figures as the actual amount of space consumed. Thus $269,030 - 93,391 = 175,639$ 1-KB blocks (about 172 MB) are consumed storing old versions and branches. To summarize, about 35% of the source disk space is spent storing latest versions, and about 65% is spent on old versions and branches.

The snapshot was taken immediately after a weed that kept only the deriveds produced by evaluating the latest version of each package, branch, and checkout session; all other deriveds were deleted. This weed left the snapshot with 2,958 derived files. (Some files can be both sources and deriveds; we omit those from this count, including them only as sources.) These deriveds occupy 288,367 1-KB blocks (about 282 MB). Thus, across the entire pool of sources and deriveds, about 17% of the space is consumed by latest source versions, 31% by old versions and branches, and 52% by deriveds.

Of course, if the snapshot had not been taken just after a weed, the proportion of deriveds would be much higher. We tend to wait until our disk (about 4 GB) is nearly full before running a weed. At this point over 93% of the disk is occupied by deriveds.

Because the repository keeps all of its directories in memory (see Section 4.3.2), we also measured the amount of memory required to store the snapshot. The repository's packed, garbage-collected memory pool used 3.0 MB to store immutable directories, 0.34 MB to store appendable and mutable directories, 0.44 MB to store mutable attributes, and a few tens of kilobytes for other structures. A checkpoint that was taken just before the snapshot is 3.77 MB long; this checkpoint is an exact image of the repository's runtime memory pool and encodes its complete state. In addition, 0.24 MB of the repository's general-purpose heap (memory allocated by `new` in C++) were consumed by an inverted index that maps from immutable directory shortids to the in-memory directory data structures (see Section 4.3.1);

this index was not included in the checkpoint because it can be reconstructed at recovery time. Thus, 4.0 MB were used to store 24,585 directories (and related structures such as attributes). The average cost was about 171 bytes per directory.

The repository saves memory as well as disk space by storing the hierarchical name space internally as a DAG. Although there are 24,585 directories with distinct hierarchical names, internally the snapshot contains only 8,411 directory data structures (7,560 immutable and 851 appendable or mutable). Within those directories, additional space is saved by recording directories as lists of changes relative to other directories where possible. Although there are 432,696 files and directories with distinct hierarchical names, and hence an average of 432,696/24,585 = 17.60 entries in each external directory, the internal directory data structure contains only 85,650 entries, for an average of 85,650/8411 = 10.18 entries per internal directory. Internal directories can also contain placeholder entries for objects that have been deleted; the snapshot contained 1,866 of these, an average of about 0.22 per internal directory.

An internal directory consists of one or more blocks (usually just one), each containing a fixed 34-byte header, a packed list of entries, and (for appendable and mutable directories) some optional space for expansion. The snapshot used 8,502 blocks to store the 8,411 directories, for a total of 0.28 MB of header (an average of 34 bytes each), and a total of 0.21 MB of expansion space in the 851 appendable and mutable directories (an average of about 258 bytes each). An entry contains a pathname arc plus either 10 or 26 bytes of overhead (depending on whether it points to a directory or file, respectively). The snapshot used a total of 2.8 MB to store active directory entries and an additional 29.0 KB to store placeholder entries for deleted objects. The average cost was 33.78 bytes per entry.

Stepping back from these details, what do we learn about the repository's overall memory and disk consumption, and how can we expect it to grow as more sources are stored?

Disk consumption for sources is proportional to the number (and size) of distinct source file versions. After about a year of developing Vesta itself under Vesta, the data above shows about a threefold expansion in disk storage by storing all package versions instead of keeping only the most recent version of each package. Even with this expansion, and even after a weed that kept only one version of the deriveds, deriveds occupy more disk space than sources. Of course, we would expect the factor of three to grow a bit worse on a source pool that was worked on for a longer time and had more active branches. Still, considering today's rapidly falling disk prices, and the fact that humans are not learning to type in new source code at correspondingly higher rates, we believe that the disk space required to store old source versions is not a problem in Vesta.[4]

Memory consumption is more of a concern, but here again it is roughly proportional to the number of distinct source file versions. Each new source file version added to the repository requires at most a small constant number of directories and directory entries to be added to the data structure. A conservative estimate, roughly valid if a separate **vadvance** is used to insert each new file version, is one new directory and two new directory entries per new file version, for a total of about 100 bytes. Thus, a repository server machine with 200 MB of physical memory (not at all unreasonable at today's memory prices) could hold the directory structure for some two million source file versions. Moreover, the repository will still run correctly with less physical memory; it will simply run more slowly due to paging. As memory prices continue to drop, larger repositories will become feasible. There are also a few opportunities to shrink the present data structures: eliminating the per-directory expansion space would save almost 10% with essentially no loss in performance,[5] and more savings would be possible by trading time for space.

9.3.3 Speed of Repository Tools

To evaluate the speed of the repository tools, we ran a simple benchmark. Each step of the benchmark applied the same operation once each to 10 separate packages. Every repository tool took well under 1.5 seconds to run, and most took 0.5 seconds or less. Steps that copied new source code into the repository, or modified an existing source file for the first time (triggering a copy-on-write), took longer but were in line with the performance measurements of Section 9.3.1 above.

Here are the steps of the benchmark in detail.

1. Run **vcreate** to create a new package.

2. Run **vcheckout** to create a checkout session for the empty package.

3. Copy the repository's own source code from an ordinary (NFS-mounted) file system into the new working directory for the package. The source code consists of 92 files, containing 835,120 bytes or 876 1-KB blocks.

4. Run **vadvance** to install the source as the first version of the checkout session.

[4] In the Vesta-1 repository, we implemented an optional feature that could compress old source versions by encoding them as deltas, similar to the representation used by RCS. When we put the system into daily use, however, we found that 80% of our disk space was typically occupied by derived files [13]. Thus, delta-compressing the source files would have saved only a small percentage of our available disk space, so we never bothered to turn on the feature, and we omitted it from Vesta-2 entirely.

[5] In fact, we made this change in a later version of Vesta than the one measured here.

5. Run **vadvance** again. In this case **vadvance** detects that the working directory has not changed and does not create a new version.

6. Touch (modify) one file in the package, triggering a copy-on-write. The chosen file was 79,357 bytes long.

7. Run **vadvance**.

8. Touch (modify) all the files in the package, triggering a copy-on-write for every one.

9. Run **vadvance**.

10. Run **vcheckin**, installing the final version from the checkout session as version 1 of the package and deleting the working directory.

11. Run **vcheckout** on the package, creating a new checkout session initialized from version 1.

12. Again touch one file, same as step 6.

13. Run **vadvance**.

14. Yet again touch one file, same as step 6.

15. Run **vadvance**.

16. Run **vcheckin**, installing the final version from the checkout session as version 2 of the package and deleting the working directory.

The entire benchmark was run 5 times, each time on 10 packages. The results are shown in Table 9.4. The average time taken per package is given for each step, rounded to two significant digits.

We are very satisfied with the tools running at this speed. Subjectively, they are more than fast enough to make the system pleasant to use.

9.3.4 Speed of Cross-Repository Tools

Several years after taking the measurements in the previous section, we repeated the benchmark to compare the performance of the tools on the single-repository and cross-repository (Section 4.4.9) cases. When the measurements in this section were taken, both the computers and the networks used for the earlier measurements had been replaced with faster hardware. We omitted steps 14 and 15 of the earlier benchmark in this run, since they are purely local and duplicate earlier steps.

Step	Description	Time per package
1	create an empty package	260 ms
2	check out the new package	560 ms
3	copy in 835 KB of source code	12000 ms
4	advance the package	1400 ms
5	advance again (no changes)	120 ms
6	touch a 79 KB file	110 ms
7	advance the package	200 ms
8	touch all 92 files in the package	7600 ms
9	advance the package	1300 ms
10	check in the package	470 ms
11	check out the package	660 ms
12	touch a 79 KB file	140 ms
13	advance the package	180 ms
14	touch a 79 KB file	120 ms
15	advance the package	190 ms
16	check in the package	460 ms

Table 9.4: Vesta repository tool performance. The entire benchmark was run 5 times, each time on 10 packages. The average time taken per package is given for each step, rounded to two significant digits.

Table 9.5 shows the results of the cross-repository performance benchmark. In this test, the steps listed were run in order, 50 times each on 50 separate packages. The table gives the average time for each step, rounded to two significant figures. The *Local* column is the single repository case, *Nearby* is the cross-repository case where the local and remote repositories are connected by a single hop of gigabit ethernet, and *Distant* is the cross-repository case where the repositories are on opposite coasts connected by ten hops through a corporate intranet. Note that copying, touching, and advancing are always local operations. Each repository was running on a 500 to 600 MHz Alpha 21164A processor. In each case, the tools were run on a client workstation with a 667 MHz Alpha 21264A processor, connected to the local server by 100 Mb ethernet. As the table shows, the tools are very fast in the local and nearby cases, and fast enough to be usable even in the distant case.

Comparing the figures from the older tests of Table 9.4 done on slower hardware with the newer ones in Table 9.5, the tools have speeded up on most tests, but appear notably slower on advance tests 4 and 9. There are two reasons for the difference: first, the tests were done on a larger package (835 KB vs. 1204 KB),

Step	Description	Local	Nearby	Distant
1	create an empty package	50 ms	250 ms	6400 ms
2	check out the new package	64 ms	590 ms	5700 ms
3	copy in 1204 KB of source code	5300 ms	5400 ms	5200 ms
4	advance the package	2500 ms	2600 ms	2500 ms
5	advance again (no changes)	170 ms	170 ms	180 ms
6	touch a 108 KB file	34 ms	32 ms	30 ms
7	advance the package	160 ms	180 ms	180 ms
8	touch all 92 files in the package	3200 ms	3300 ms	3200 ms
9	advance the package	2500 ms	2500 ms	2600 ms
10	check in the package	110 ms	780 ms	18000 ms
11	check out the package	49 ms	730 ms	5800 ms
12	touch a 108 KB file	49 ms	73 ms	59 ms
13	advance the package	150 ms	160 ms	160 ms
14	check in the package	64 ms	170 ms	4700 ms

Table 9.5: Vesta repository tool performance, comparing the local (single-repository) case with two cross-repository cases, one where the remote repository is nearby and the other where it is distant.

and second, the measurements in Table 9.4 were taken before we implemented fingerprinting files by content (Section 4.2.3), so they do not include fingerprinting costs.

9.3.5 Speed of the Replicator

Table 9.6 gives a few simple measurements of the Vesta replicator **vrepl** described in Section 4.4.8. Like the cross-repository tool measurements, these measurements were taken several years later than the others in this chapter, and both the computers and the networks used for the earlier measurements had been replaced with faster hardware. For the following measurements, each repository was running on a 500 to 600 MHz Alpha 21164A processor. In the *Nearby* cases, the two repository host machines were directly connected via gigabit ethernet. In the *Distant* cases, the two machines were on opposite coasts, connected via 10 hops through a corporate intranet. The replicator itself was run on a client workstation with a 667 MHz Alpha 21264A processor, connected to the local server by 100 Mb ethernet. As a rough point of comparison, the *rcp* columns give the time to copy the same files directly out of the native file system that the repository is built on top of. All times are averaged over three trials and rounded to two significant figures.

		vrepl		rcp	
Replicate	Size	Nearby	Distant	Nearby	Distant
+repos/124 to empty repository	1.2 MB	1.7 s	45 s	17 s	57 s
+repos/125 with 124 present	582 KB	0.6 s	14 s	1.6 s	14 s
@repos/124 to empty repository	127 MB	140 s	3200 s	620 s	2600 s
@repos/125 with 124 present	640 KB	9.6 s	120 s	1.4 s	13 s

Table 9.6: Vesta replicator performance.

9.4 Function Cache Performance

9.4.1 Server Performance

Recall from Section 6.3 that looking up an entry in the function cache is a two-step process. In the first step, the evaluator invokes the cache's SecondaryNames function to learn the set of all secondary dependency names associated with a given primary key (PK). In the second step, it invokes the cache's Lookup function. In the event that there are no entries associated with the primary key passed to the SecondaryNames function, the Lookup call is skipped. In the event of a cache miss, the evaluator invokes the cache's AddEntry function to create a new entry and add it to the cache.

We measured the elapsed time spent in the function cache server process to handle various RPC requests from the client. These measurements were made by performing several identical experiments. Each experiment consisted of performing a scratch build of the Vesta evaluator (starting from a nearly empty function cache in which only the standard environment had been built), followed by five incremental builds of the evaluator triggered by touching a single evaluator source file. The mean times for each operation were calculated for each experiment; the means and standard deviations of those mean times are reported in Table 9.7.

Operation	Number of Calls	Mean Time (ms)	Std. Dev. (% of Mean)
SecondaryNames	518	16.8	0.63%
Lookup	112	11.7	0.44%
AddEntry	438	8.1	0.24%

Table 9.7: Elapsed times of various function cache server operations.

A more detailed analysis reveals that misses account for 34% of the Lookup calls and take 17.8 ms on average. In this experiment, 90% of the hits were to cache entries in memory, taking only 6.3 ms each, while the remaining hits to cache entries on disk took 30.3 ms on average. The limiting factor in cache operations seems to be disk latency. We suspect the function cache and repository may be competing for CPU and disk. Such contention could be eliminated by running the processes on separate server machines and by storing their files on separate file systems.

The performance results above are for a relatively small build. As another experiment, we performed a scratch build of the entire Vesta release followed by incremental builds of newer and older versions of the entire Vesta release. The number of cache operations in this second experiment was significantly larger, as was the number of new entries added to the cache. Table 9.8 shows the results.[6]

Operation	Number of Calls	Mean Time (ms)
SecondaryNames	4,351	29.0
Lookup	763	44.0
AddEntry	3,948	21.6

Table 9.8: Elapsed times of various function cache server operations for a scratch build of the complete Vesta release followed by two incremental builds.

Comparing Tables 9.7 and 9.8, we see that cache performance does degrade as the number of cache entries is increased. Although we do not have enough data to meaningfully extrapolate how the function cache will perform for even larger builds, the results presented in Section 9.2.2 indicate that the function cache performs well for the medium-size scratch and incremental builds we have performed.

9.4.2 Stable Cache Attributes

Recall from Section 6.5 that cache entries are partitioned first into PKFiles, and then into CFP groups; only the entries in a single CFP group need be consulted on each Lookup operation. To determine the effectiveness of this organization, we measured various attributes of the function cache's stable cache entry files.

The mean values of various function cache attributes are shown in Table 9.9. These statistics were measured from a stable cache containing 13,900 cache entries distributed over 10,183 PKFiles. The total disk space required to store these entries was 112 MB, or 8.2KB per cache entry.

The statistics indicate that the separation of entries into CFP groups is working well: the average number of entries per CFP group is only slightly more than 1. Moreover, of the average 52.4 secondary names per PKFile, more than 99% are

[6] This table does not include standard deviations because the experiment was performed only once.

Attribute	Mean Value
Number of CFP groups per PKFile	1.33
Number of entries per CFP group	1.02
Cache entry size (Kbytes)	8.24
Function result value size (Kbytes)	5.82
Number of secondary names per PKFile	52.4
Percentage of common names per PKFile	99.1%
Percentage of uncommon names per cache entry	2.5%

Table 9.9: Mean values of various function cache attributes.

common. Thus, only a very small number of fingerprints need be compared during a typical lookup operation.

9.4.3 Resource Usage

This subsection reports on the function cache's memory and disk usage; its CPU usage was reported earlier in Section 9.2.4.

The cache server's memory requirements are dominated by the storage used for in-memory cache entries. Much like a virtual memory system, the function cache flushes unused cache entries to disk over time. Its policy can be adjusted to flush entries more aggressively if too many entries are being retained in memory.

As shown in Table 9.9, the average function result value alone is nearly 6KB in size. The function result value is the dominant factor in the cache's memory use. Hence, on a machine with 100MB of available main memory, the cache could keep about 18,000 entries in memory simultaneously. Considering that a build of the entire Vesta system (including the complete set of Modula-3 libraries) generates only about 3,500 cache entries, the function cache's memory usage should scale up well to larger projects.

The function cache uses the same garbage collector as the evaluator, so as described in Section 9.2.4, its actual usage may be somewhat more than strictly needed. The comments in that section about tuning the collector's heuristics apply equally well to the function cache.

The function cache's disk requirements per cache entry are larger than its memory requirements. The extra costs are due to metadata—such as the PKFile and CFP group data structures—and to extra information stored with stable cache entries that allows them to be efficiently shuffled into new CFP groups when a PKFile's set of common names changes. On average, a cache entry requires roughly 8.25KB of disk space. To accommodate the design target of 12 million cache entries, the function cache would thus require roughly 95GB of disk storage.

9.4.4 Scalability

In this subsection, we describe potential scalability bottlenecks in the function cache, and the steps we have taken to avoid them.

- **Lock Contention.** As the number of clients increases, overall function cache performance might degrade due to lock contention on the cache's in-memory data structures. To avoid this problem, the function cache uses relatively fine-grained locking: there is a separate lock on each in-memory PKFile and its cache entries. Moreover, the lock on the cache's central structures is held as briefly as possible in the places where it is required.

- **Bad Cache Entry Distribution.** The function cache's performance will suffer if the number of cache entries per CFP group grows too large. However, due to the function cache's scheme of dividing PKFiles into CFP groups (as described in Section 6.5), we do not expect this problem to be serious. When the problem has arisen in the past, it was usually due to a flaw in the evaluator's caching strategy, or it could be corrected by changing bridge models so the evaluator computed better primary keys for their functions. The VCacheStats program that was used to gather the statistics for Table 9.9 can also be used to uncover skewed cache entry distributions, so such problems are easily detected.

- **Disk Latency.** As the number of cache entries per PKFile increases, the files grow larger and more time may be spent waiting on disk reads during lookups. Two factors mitigate this effect. First, we expect that an increased number of entries per PKFile will account for at most one of the two orders of magnitude in cache entry growth cited above. (The other order of magnitude will appear in the form of an increased number of PKFiles.) Second, the PKFile disk format has been designed to minimize the number of disk reads required per lookup. In particular, the index of CFP groups is stored separately from the cache entries in the hope that the entire index can be read in a single disk operation. Similarly, extra cache entry information not required for lookup is stored separately from the entries, again to decrease the number of read and seek operations in a typical lookup.

- **Memory Usage.** There is an obvious time-space tradeoff between the number of cache entries kept in memory and the speed of a typical lookup. As the number of cache entries and clients increases, a smaller fraction of the "working set" entries can be kept in memory. One solution to this problem is simply to install more memory on the server machine. But even if no entries were kept in memory, the detailed results described in Section 9.4.1 above imply that the overall function cache performance would not degrade badly.

9.5 Weeder Performance

In evaluating the performance of the weeder, two measures are of interest. First, we consider how often weeds are necessary. Second, we consider how long it takes the weeder to run. Because weeding is a background process, both issues are nearly invisible to users. However, some administrative effort is needed to run the weeder (or to set it up to run automatically), and builds are likely to run a bit more slowly while the weeder is active and competing for resources, so these performance issues are worth considering.

How often are weeds necessary? The answer depends on how quickly the disk used by the repository and function cache fills up, which in turn is a function of the disk's size, the rate at which new cache entries and deriveds are created, and the sizes of those cache entries and derived files on disk. Our experience, in which three developers were actively doing builds against a 4GB disk, is that weeding was required about once every two weeks.

The weeding experience of the larger Compaq engineering group discussed in the next chapter provides another data point. They were using a 100 GB disk cluster, and their builds produced roughly 10 GB of derived files per day. They wrote a cron job that ran every night, and that automatically performed a weed whenever the disk usage passed a predetermined threshold. This same job also ran automatically during the day with a higher threshold to make sure that a spike in the rate of disk consumption would not fill the disk before the next nightly run. Weeds generally occurred once or twice a week.

How long does it take the weeder to run? Recall from Chapter 8 that the weeder runs in two phases: a mark phase in which the cache entries and derived files to keep are discovered, followed by a deletion phase in which the function cache and repository do cache entry deletion and derived file deletion in parallel. In our experience, the deletion phase takes significantly longer than the mark phase. In particular, the deletion phase takes on the order of 10–15 minutes when weeding against a 4GB disk. A weed by the Compaq engineering group against their 100GB disk generally took about an hour. We expect the deletion time will scale linearly with the number of cache entries being deleted, which in turn is bounded linearly by the size of the backing disk.

The performance of the weeder's mark phase is a function of the number of graph log entries buffered in memory: the smaller the buffer, the more passes over the graph log are required. As one point along the time-space curve, we found that weeding a graph log containing over 30,000 entries using a buffer of 10,000 entries required four scans of the disk file and 10 seconds of elapsed time. As the size of the graph log increases, more scans of the disk file will be required, but we expect that the entire mark phase will require no more than ten minutes on a cache containing one million cache entries.

9.6 Interprocess Communication

For interprocess communication, Vesta uses SRPC, a simple RPC (remote procedure call) protocol and library for C++ that we implemented on top of TCP sockets. SRPC does not include an automatic stub generator, so all of our marshaling and unmarshaling stubs were written by hand. To simplify stub coding, SRPC provides methods for sending and receiving common data types like integers, null-terminated strings, arrays of bytes, and sequences. The SRPC implementation uses TCP keepalives to detect network partitions and other connection failures.

Except where noted in the more recent cross-repository experiments, all of the tests described above were run on our local AN2 network. AN2 is a switched Asynchronous Transfer Mode (ATM) network that uses 155 megabit per second (Mbit/sec) links. On this network, our RPC package performs as follows. The round-trip elapsed time for a null RPC (i.e., no arguments or results) is 1.2 milliseconds on average. In simple tests performed over varying argument and result sizes, clients send arguments at an average rate of 75 to 100 Mbit/sec, and receive results at an average rate of 65 to 132 Mbit/sec.

Chapter 10

Conclusions

The goal of the Vesta project has been to provide a solid base for software configuration management by dealing well with the core problems of source management and building, in a way that scales up to very large software projects (tens of millions of lines of code), yet also works well for smaller ones.

To the extent that we are able to evaluate it, Vesta has met its goal. The source management system preserves source code immutably and immortally, supports both simple linear versioning and arbitrarily complex branching for parallel development, makes all versions directly accessible through the file system, and provides very fast checkout and checkin using copy-on-write. It supports distributed development with source replication, cross-repository checkout and checkin, and cross-realm access control. The build system provides repeatable, incremental, and consistent builds. Despite providing much stronger guarantees on build consistency, it runs as fast as Make for scratch builds, and faster for incremental builds. The builder also supports parallel builds on multiple machines. The system modeling language is modular, flexible, and general. It allows the description of a large, complex software system to be broken down into small, relatively simple modules. It allows new build tools to be integrated by coding within the modeling language, with no need to modify the base system.

For over two and a half years, Vesta was in daily use by Compaq's Araña group, a team of about 130 developers working on a large microprocessor design. The team was split into two subgroups, one in New England and the other in California, each with its own repository. Both the chip design itself and the team's custom design software were stored in the repositories and processed using the build system. The final code base consisted of about 700,000 lines. We found this experience highly valuable in validating the Vesta design, shaking out bugs in the implementation, and exposing the need for various small features that were not initially anticipated.

The Araña group in turn was pleased with Vesta. They have stated that its strong support for parallel source development and reproducible builds saved them considerable time (3 to 6 months in the architectural design phase alone), and that the distributed development features provided answers to some extremely difficult problems they faced in bicoastal software and design database management. They also found Vesta to be extremely useful for tracking down difficult bugs, and expected it to become even more valuable as the project drew closer to completion.

Some questions do remain, including scalability, generality, ease of use, and conversion and learning costs.

The Vesta implementation shows excellent performance when managing and building the software projects we have tested it on, but our goal was to support projects at least an order of magnitude larger. We have argued at appropriate points throughout this report that our algorithms and implementation structure can be expected to scale up well, but our case cannot really be convincing until the system is put into use on the scale it was intended for. In the future we hope to further test Vesta's scalability by finding additional users in the free software arena. Ports to Alpha Linux and 32-bit Intel Linux have been completed, and Compaq has agreed to release the system's source code to the public under the LGPL [19]. The complete sources will soon be made available through the Vesta web site [51].

The Vesta builder always builds derived objects from source, but does its own caching of build steps to save work. This paradigm limits the builder's generality somewhat, because some software tool sets do not work best when building directly from source and giving Vesta control over each individual step. For example, an incremental linker increases the speed of linking by modifying a previously linked program, replacing only the parts that have changed, rather than relinking the entire program. Some language compilers (such as the Modula-3 compiler) work most efficiently when given all the source files that go into a program at once; this allows them to parse interface files only once and cache the results for reuse, rather than reparsing them for each implementation file that uses them. Some build tools may ask for human intervention during the build process. Generally, Vesta can be used with tools of these kinds by bypassing the incremental features and scripting any human interaction that would otherwise be required, but there can be a performance penalty in doing so. For example, the Araña group had one tool that operated in an incremental mode, and in fact it was in the process of being modified to have even more incremental features at the time they began to adopt Vesta. Although disabling these features made the tool slower, the group decided that it was worthwhile paying this performance cost in order to get Vesta's other benefits. Ultimately they were able to modify the tool to work better with Vesta and get back a good deal of the lost performance.

We have found Vesta fairly easy to use for our own software development activities, as did the Araña group. Once Vesta was mature enough to be used to develop itself, we happily switched to it and never wanted to go back. Our experience with Vesta-1 also taught us much; Vesta-2 preserves the strengths of Vesta-1 while correcting its major deficiencies, both in ease of use and performance. Some pieces that could make Vesta-2 easier to use are missing at this writing, however, including a graphical user interface, tools to synthesize simple models automatically, and tutorial user documentation. The Araña group developed their own specialized versions of each of these items for their own environment, however, and in the future we may be able to generalize some of them for use by others.

Because Vesta represents a comprehensive new approach, software development groups will incur some costs in switching to it. Groups with large software bases already under development using existing configuration management systems will need conversion tools. (In working with the Araña group, we were fortunate to come in at the beginning of the project, when very little of the chip design code base had yet been written.) We have several unpublished designs for tools to ease the conversion from Makefiles to models, and designing tools to convert from RCS, CVS, and the like to the repository would be fairly straightforward. However, we have not implemented such tools to date. There will also be a training cost in learning to use Vesta. We have tried to design the repository tools to be simple and intuitive to use, but they do differ somewhat from existing systems. A more serious concern is the modeling language. We have greatly simplified and improved the language since Vesta-1, and we have made sure that the models users normally see and write are little more than lists of source files with a few lines of boilerplate. Yet we anticipate that the language may still be an obstacle to acceptance; naive users will find it unfamiliar, and sophisticated users will have a great deal of learning to do if they need to modify the complex models in the standard construction environment. (Several Araña users did do this successfully; a few became quite expert at writing models.)

In summary, the research phase of the Vesta project is complete, the system has seen serious use within Compaq for more than two years, and we are now on the verge of releasing it as free software for external use. We believe the system has many strengths and will serve its future users well.

Appendix A

The Vesta System Description Language

A.1 Introduction

This appendix describes the formal syntax and semantics of the Vesta-2 System Description Language (SDL). We expect it will be used as a reference by Vesta users. Because this description is meant to be complete and unambiguous, its treatment is rather formal. A less formal language reference is available [45].

In Vesta, the instructions for building a software artifact are written as an SDL program. Evaluating the program causes the software system to be constructed; the program's result value typically contains the derived files produced by the evaluation.

The Vesta SDL is a functional language with lexical scoping. Its value space includes Booleans, integers, texts, lists (similar to LISP lists), sequences of name-value pairs called *bindings*, closures, and a unique error value.

The language is dynamically typed; that is, types are associated with run-time values instead of with static names and expressions. Even without static type checking, the language is strongly typed: an executing Vesta program cannot breach the language's type system. The expected types of parameters to language primitives are defined, and those types are checked when the primitives are evaluated. The language includes provisions for specifying the types of user-defined function arguments and local variables, but these type declarations are currently unchecked.

The language contains roughly 60 primitive functions. There is a `_run_tool` primitive for invoking external tools like compilers and linkers as function calls. External tools can be invoked from Vesta without modification.

Conceptually, every software artifact built with Vesta is built from scratch, thereby guaranteeing that the resulting artifact is composed of consistent pieces. Vesta uses extensive caching to avoid unnecessary rebuilding. Vesta records software dependencies automatically. The techniques by which the implementation caches function calls and determines dependencies are described in Chapters 6 and 7 of this report.

A.2 Lexical Conventions

This section defines the meta-notation and terminals used in subsequent sections. Section A.3 introduces each language construct by giving its syntax and semantics. The syntax of the complete language is given in Section A.4.

A.2.1 Meta-notation

Nonterminals of the grammar begin with an uppercase letter, are at least two characters in length, and include at least one lowercase letter. Except for the four terminals listed in Section A.2.2 below, each of which denotes a class of tokens, the terminals of the grammar are character strings not of this form.

The grammar is written in a variant of BNF (Backus-Naur Form). The metacharacters of this notation are:

```
  ::=   |   [   ]   {   }   *   +   ‘   ’
```

The meaning of the metacharacters is as follows:

NT ::= *Ex*	Non-terminal *NT* rewrites to expression *Ex*
Ex1 | *Ex2*	*Ex1* or *Ex2*
[*Ex*]	optional *Ex*
{ *Ex* }	meta-parentheses for grouping
*Ex**	zero or more *Ex*’s
*Ex**,	zero or more *Ex*’s separated by commas, trailing comma optional
*Ex**;	zero or more *Ex*’s separated by semicolons, trailing semi optional
Ex+	one or more *Ex*’s
Ex+,	one or more *Ex*’s separated by commas, trailing comma optional
Ex+;	one or more *Ex*’s separated by semicolons, trailing semi optional
‘*s*’	the literal character or character sequence *s*

When used as terminals, square brackets, curly brackets, and vertical bar appear in single quotes to avoid ambiguity with the corresponding metacharacters (i.e., ‘[’, ‘]’, ‘{’, ‘}’, ‘|’).

A.2.2 Terminals

The following names are used as terminals in the grammar. They denote classes of tokens, and are defined precisely in Section A.4.3.

`Delim` A pathname delimiter. Either forward or backward slashes are allowed within pathnames, but not both.

`Integer` An integer, expressed in either decimal, octal, or hexadecimal.

`Id` An identifier. An identifier is any sequence of letters, digits, periods, and underscores that does not represent an integer. For example, foo and 36.foo are identifiers, but 36 and 0x36 are not.

`Text` A text string. Texts are enclosed in double-quotes. They may contain escape sequences and spaces.

Comments and white space follow C++ conventions. A comment either begins with // and ends with the first subsequent newline, or begins with /* and ends with */ (the latter form does not nest). Of course, these delimiters are only recognized outside text literals. White space delimits tokens but is otherwise ignored (except that the Space character, the ASCII character represented by the decimal number 32, is significant within text literals). The grammar prohibits white space other than the Space character within text literals.

The names of the built-in functions begin with an underscore character, and the identifier consisting of a single period (i.e., ".") plays a special role in the Vesta SDL. It is therefore recommended that Vesta programs avoid defining identifiers of these forms.

A.3 Semantics

The semantics of programs written in the Vesta SDL are described by a function *Eval* that maps a syntactic *expression* and a *context* to a *value*. That is, Eval(E, C) returns the value of the syntactic expression E in the context C. In addition to syntactic expressions (denoted by the non-terminal Expr in the grammar), the domain of Eval includes additional syntactic constructs. Some of these additional constructs are defined by the concrete grammar, while others are introduced as "intermediate results" during the evaluation process. The latter are noted where they are introduced. Each value returned by Eval is in the Vesta value space, described in the next section. The context parameter C to Eval is a value of type t_binding in the Vesta value space.

Type name	Values of the type
t_bool	*true, false*
t_int	integers
t_text	arbitrary byte sequences
t_list	sequences of zero or more arbitrary values
t_binding	sequences of zero or more (*name*, *value*) pairs
t_closure	closures, each of which is a triple $\langle e,\ f, b\rangle$
t_err	*err*
t_value	union of all of the above types

Table A.1: The types and values of the Vesta language.

A.3.1 Value Space

Values are typed. The types and values of the language are shown in Table A.1.

The values *true*, *false, emptylist* (the list of length zero), *emptybinding* (the binding of length zero), and *err* are not to be confused with the language literals TRUE, FALSE, <>, [ ], and ERR that denote those values.

The type t_bool contains the Boolean values *true* and *false*, denoted in the language by the literals TRUE and FALSE.

The type t_int contains integers over at least the range -2^{31} to $2^{31}-1$; the exact range is implementation dependent.

The type t_text contains arbitrary sequences of 8-bit bytes. This type is used to represent text literals (quoted strings) in SDL programs as well as the contents of files introduced through the Files nonterminal of the grammar. Consequently, an implementation must reasonably support the representation of large values of this type (millions of bytes), but is not required to support efficient operations on large text values.

The type t_list contains sequences of values. The elements of a list need not be of the same type.

The type t_binding contains sequences of pairs (t_i, v_i), in which each t_i is a non-empty value of type t_text, each v_i is an arbitrary Vesta value (i.e., of type t_value), and the t_i are all distinct. Note that bindings are sequences: they are ordered. The *domain* of a binding is the sequence of names t_i at its top level. Bindings may be nested.

Bindings play an important role in the Vesta language. They are used to represent a variety of interesting objects. For example, flat bindings that map names to texts can be used to represent command-line switches and environment variables; bindings that contain nested bindings can be used to represent file systems;

and bindings that map names to closures can be used to represent interfaces. Section A.3.4.5 describes the primitive functions and operators for manipulating bindings, including three primitives for combining two bindings.

The type t_closure contains closure values for the primitive operators and functions (defined in Section A.3.4) as well as for user-defined functions. In a closure $\langle e, f, b \rangle$:

- e is a function body (i.e., a Block as per the grammar below);
- f is a list of pairs (t_i, e_i), where t_i is a t_text value (a formal parameter name) and e_i is either the distinguished expression ⟨emptyExpr⟩ or an Expr (a default parameter value); and
- b is a value of type t_binding (the closure context).

The type t_err consists of the single distinguished value *err*, denoted in the language by the literal `ERR`. Programmers can use this value as they choose; it has no predefined semantics.

A.3.2 Type Declarations

The language includes a rudimentary mechanism for declaring the expected types of values computed during evaluation. The grammar defines a small sub-language of type expressions, which includes the ability to give names to types and to describe aggregate types (lists, bindings, functions) with varying degrees of detail. Type expressions may be attached to function arguments and results and to local variables, indicating the type of the expected value for these identifiers and expressions during evaluation.

The Vesta evaluator currently treats type names and type expressions as syntactically checked comments; it performs no other checking. Future implementations may type-check expressions at run time and report an error if the value does not match the specified type (according to some as yet unspecified definition of what it means for a value to "match" a type specification).

The syntax fragments and semantic descriptions in subsequent sections omit any further reference to type expressions entirely.

A.3.3 Evaluation Rules

The evaluation of a Vesta program corresponds to the abstract evaluation:

```
Eval( M([]) , C-initial)
```

```
class val {
  public:
    operator int();
    // converts Vesta t_int or t_bool to C++ int

    val(int);
    // converts a C++ integer to a Vesta t_int

    int operator== (val);
    // compares two Vesta values, returning true (1)
    // if they have the same type and are equal, and
    // false (0) otherwise
}
```

Table A.2: A C++ class declaration for Vesta values.

```
static val true;         // value of literal TRUE
static val false;        // value of literal FALSE
static val emptylist;    // value of literal < >
static val emptybinding; // value of literal [ ]
static val err;          // value of literal ERR
```

Table A.3: Definitions of constants used by the pseudo-code.

where *M* is the closure corresponding to the contents of an immutable file (a system model) in the Vesta repository and *C-initial* is an initial context. *M*'s model should have the syntactic form defined by the nonterminal Model described in Section A.3.3.13 below. *C-initial* defines the names and associated values of the built-in primitive operators and functions described in Section A.3.4 below.

The definition of Eval by cases follows. Unless E is handled by one of these cases, Eval(E, C) yields a runtime error. As mentioned above, the domain of Eval includes the language generated by the concrete grammar as a proper subset. Thus, in some of the cases below, the expression E can arise only as an intermediate result of another case of Eval. These cases are explicitly noted.

The pseudo-code that defines the various cases of Eval and the primitive functions should be read like C++. That code assumes the declaration for the representation of Vesta values shown in Table A.2. Note that the `operator==` is the one invoked by uses of "`==`" in the C++ pseudo-code. It is not to be confused with the primitive equality operator defined on various Vesta types in Section A.3.4. The pseudo-code also refers to the constants shown in Table A.3.

For convenience, the pseudo-code adopts the following notational conventions:

- Eval is defined by cases rather than by one C++ function with an enormous embedded case selection.

- Recursive references to Eval appear inline in the same form that is used to identify the individual cases.

- Primitive functions of the Vesta language, whose names begin with an underscore, are invoked inline from the pseudo-code as if they were ordinary C++ functions. The primitive operators of the Vesta language are invoked in this way too; for example, when the pseudo-code refers to operator+, it means the Vesta primitive function, not the C++ operator. Note that some of the Vesta operators are overloaded by type, but not by arity. For example, operator+ is defined on integers, texts, lists, and bindings, but it always takes two arguments.

- In the pseudo-code for rules that contain the terminal Id, the variable `id` denotes the value of the Id represented as a t_text.

- If the pseudo-statement `error` is reached, evaluation halts with a runtime error and appropriate error message. No value is produced.

In each of the following sections, we first present the relevant portions of the language syntax. We then present the evaluation rules that apply to those syntactic constructs. The complete language syntax is given in Section A.4.

A.3.3.1 Expr

Syntax:

```
Expr    ::= if Expr then Expr else Expr | Expr1
Expr1   ::= Expr2 {  =>  Expr2 }*
Expr2   ::= Expr3 {  ||  Expr3 }*
Expr3   ::= Expr4 {  &&  Expr4 }*
Expr4   ::= Expr5 [ { == | != | < | > | <= | >= } Expr5 ]
Expr5   ::= Expr6 { AddOp Expr6 }*
AddOp   ::= +  |  ++  |  -
Expr6   ::= Expr7 { MulOp Expr7 }*
MulOp   ::= *
Expr7   ::= [ UnaryOp ] Expr8
UnaryOp ::= -  |  !
Expr8   ::= Primary [ TypeQual ]
Primary ::= ( Expr ) | Literal | Id | List
            | Binding | Select | Block | FuncCall
```

The grammar lists the operators in increasing order of precedence. The binary operators at each precedence level are left-associative.

Evaluation Rules:

The evaluation rules for conditional, implication, disjunction, conjunction, comparison, AddOp, MulOp, UnaryOp, and parenthesized expressions are shown in Tables A.4 and A.5. There are seven remaining possibilities for a Primary: Literal, Id, List, Binding, Select, Block, and FuncCall. These are treated separately in subsequent sections.

A.3.3.2 Literal

Syntax:

```
Literal    ::= ERR | TRUE | FALSE | Text | Integer
```

Evaluation Rules:

```
Eval( ERR    , C) = err
Eval( TRUE   , C) = true
Eval( FALSE  , C) = false
Eval( Text   , C) = the corresponding t_text value
Eval( Integer, C) = the corresponding t_int value
```

In the Text evaluation rule, the C++ interpretation of escape characters is used. In the Integer evaluation rule, evaluation halts with a runtime error if the integer is too large or small to be represented by the implementation.

A.3.3.3 Id

Evaluation Rules:

```
Eval( Id , C) = _lookup(C, id)
```

As defined in Section A.3.4.5, `_lookup`(*b*, *nm*) is the value associated with the non-empty name *nm* in the binding *b*. The evaluation halts with a runtime error if *nm* is empty or is not in *b*'s domain.

A.3.3.4 List

Syntax:

```
List       ::= < Expr*, >
```

```
// conditional expression
Eval( if Expr_1 then Expr_2 else Expr_3 , C) =
{
  val b = Eval( Expr_1 , C);
  if (_is_bool(b) == false) error;
  if (b == true) return Eval( Expr_2 , C);
  else return Eval( Expr_3 , C);
}

// conditional implication
Eval( Expr_1 => Expr_2 , C) =
{
  val b = Eval( Expr_1 , C);
  if (_is_bool(b) == false) error;
  if (b == false) return true;
  b = Eval( Expr_2 , C);
  if (_is_bool(b) == false) error;
  return b;
}

// conditional OR (disjunction)
Eval( Expr_1 || Expr_2 , C) =
{
  val b = Eval( Expr_1 , C);
  if (_is_bool(b) == false) error;
  if (b == true) return true;
  b = Eval( Expr_2 , C);
  if (_is_bool(b) == false) error;
  return b;
}

// conditional AND (conjunction)
Eval( Expr_1 && Expr_2 , C) =
{
  val b = Eval( Expr_1 , C);
  if (_is_bool(b) == false) error;
  if (b == false) return false;
  b = Eval( Expr_2 , C);
  if (_is_bool(b) == false) error;
  return b;
}
```

Table A.4: Evaluation rules for conditional, implication, disjunction, and conjunction expressions. As defined in Section A.3.4.7, `_is_bool`(*b*) is *true* if *b* is a value of type t_bool and *false* otherwise.

```
// comparison
Eval( Expr_1 == Expr_2 , C) =
  operator==(Eval( Expr_1 , C), Eval( Expr_2 , C))
Eval( Expr_1 != Expr_2 , C) =
  operator!=(Eval( Expr_1 , C), Eval( Expr_2 , C))
Eval( Expr_1 <  Expr_2 , C) =
  operator< (Eval( Expr_1 , C), Eval( Expr_2 , C))
Eval( Expr_1 >  Expr_2 , C) =
  operator> (Eval( Expr_1 , C), Eval( Expr_2 , C))
Eval( Expr_1 <= Expr_2 , C) =
  operator<=(Eval( Expr_1 , C), Eval( Expr_2 , C))
Eval( Expr_1 >= Expr_2 , C) =
  operator>=(Eval( Expr_1 , C), Eval( Expr_2 , C))

// AddOp
Eval( Expr_1 +  Expr_2 , C) =
  operator+ (Eval( Expr_1 , C), Eval( Expr_2 , C))
Eval( Expr_1 ++ Expr_2 , C) =
  operator++(Eval( Expr_1 , C), Eval( Expr_2 , C))

// MulOp
Eval( Expr_1 -  Expr_2 , C) =
  operator- (Eval( Expr_1 , C), Eval( Expr_2 , C))
Eval( Expr_1 *  Expr_2 , C) =
  operator* (Eval( Expr_1 , C), Eval( Expr_2 , C))

// UnaryOp
Eval( ! Expr , C) = operator!(Eval( Expr , C))
Eval( - Expr , C) = operator-(Eval( Expr , C))

// parenthesization
Eval( ( Expr ) , C) = Eval( Expr , C)
```

Table A.5: Evaluation rules for comparison, AddOp, MulOp, UnaryOp, and parenthesized expressions.

The use of <, > as both binary operators and list delimiters makes the grammar ambiguous. Section A.4.2 explains how the ambiguity is resolved.

Syntactic Desugarings:

<Expr$_1$, . . . , Expr$_n$>   desugars to   <Expr$_1$> + <Expr$_2$, . . . , Expr$_n$>

Here, '+' is the concatenation operator on lists.

Evaluation Rules:

```
Eval( <>       , C) = emptylist
Eval( < Expr > , C) = _list1(Eval( Expr , C))
```

As defined in Section A.3.4.4, `_list1`(*val*) evaluates to a list containing the single value *val*.

A.3.3.5 Binding

Syntax:

```
Binding    ::= '[' BindElem*, ']'
BindElem   ::= SelfNameB | NameBind
SelfNameB  ::= Id
NameBind   ::= GenPath = Expr
GenPath    ::= GenArc { Delim GenArc }* [ Delim ]
GenArc     ::= Arc | $ Id | $ ( Expr ) | % Expr %
Arc        ::= Id | Integer | Text
```

Syntactic Desugarings:

The following desugarings apply to BindElem's within a Binding.

Id	desugars to	Id = Id
GenArc Delim = Expr	desugars to	GenArc = Expr
GenArc Delim GenPath = Expr	desugars to	GenArc = [GenPath = Expr]
$ Id = Expr	desugars to	$ (Id) = Expr
% Expr$_1$ % = Expr$_2$	desugars to	$ (Expr$_1$) = Expr$_2$

The SelfNameB syntactic sugar allows names from the current scope to be copied into bindings more succinctly. For example, the binding value:

```
  [ progs = progs, tests = tests, lib = lib ]
```

can instead be written:

```
  [ progs, tests, lib ]
```

The GenPath syntactic sugar allows bindings consisting of a single path to be written more succinctly. For example, the binding value:

```
  [ env_ovs = [ Cxx = [ switches = [ compile =
    [ debug = "-g3", optimize = "-O" ]]]]]
```

can instead be written:

```
  [ env_ovs/Cxx/switches/compile =
    [ debug = "-g3", optimize = "-O" ]]
```

Evaluation Rules:

First, the rules for constructing empty and singleton bindings:

```
Eval( [ ]            , C) = emptybinding
Eval( [ Arc = Expr ] , C) = _bind1(id, Eval( Expr , C))
```

Here *id* is the t_text representation of *Arc*. The conversion from an Arc to a t_text is straightforward. If the Arc is an Id, the literal characters of the identifier become the text value. If the Arc is an Integer, the literal characters used to represent the integer in the source of the model become the text value. If the Arc is a Text, the result of Eval(Arc, C) is used. As defined in Section A.3.4.5, `_bind1`(*id*, *v*) evaluates to a singleton binding that associates the non-empty t_text *id* with the value *v*.

The `$(`*`Expr`*`)` syntax allows the name introduced into a binding to be computed:

```
Eval( [ $ ( Expr_1 ) = Expr_2 ] , C) =
  _bind1(Eval(Expr_1, C), Eval( Expr_2 , C))
```

When the field name is computed using the `$` syntax, the expression must evaluate to a non-empty t_text (see the `_bind1` primitive in Section A.3.4.5 below).

The following rule handles the case where multiple BindElem's are given.

```
Eval( [ BindElem_1, ..., BindElem_n ] , C) =
{
  val b1 = Eval( [ BindElem_1 ] , C);
  val b2 = Eval( [ BindElem_2, ..., BindElem_n ] , C);
  return _append(b1, b2);
}
```

As defined in Section A.3.4.5, `_append`(*b1*, *b2*) evaluates to the concatenation of the bindings *b1* and *b2*; it requires that their domains are disjoint.

A.3.3.6 Select

Syntax:

```
Select     ::= Primary Selector GenArc
Selector   ::= Delim | !
GenArc     ::= Arc | $ Id | $ ( Expr ) | % Expr %
Arc        ::= Id | Integer | Text
```

A Select expression denotes a selection from a binding, so the Primary must evaluate to a binding value.

Syntactic Desugarings:

Primary Selector `%` Expr `%`   desugars to   Primary Selector `$ (` Expr `)`

Evaluation Rules:

The Delim syntax selects a value out of a binding by name.

```
Eval( Primary Delim Arc , C) =
  _lookup(Eval( Primary , C), id)
```

Here *id* is the t_text value of *Arc*, as defined in Section A.3.3.5 above.

The `$(`*`Expr`*`)` syntax allows the selected name to be computed:

```
Eval( Primary Delim $ ( Expr ) , C) =
  _lookup(Eval( Primary , C), Eval( Expr , C))
```

The `!` syntax tests whether a name is in a binding's domain:

```
Eval( Primary ! Id , C) =
  _defined(Eval( Primary , C), id),
```

As defined in Section A.3.4.5, `_defined`(*b*, *nm*) evaluates to *true* if *nm* is non-empty and in *b*'s domain, and to *false* otherwise. As above, the `$(`*`Expr`*`)` syntax can be used to compute the name:

```
Eval( Primary ! $ ( Expr ) , C) =
  _defined(Eval( Primary , C), Eval( Expr , C))
```

In both cases where the GenArc is a computed expression, the Expr must evaluate to a t_text.

A.3.3.7 Block

Syntax:

```
Block      ::= ‘{’ Stmt*; Result; ‘}’
Stmt       ::= Assign | Iterate | FuncDef | TypeDef
Result     ::= { value | return } Expr
```

Syntactic Desugarings:

`return` Expr desugars to `value` Expr

That is, the keywords `return` and `value` are synonyms, provided for stylistic reasons. The `return`/`value` statement must appear at the end of a Block; there is no analog of the C/C++ return statement that terminates execution of the function in which it appears.

Evaluation Rules:

Since the Vesta SDL is functional, evaluation of a statement does not produce side-effects, but rather produces a binding. Evaluation of a block occurs by augmenting the context with the bindings produced by evaluating the Stmts, then evaluating the final Expr in the augmented context.

```
Eval( { value Expr } , C) = Eval( Expr , C)

Eval( { Stmt_1; ...; Stmt_n; value Expr } , C) =
{
  val b = Eval( { Stmt_1; ...; Stmt_n } , C);
  return Eval( Expr , operator+(C, b));
}
```

Notice that this second rule introduces an argument to Eval in the “extended” language that is not generated by any non-terminal of the grammar.

A.3.3.8 Stmt

Evaluation Rules:

Evaluating a Stmt or sequence of Stmts produces a binding. Note that the binding resulting from the evaluation of a sequence of Stmts is simply the overlay (operator ‘+’) of the bindings resulting from evaluating each Stmt in the sequence, and does not include the context *C*.

```
Eval( { } , C) = emptybinding

Eval( { Stmt_1; Stmt_2 ...; Stmt_n } , C) =
{
  val b0 = Eval( Stmt_1 , C);
  val b1 = Eval( { Stmt_2; ...; Stmt_n }, operator+(C, b0));
  return operator+(b0, b1);
}
```

These rules apply to constructs in the "extended" language. There are three possibilities for a Stmt: Assign, Iterate, and FuncDef. They are covered in the next three sections.

A.3.3.9 Assign

Since the Vesta SDL is functional, assignments do not produce side-effects. Instead, they introduce a new name into the evaluation context whose value is that of the given expression.

Syntax:

```
Assign     ::= Id [ TypeQual ] [ Op ] = Expr
Op         ::= AddOp | MulOp
AddOp      ::= +  |  ++  |  -
MulOp      ::= *
```

Syntactic Desugarings:

Id Op = Expr   desugars to   Id = Id Op Expr

Evaluation Rules:

```
Eval( Id = Expr , C) = _bind1(id, Eval( Expr , C))
```

A.3.3.10 Iterate

The language includes expressions for iterating over both lists and bindings. There is also a `_map` primitive defined on lists (Section A.3.4.4) and bindings (Section A.3.4.5). `_map` is more efficient but less general than the language's Iterate construct.

Syntax:

```
Iterate    ::= foreach Control in Expr do IterBody
Control    ::= Id | ‘[’ Id = Id ‘]’
IterBody   ::= Stmt | ‘{’ Stmt+; ‘}’
```

The two Control forms are used to iterate over lists and bindings, respectively.

Evaluation Rules:

```
// iteration with single-statement body
Eval( foreach Control in Expr do Stmt , C) =
  Eval( foreach Control in Expr do { Stmt } , C)
```

The semantics of a loop are to conceptually unroll the loop *n* times, where *n* is the length of the list or binding being iterated over. The evaluation rules for iterating over lists and bindings are shown in Table A.6. Note that the iteration variables (that is, `Id`, `Id1`, and `Id2` in the Table) are not bound in the binding that results from evaluating the `foreach` statement. However, any assignments made in the loop body *are* included in the result binding.

Iteration statements are typically used to walk over or collect parts of a list or binding. For example, Table A.7 presents functions for reversing a list and for counting the number of leaves in a binding.

A.3.3.11 FuncDef

Syntax:

The function definition syntax allows a suffix of the formal parameters to have associated default values.

```
FuncDef     ::= Id Formals+ [ TypeQual ] Block
Formals     ::= ( FormalArgs )
FormalArgs ::= TypedId*,
              | { TypedId = Expr }*,
              | TypedId { , TypedId }* { , TypedId = Expr }+
```

The three alternatives for FormalArgs correspond to the cases in which no arguments are defaulted, all arguments are defaulted, and some arguments are defaulted, respectively.

Note that the syntax allows multiple `Formals` to follow the function name. As the rules below describe, the use of multiple Formals produces a sequence of curried functions, all but the first of which is anonymous.

```
// iteration over a list
Eval( foreach Id in Expr do { Stmt_1; ...; Stmt_n } , C) =
{
  val l = Eval( Expr, C);
  if (_is_list(l) == false) error;
  t_text id = Id; // identifier Id as a t_text
  val r = emptybinding;
  for (; !(l == emptylist); l = _tail(l)) {
    val r1 = operator+(C, r);
    r1 = operator+(r1, _bind1(id, _head(l)));
    r = operator+(r, Eval( { Stmt_1; ...; Stmt_n } , r1));
  }
  return r;
}

// iteration over a binding
Eval(foreach [Id1=Id2] in Expr do {Stmt_1;...;Stmt_n}, C) =
{
  val b = Eval( Expr, C);
  if (_is_binding(b) == false) error;
  t_text id1 = Id1; // identifier Id1 as a t_text
  t_text id2 = Id2; // identifier Id2 as a t_text
  val r = emptybinding;
  for (; !(b == emptybinding); b = _tail(b)) {
    val r1 = operator+(C, r);
    r1 = operator+(r1, _bind1(id1, _n(_head(b))));
    r1 = operator+(r1, _bind1(id2, _v(_head(b))));
    r = operator+(r, Eval( { Stmt_1; ...; Stmt_n } , r1));
  }
  return r;
}
```

Table A.6: Evaluation rules for iterating over lists and bindings. As defined in Section A.3.4.7, `_is_list`(*l*) is *true* if *l* is of type t_list, and *false* otherwise; `_is_binding`(*b*) is *true* if *b* is of type t_binding, and *false* otherwise.

```
reverse_list(l: list): list
{
  res: list = <>;
  foreach elt in l do
    res = <elt> + res;
  return res;
}

count_leaves(b: binding): int
{
  res: int = 0;
  foreach [ nm = val ] in b do
    res += if _is_binding(val)
      then count_leaves(val) else 1;
  return res;
}
```

Table A.7: Two functions demonstrating the use of `foreach` to iterate over a list and a binding.

Evaluation Rules:

```
Eval( Id Formals_1 ... Formals_n Block , C) =
  _bind1(id, Eval( e , C1)),
```

where:

- `e = LAMBDA Formals_1 ... LAMBDA Formals_n Block`
- `C1 = operator+(C, _bind1(id, Eval( e , C1)))`

Notice the recursive definition of C1. This allows functions to be self-recursive, but not mutually recursive. Although this recursive definition looks a little odd, it can be implemented by the evaluator by introducing a cycle into the context C1. This is the only case where any Vesta value can contain a cycle (the language syntax and operators do not allow cyclic lists or bindings to be constructed), and the cycle is invisible to clients. There is no practical difficulty in constructing the cycle because, as we are about to see, the "evaluation" of a `LAMBDA` is purely syntactic.

Also note that this rule produces a `LAMBDA` construct in the "extended" language that is not generated by any non-terminal of the grammar. The evaluation of `LAMBDA` produces a t_closure value $\langle e, f, b \rangle$ as described in Section A.3.1.

The following is the simple case of `LAMBDA`, where all actual parameters must be given in any application of the closure. The reason for the restriction on the use of "." as a formal parameter is treated below in the section on function calls.

```
Eval( LAMBDA (Id_1, ..., Id_m)
      LAMBDA Formals_2 ... LAMBDA Formals_n Block , C) =
  <LAMBDA Formals_2 ... LAMBDA Formals_n Block, f, C>,
```

where *f* is a list of pairs $(id_i, \langle\text{emptyExpr}\rangle)$ such that id_i is the t_text representation of Id_i, for i in the closed interval $[1, m]$. If any of the identifiers Id_i is ".", the evaluation halts with a runtime error.

In the typical case where only one set of Formals is specified (that is, $n = 1$), the first element of the resulting closure value is simply a Block.

Next is the general case of `LAMBDA`, in which "default expressions" are given for a suffix of the formal parameter list. Functions may be called with fewer actuals than formals if each formal corresponding to an omitted actual includes an expression specifying the default value to be computed. When the closure is applied, if an actual parameter is missing, its formal's expression is evaluated (in the context of the `LAMBDA`) and passed instead. The following section on FuncCall defines this precisely.

```
Eval(
    LAMBDA (Id_1,... Id_k, Id_k+1=Expr_k+1,... Id_m=Expr_m)
    LAMBDA Formals_2 ... LAMBDA Formals_n Block , C) =
  <LAMBDA Formals_2 ... LAMBDA Formals_n Block, f, C>,
```

where *f* is a list of pairs $(id_i, expr_i)$ such that:

- id_i is the t_text representation of Id_i, for i in $[1, m]$,
- $expr_i$ is $\langle\text{emptyExpr}\rangle$, for i in $[1, k]$, and
- $expr_i$ is Expr_i, for i in $[k + 1, m]$.

As before, if any of the identifiers Id_i is ".", the evaluation halts with a runtime error.

A.3.3.12 FuncCall

Syntax:

```
FuncCall   ::= Primary Actuals
Actuals    ::= ( Expr*, )
```

Evaluation Rules:

The function call mechanism provides special treatment for the identifier consisting of a single period, called the *current environment* and pronounced dot. Dot is typically assigned a binding that contains the tools, switches, and file system required for the rest of the build. The initial environment, *C-initial* (see Section A.3.3 above), does not bind dot (that is, "`_defined(`*`C-initial`*`, ".") == false`").

When a function is called, the context in which its body executes may bind "." to a value established as follows:

- if the function is defined with *n* formals and called with *n* or fewer actuals, then the value for "." at the point of call is bound to the implicit formal parameter named "." in the callee;
- if the function is defined with *n* formals and called with $n + 1$ actuals, then the value bound to the implicit formal parameter named "." is the value of the last actual.

Thus, the binding for ".", if any, is passed through the dynamic call chain until it is altered either explicitly by an Assign statement (Section A.3.3.9) or implicitly by calling a function with an extra actual parameter. The pseudo-code shown in Table A.8 makes this precise. In this code, the comparison with ⟨emptyExpr⟩ has not been formalized, but it should be intuitively clear.

A.3.3.13 Model

Syntax:

```
Model      ::= Files Imports Block
```

Evaluation Rules:

The nonterminal Model is treated like the body of a function definition (i.e., like a FuncDef (Section A.3.3.11), but without the identifier naming the function and with an empty list of formal parameters). More precisely:

```
Eval( Files Imports Block , C) =
{
  val C0 = Eval( Files Imports , emptybinding);
  return Eval( LAMBDA () Block , _append(C0, C));
}
```

```
Eval( Primary ( Expr_1, ..., Expr_n ) , C) =
{
  val cl = Eval( Primary , C);
  if (_is_closure(cl) == false) error;

  /* cl.e is the function body, cl.f are the formals, and
     cl.b is the context */
  int m = _length(cl.f);       // number of formals
  if (n > m + 1) error;        // too many actuals
  val C1 = cl.b;               // t_binding
  val f = cl.f;                // t_list (of (t, e) pairs)

  /* augment C1 to include formals bound to corresponding
     actuals */
  int i;
  for (i = 1; i <= m; i++) {
    val form = _head(f);       // i-th formal
    val act;                   // corresponding actual
    if (i <= n)
      act = Eval( Expr_i , C); // value for i-th actual
    else {
      if (form.e == <emptyExpr>) {
        // a required actual is missing
        error;
      }
      act = Eval( form.e , cl.b); // defaulted formal value
    }
    C1 = operator+(C1, _bind1(form.t, act));
    f = _tail(f);
  }

  // bind "." in C1
  val dot;
  if (n <= m)
    dot = _lookup(C, ".");     // inherit "." from C
  else
    dot = Eval( Expr_n , C);   // last actual value supplied
  C1 = operator+(C1, _bind1(".", dot));

  /* C1 is now a suitable environment.  If the closure is
     a primitive function, then invoke it by a special
     mechanism internal to the evaluator and return the
     value it computes. Otherwise, perform the following:
  */
  return Eval( cl.e , C1);
}
```

Table A.8: Evaluation rule for FuncCall.

As this rule indicates, the Files and Imports constructs are evaluated in an empty context, and they add to the closure context in which the model's LAMBDA is evaluated. In practice, the context C will always be the initial context *C-initial* when this rule is applied (cf. Sections A.3.3 and A.3.3.15).

The Files nonterminal introduces values corresponding to the contents of ordinary files and directories. The Imports nonterminal introduces closure values corresponding to other Vesta SDL models.

The evaluation rules handle Files and Imports clauses by augmenting the context using the _append primitive, thereby ensuring that the names introduced by these clauses are all distinct, just as if the Files and Imports clauses of the Model were a single binding constructor. The Files and Imports clauses are evaluated independently:

```
Eval( Files Imports , C) =
  _append(Eval( Files , C), Eval( Imports , C))
```

The following two sections give the rules for evaluating Files and Imports clauses individually. It is worth noting that the evaluation context C is ignored in those rules.

A.3.3.14 Files

A Files clause introduces names corresponding to files or directories in the Vesta repository. Generally, these files or directories are named by relative paths, which are interpreted relative to the location of the model containing the Files clause. Absolute paths are permitted, though they are expected to be rarely used.

Syntax:

```
Files       ::= FileClause*
FileClause  ::= files FileItem*;
FileItem    ::= FileSpec | FileBinding
FileSpec    ::= [ Arc = ] DelimPath
FileBinding ::= Arc = '[' FileSpec*, ']'

DelimPath   ::= [ Delim ] Path [ Delim ]
Path        ::= Arc { Delim Arc }*
Arc         ::= Id | Integer | Text
```

Each FileItem in a Files clause takes one of two forms: a FileSpec or a FileBinding. Each form introduces (binds) exactly one name. In the FileSpec case, the name corresponds to the contents of a single file or directory; in the FileBinding case, the name corresponds to a binding consisting of perhaps many files or directories. In both cases, the identifier introduced into the Vesta naming context or the identifiers introduced into the binding can be specified explicitly or derived from an Arc in the Path.

For example, consider the following `files` clause:

```
files
  scripts = bin;
  c_files = [ utils.c, main.c ];
```

Suppose the directory containing this model also contains a directory named `bin` and files named `utils.c` and `main.c`. Then this `files` clause introduces the two names `scripts` and `c_files` into the context. The former is bound to a binding whose structure corresponds to the `bin` directory. The latter is bound to a binding that maps the names `utils.c` and `main.c` to the contents of those files, respectively. The file contents are values of type t_text.

Syntactic Desugaring:

When multiple FileItem's are given in a FileClause, the `files` keyword simply distributes over each of the FileItem's. That is:

```
files FileItem_1; ...; FileItem_n;
```

desugars to:

```
files FileItem_1;
...;
files FileItem_n;
```

When the initial Arc is omitted from a FileSpec, it is inferred from the path. In particular:

```
files [ Delim ] { Arc Delim }* Arc [ Delim ];
```

desugars to:

```
files Arc = [ Delim ] { Arc Delim }* Arc [ Delim ];
```

Evaluation Rules:

Multiple FileClauses are evaluated independently:

```
Eval( FileClause_0 FileClause_1 ... FileClause_n , C) =
{
  val C2 = Eval( FileClause_1 ... FileClause_n , C);
  return _append(Eval( FileClause_0 , C), C2);
}
```

That leaves only two cases to consider: FileSpec (in which the initial Arc is specified) and FileBinding.

```
// FileSpec
Eval( files Arc = DelimPath , C) = _bind1(id, v)
```

where:

- *id* is the t_text representation of *Arc*, as defined in Section A.3.3.5 above.
- If *DelimPath* begins with a Delim, it is interpreted as an absolute path, which must nevertheless resolve to a file or directory in the Vesta repository. If *DelimPath* does not begin with a Delim, it refers to a file or directory named relative to the directory of the enclosing Model.
- If the entity named by *DelimPath* is a file, *v* is a t_text value formed by taking the file's contents. If *DelimPath* names a directory, *v* is a t_binding value constructed from the contents of the directory, treating the files (if any) in the directory as above (i.e., as t_text values) and the directories (if any) recursively (i.e., as bindings). The members of the resulting binding are in an unspecified order. If *DelimPath* does not correspond to either an extant file or a directory, the evaluation halts with a runtime error.

```
// FileBinding
Eval( files Arc = [ FileSpec_1, ..., FileSpec_n ] , C) =
  _bind1(id, Eval( files FileSpec_1; ...; FileSpec_n , C))
```

Again, *id* is the t_text representation of *Arc*.

The FileBinding form of the Files clause provides a convenient way to create a binding containing multiple FileSpecs. Without this construct, it would be necessary to name each file twice, once in the FileSpec and once in a subsequent binding constructor. Making a binding with FileBinding is semantically similar to constructing a file system directory, with the additional property that there is an enumeration order for the component files.

Notice that the grammar and evaluation rules given above for *FileSpec* and *FileBinding* allow a general Arc on the left-hand side of each equal sign, not just an Id. This was done to simplify the definitions and desugaring rules. However, it would be useless to write constructs like the following, which introduce names that cannot be referenced in the body of the model:

```
files
    33;
    34 = 34;
    "hash-table.c";
    "foo bar" = [ foo, bar ];
```

Therefore, we introduce an additional restriction: the context created by a Files clause must bind only names that are legal identifiers; that is, names that match the syntax of the Id token.

If you need to use files whose names are not legal identifiers, you should either assign them legal names with the equal sign syntax or embed them in a binding. Some possibilities:

```
// Choose a legal name
files
    f33 = 33;
    f34 = 34;
    hash_table.c = "hash-table.c";
    foo_bar = [ foo, bar ];

// Embed in a binding
files
    f = [ 33, 34 ];
    src = [ "hash-table.c" ];
```

A.3.3.15 Imports

The Imports clause enables one Vesta SDL model to reference and use others; that is, it supports modular decomposition of Vesta SDL programs.

Syntax:

```
Imports    ::= ImpClause*
ImpClause  ::= ImpIdReq | ImpIdOpt
```

There are two major forms of the Imports clause: one where identifiers are required (ImpIdReq), and one where they are optional (ImpIdOpt). Both forms have two sub-forms in which either a single model or a list of models may be imported.

First, consider the ImpIdReq case. This form is typically used to import models in the same package as the importing model. Each ImpItemR in the ImpIdReq clause takes one of two forms: an ImpSpecR or an ImpListR. Each form binds exactly one name.

```
ImpIdReq   ::= import ImpItemR*;
ImpItemR   ::= ImpSpecR | ImpListR
ImpSpecR   ::= Arc = DelimPath
ImpListR   ::= Arc = ‘[’ ImpSpecR*, ‘]’

DelimPath  ::= [ Delim ] Path [ Delim ]
Path       ::= Arc { Delim Arc }*
Arc        ::= Id | Integer | Text
```

In the ImpSpecR case, the name is bound to the t_closure value that results from evaluation of the contents of a file according to the Model evaluation rules of Section A.3.3.13. For example, consider the Import clause:

```
  import self = progs.ves;
```

This clause binds the name `self` to the closure corresponding to the local *progs.ves* model in the same directory as the model in which it appears.

In the ImpListR case, the name is bound to a binding of such values. For example:

```
  import sub =
    [ progs = src/progs.ves, tests = src/tests.ves ];
```

This clause binds the name `sub` to a binding containing the names `progs` and `tests`; these names within the binding are bound to the closures corresponding to the models named *progs.ves* and *tests.ves* in the package’s `src` subdirectory. For example, the *progs.ves* model could be invoked by writing “`sub/progs()`”.

Because the Imports clause often mentions several files with names that share a common prefix, a syntactic form is provided to allow the prefix to be written once. This is the ImpIdOpt form. It is used to import models from other packages. The semantics are defined so that many identifiers are optional; when omitted, they default to the name of the package from which the model is being imported. As in the ImpIdReq case, ImpIdOpt has forms for importing both single models and lists of multiple models.

```
ImpIdOpt   ::= from DelimPath import ImpItemO*;
ImpItemO   ::= ImpSpecO | ImpListO
ImpSpecO   ::= [ Arc = ] Path [ Delim ]
ImpListO   ::= Arc = ‘[’ ImpSpecO*, ‘]’
```

Here are some examples of ImpIdOpt imports:

```
  from /vesta/west.vestasys.org/vesta import
    cache/12/build.ves;
    libs = [ srpc/2/build.ves, basics/5/build.ves ];
```

This example binds the name `cache` to the closure corresponding to version 12 of that package’s *build.ves* model, and it binds the name `libs` to a binding containing the names `srpc` and `basics`, bound to versions 2 and 5 of those package’s *build.ves* models. (As the evaluation rules below describe, the three occurrences of “`/build.ves`” in this example could actually have been omitted.)

Syntactic Desugaring:

When multiple ImpItemR’s are given in a ImpIdReq, the `import` keyword distributes over each of the ImpItemR’s. That is:

```
  import ImpSpec_1; ...; ImpSpec_n;
```

desugars to:

```
  import ImpSpec_1;
  ...;
  import ImpSpec_n;
```

Similarly, the `from` clause distributes over the individual imports of an ImpIdOpt. In particular:

```
  from DelimPath import ImpItemO_1; ...; ImpItemO_n;
```

desugars to:

```
  from DelimPath import ImpItemO_1;
  ...;
  from DelimPath import ImpItemO_n;
```

The use of `from` makes it optional to supply a name for the closure value being introduced; if the name is omitted, it is derived from the Path following the `import` keyword as follows:

```
  from DelimPath import
    [ Arc_1 = ] [ Delim ] Arc_2 { Delim Arc }* [ Delim ]
```

desugars to:

```
  import Arc =
    DelimPath Delim Arc_2 { Delim Arc }* [ Delim ]
```

where *Arc* is Arc_1 if it is present and is Arc_2 otherwise. Similarly:

```
  from DelimPath import Arc = [
    [ Arc1_1 = ] [ Delim ] Arc2_1 { Delim Arc }* [ Delim ],
    ...,
    [ Arc1_n = ] [ Delim ] Arc2_n { Delim Arc }* [ Delim ] ]
```

desugars to:

```
  import Arc = [
    Arc_1 = DelimPath Delim Arc2_1 {Delim Arc }* [ Delim ],
    ...,
    Arc_n = DelimPath Delim Arc2_n {Delim Arc }* [ Delim ] ]
```

where Arc_i is Arc1_i if it is present and is Arc2_i otherwise.

Evaluation Rules:

Multiple ImpClause's are evaluated independently:

```
Eval( ImpClause_0 ImpClause_1 ... ImpClause_n , C) =
{
  val C2 = Eval( ImpClause_1 ... ImpClause_n , C);
  return _append(Eval( ImpClause_0 , C), C2);
}
```

This leaves two fundamental forms of the Imports clause, whose semantics are defined as follows:

```
// ImpSpecR
Eval( import Arc = DelimPath , C) =
  _bind1(id, Eval( model , C-initial))
```

where:

- *id* is the t_text representation of *Arc*, as defined in Section A.3.3.5 above.

- Let *f* be the sequence of Delims and Arcs that constitute the DelimPath.

  1. If *f* does not begin with a Delim, prepend "`Delim Path0 Delim`" to *f*, where *Path0* names the directory containing the Model in which this Imports clause appears.
  2. Look up the path *f* in the Vesta repository. (See Section A.3.3.16 below.) If *f* names a directory, append a Delim (if *f* doesn't already end in one) and the string "`build.ves`", then look up the augmented path *f* in the repository again. If *f* does not name a directory and its final element does not end in "`.ves`", append the string "`.ves`" to the final element of *f*, and look it up in the repository again.

- *model* is the Vesta SDL Model represented by the contents of the file in the Vesta repository named by the sequence *f*. If no such expression can be produced (e.g., the file doesn't exist, or can't be parsed as an expression), evaluation halts with a runtime error.

```
// ImpListR
Eval( import Arc = [ ImpSpecR_1, ..., ImpSpecR_n ] , C) =
  _bind1(id, Eval( import ImpSpecR_1; ...; ImpSpecR_n , C))
```

Again, *id* is the t_text representation of *Arc*.

As with the Files clause, and for the same reason, we add one restriction to the rules just given: the context created by an Imports clause must bind only names that are legal identifiers; that is, names that match the syntax of the Id token.

A.3.3.16 Filename Interpretation

The evaluation rules for the Files and Imports clauses do not specify how the sequence of Arcs and Delims making up a DelimPath is converted into a filename in the underlying file system. While this is somewhat system-dependent, it is nevertheless intended to be intuitive. In particular,

- Multiple adjacent Delims are replaced by a single one. (The grammar above doesn't permit adjacent Delims, but they can be produced by the desugaring rules.)

- The Vesta SDL syntax allows the arbitrary intermingling of "`/`" and "`\`" as arc separators. However, the implementation actually requires that Vesta programs use one or the other uniformly. When creating a filename from a sequence of Arcs and Delims, the implementation inserts the appropriate arc separator required by the underlying file system. The choice is not influenced by the choice of Delim that appears in the Vesta SDL program.

- The grammar permits an Arc to be an arbitrary Text. An Arc in a filename, however, is forbidden to contain a Delim character (i.e., forward or backward slash), and the Arcs "`..`" and "`.`" are forbidden in filenames as well. In particular, "`..`" cannot be used to mean *parent directory* and "`.`" cannot be used to mean *current directory*. The "`..`" notation is forbidden for technical reasons related to Vesta caching, while the "`.`" notation is simply unimplemented. However, the empty Arc "" can be used to denote the current directory.

A.3.4 Primitives

The primitive names and associated values described below are provided by the Vesta SDL interpreter in *C-initial*, the initial context. Most of these values are closures with empty contexts; that is, they are primitive functions.

In the descriptions that follow, the notation used for the function signatures follows C++, with the result type preceding the function name and each argument type preceding the corresponding argument name. Defaulting conventions also follow C++; if an argument name is followed by "`= `*`value`*", then omitting the corresponding actual argument is equivalent to supplying *value*.

Some of the function signatures use the C++ operator definition syntax, which should be understood as defining a function whose name is not an Id in the sense of the grammar above. Such operator names cannot be rebound. These operators are typically overloaded, as the descriptions below indicate. Uses of these built-in Vesta primitives within C++ code are denoted by the `operator` syntax.

The pseudo-code of this section assumes the definition of the Vesta value class given at the start of Section A.3.3. Invocation of a Vesta operator primitive within the pseudo-code is denoted by the `operator` syntax. All other operators appearing in the pseudo-code denote the C++ operators.

In these descriptions, the argument types represent the natural domain; the result type is the natural range. If a primitive function is passed a value that lies outside its natural domain, evaluation halts with a runtime error. This type-checking occurs when the primitive function is called, not before.

A.3.4.1 Functions on Type t_bool

Recall that *true* and *false* are Vesta values, not C++ quantities.

```
t_bool
operator==(t_bool b1, t_bool b2)
```

Returns *true* if *b1* and *b2* are the same, and *false* otherwise.

```
t_bool
operator!=(t_bool b1, t_bool b2)
  operator!(operator==(b1, b2))
```

```
t_bool
operator!(t_bool b)
```

Returns the logical complement of *b*.

A.3.4.2 Functions on Type t_int

```
t_bool
operator==(t_int i1, t_int i2)
```

Returns *true* if *i1* and *i2* are equal, and *false* otherwise.

```
t_bool
operator!=(t_int i1, t_int i2) =
  operator!(operator==(i1, i2))
```

```
t_int
operator+(t_int i1, t_int i2)
```

Returns the integer sum *i1* + *i2* unless it lies outside the implementation-defined range, in which case evaluation halts with a runtime error.

```
t_int
operator-(t_int i1, t_int i2)
```

Returns the integer difference *i1* - *i2* unless it lies outside the implementation-defined range, in which case evaluation halts with a runtime error.

```
t_int
operator-(t_int i) =
  operator-(0, i)
```

```
t_int
operator*(t_int i1, t_int i2)
```

Returns the integer product *i1* * *i2* unless it lies outside the implementation-defined range, in which case evaluation halts with a runtime error.

```
t_int
_div(t_int i1, t_int i2)
```

Returns the integer quotient *i1* / *i2* (that is, the floor of the real quotient) unless it lies outside the implementation-defined range, in which case evaluation halts with a runtime error. This error is possible only if *i2* is zero or if *i2* is -1 and *i1* is the largest implementation-defined negative number.

```
t_int
_mod(t_int i1, t_int i2) =
  operator-(i1, operator*(_div(i1,i2), i2))
```

```
t_bool
operator<(t_int i1, t_int i2)
```

Returns *true* if and only if *i1* is less than *i2*.

```
t_bool
operator>(t_int i1, t_int i2) =
  operator<(i2, i1)
```

```
t_bool
operator<=(t_int i1, t_int i2)
```

Returns *true* if and only if *i1* is at most *i2*.

```
t_bool
operator>=(t_int i1, t_int i2) =
  operator<=(i2, i1)
```

```
t_int
_min(t_int i1, t_int i2) =
{ if (operator<(i1, i2)) return i1; else return i2; }
```

```
t_int
_max(t_int i1, t_int i2) =
{ if (operator>(i1, i2)) return i1; else return i2; }
```

A.3.4.3 Functions on Type t_text

The first byte of a t_text value has index 0.

```
t_bool
operator==(t_text t1, t_text t2)
```

Returns *true* if *t1* and *t2* are identical byte sequences, and *false* otherwise.

```
t_bool
operator!=(t_text t1, t_text t2) =
  operator!(operator==(t1, t2))
```

```
t_text
operator+(t_text t1, t_text t2)
```

Returns the byte sequence formed by appending the byte sequence *t2* to the byte sequence *t1* (concatenation).

```
t_int
_length(t_text t)
```

Returns the number of bytes in the byte sequence *t*.

```
t_text
_elem(t_text t, t_int i)
```

If $0 \le i <$ `_length(`*t*`)`, returns a byte sequence of length 1 consisting of byte *i* of the byte sequence *t*. Otherwise, returns the empty byte sequence.

```
t_text
_sub(t_text t, t_int start = 0, t_int len = _length(t)) =
{
  int w = _length(t);
  int i = _min(_max(start, 0)), w);
  int j = _min(i + _max(len, 0), w);
  // 0 <= i <= j <= _length(t); extract [i..j)
  t_text r = "";
  for (; i < j; i++) r = operator+(r, _elem(t, i));
  return r;
}
```

Extracts from *t* and returns a byte sequence of length *len* beginning at byte *start*. Note the boundary cases defined by the pseudo-code; `_sub` produces a runtime error only if it is passed arguments of the wrong type.

```
t_int
_find(t_text t, t_text p, t_int start = 0) =
{
  int j = _length(t) - _length(p);
  if (j < 0) return -1;
  int i = _max(start, 0);
  if (i > j) return -1;
  for (; i <= j; i++) {
    int k = 0;
    while (k < _length(p) &&
           _elem(t, i+k) == _elem(p, k)) k++;
    if (k == _length(p)) return i;
  }
  return -1;
}
```

Finds the leftmost occurrence of *p* in *t* that begins at or after position *start*. Returns the index of the first byte of the occurrence, or -1 if none exists.

```
t_int
_findr(t_text t, t_text p, t_int start = 0) =
{
  int j = _length(t) - _length(p);
  if (j < 0) return -1;
  int i = _max(start, 0);
  if (i > j) return -1;
  for (; i <= j; j--) {
    int k = 0;
    while (k < _length(p) &&
           _elem(t, j+k) == _elem(p, k)) k++;
    if (k == _length(p)) return j;
  }
  return -1;
}
```

Finds the rightmost occurrence of *p* in *t* that begins at or after position *start*. Returns the index of the first byte of the occurrence, or -1 if none exists.

A.3.4.4 Functions on Type t_list

```
t_bool
operator==(t_list l1, t_list l2)
```

Returns *true* if *l1* and *l2* are lists of the same length containing (recursively) equal values, and *false* otherwise.

```
t_bool
operator!=(t_list l1, t_list l2) =
  operator!(operator==(l1, l2))
```

```
t_list
_list1(t_value v)
```

Returns a list containing a single element whose value is *v*.

```
t_value
_head(t_list l)
```

Returns the first element of *l*. If *l* is empty, evaluation halts with a runtime error.

```
t_list
_tail(t_list l)
```

Returns the list consisting of all elements of *l*, in order, except the first. If *l* is empty, evaluation halts with a runtime error.

```
t_int
_length(t_list l)
```

Returns the number of (top-level) values in the list *l*.

```
t_value
_elem(t_list l, t_int i)
```

Returns the *i*-th value in the list *l*. If no such value exists, evaluation halts with a runtime error. The first value of a list has index 0.

```
t_list
operator+(t_list l1, t_list l2)
```

Returns the list formed by appending *l2* to *l1*.

```
t_list
_sub(t_list l, t_int start = 0, t_int len = _length(l))
{
  int w = _length(l);
  int i = _min(_max(start, 0)), w);
  int j = _min(i + _max(len, 0), w);
  // 0 <= i <= j <= _length(l); extract [i..j)
  t_list r = emptylist;
  for (; i < j; i++) r = operator+(r, _elem(l, i));
  return r;
}
```

Returns the sub-list of *l* of length *len* starting at element *start*. Note the boundary cases defined by the pseudo-code; `_sub` produces a runtime error only if it is passed arguments of the wrong type.

```
t_list
_map(t_closure f, t_list l) =
{
  t_list res = emptylist;
  for (; !(l == emptylist); l = _tail(l)) {
    t_value v = f(_head(l)); // apply the closure "f"
    res = operator+(res, v);
  }
  return res;
}
```

Returns the list that results from applying the closure *f* to each element of the list *l*, and concatenating the results in order. The closure *f* should take one value (of type t_value) as argument and return a value of any type. If *f* has the wrong signature, the evaluation halts with a runtime error.

```
t_list
_par_map(t_closure f, t_list l)
```

Formally equivalent to `_map`, but the implementation may perform each application of *f* in a separate parallel thread. External tools invoked by `_run_tool` in different threads may be run simultaneously on different machines. If a runtime error occurs in one thread, the other threads may still run to completion before the evaluation terminates.

A.3.4.5 Functions on Type t_binding

```
t_bool
operator==(t_binding b1, t_binding b2)
```

Returns *true* if *b1* and *b2* are bindings of the same length containing the same names (in order) bound to (recursively) equal values, and *false* otherwise.

```
t_bool
operator!=(t_binding b1, t_binding b2) =
  operator!(operator==(b1, b2))
```

```
t_binding
_bind1(t_text n, t_value v)
```

If *n* is not empty, returns a binding with the single (name, value) pair (*n*, *v*). If *n* is empty, the evaluation halts with a runtime error.

```
t_binding
_head(t_binding b)
```

Returns a binding with one (name, value) pair equal to the first element of *b*. If *b* is empty, the evaluation halts with a runtime error.

```
t_binding
_tail(t_binding b)
```

Returns the binding consisting of all elements of *b*, in order, except the first. If *b* is empty, the evaluation halts with a runtime error.

```
t_int
_length(t_binding b)
```

Returns the number of (name, value) pairs in *b*.

```
t_binding
_elem(t_binding b, t_int i)
```

Returns a binding consisting solely of the *i*-th (name, value) pair in the binding *b*. If no such pair exists, the evaluation halts with a runtime error. The first pair of a binding has index 0.

```
t_text
_n(t_binding b)
```

If `_length(b) = 1`, returns the name part of the (name, value) pair that constitutes *b*. Otherwise, the evaluation halts with a runtime error.

```
t_value
_v(t_binding b)
```

If _length(*b*) differs from 1, the evaluation halts with a runtime error. Otherwise, let *v* be the value part of the (name, value) pair that constitutes *b*. This function returns *v*.

```
t_bool
_defined(t_binding b, t_text name)
```

If *name* is empty, the evaluation halts with a runtime error. Otherwise, the function returns *true* if the binding *b* contains a pair (*n, v*) with *n* identical to *name*, and *false* otherwise.

```
t_value
_lookup(t_binding b, t_text name)
```

If *name* is nonempty and is defined in *b*, returns the value associated with it; otherwise, the evaluation halts with a runtime error.

```
t_binding
_append(t_binding b1, t_binding b2)
```

Returns a binding formed by appending *b2* to *b1*, but only if all the names in *b1* and *b2* are distinct. Otherwise, the evaluation halts with a runtime error.

```
t_binding
operator+(t_binding b1, t_binding b2) =
{
  val r = emptybinding;
  for (; !(b1 == emptybinding); b1 = _tail(b1)) {
    val n = _n(_head(b1));
    val v;
    if (_defined(b2, n) == true)
      v = _lookup(b2, n);
    else v = _v(_head(b1));
    r = _append(r, _bind1(n, v));
  }
  for (; !(b2 == emptybinding); b2 = _tail(b2)) {
    if (_defined(b1, _n(_head(b2)) == false)
      r = _append(r, _head(b2));
  }
  return r;
}
```

Returns a binding formed by appending *b2* to *b1*, giving precedence to *b2* when both *b1* and *b2* contain (name, value) pairs with the same *name*.

```
t_binding
operator++(t_binding b1, t_binding b2) =
{
  val r = emptybinding;
  for (; !(b1 == emptybinding); b1 = _tail(b1)) {
    val n = _n(_head(b1));
    val v;
    if (_defined(b2, n) == true) {
      val v2 = _lookup(b2, n);
      if (_is_binding(v2) == true) {
        v = _v(_head(b1);
        if (_is_binding(v) == true)
          v = operator++(v, v2);
        else v = v2;
      }
      else v = v2;
    }
    else v = _v(_head(b1));
    r = _append(r, _bind1(n, v));
  }
  for (; !(b2 == emptybinding); b2 = _tail(b2)) {
    if (_defined(r, _n(_head(b2)) == false)
      r = _append(r, _head(b2));
  }
  return r;
}
```

Similar to operator+, but performs the operation recursively for each name *n* that is associated with a binding value in both *b1* and *b2*.

```
t_binding
operator-(t_binding b1, t_binding b2) =
{
  val r = emptybinding;
  for (; !(b1 = emptybinding); b1 = _tail(b1)) {
    val n = _n(_head(b1));
    if (_defined(b2, n) == false)
      r = _append(r, _head(b1));
  }
  return r;
}
```

Returns a binding formed by removing from *b1* any pair (*n*, *v*) such that the name *n* is defined in *b2.* The value *v* associated with *n* in *b2* is irrelevant.

```
t_binding
_sub(t_binding b, t_int start = 0, t_int len = _length(b))
{
  int w = _length(b);
  int i = _min(_max(start, 0)), w);
  int j = _min(i + _max(len, 0), w);
  // 0 <= i <= j <= _length(b); extract [i..j)
  t_binding r = emptybinding;
  for (; i < j; i++) r = _append(r, _elem(b, i));
  return r;
}
```

Returns the sub-binding of *b* of length *len* starting at element *start*. Note the boundary cases defined by the pseudo-code; `_sub` produces a runtime error only if it is passed arguments of the wrong type.

```
t_binding
_map(t_closure f, t_binding b) =
{
  t_binding res = emptybinding;
  for (; !(b == emptybinding); b = _tail(l)) {
    // apply the closure "f"
    t_binding b1 = f(_n(_head(b)), _v(_head(b)));
    res = _append(res, b1);
  }
  return res;
}
```

Returns the binding that results from applying the closure *f* to each (*name*, *value*) pair of the binding *b*, and appending the resulting bindings together. The closure *f* should take the *name* (of type t_text) and *value* (of type t_value) as arguments, and return a value of type t_binding. If *f* has the wrong signature, the evaluation halts with a runtime error.

```
t_binding
_par_map(t_closure f, t_binding b)
```

Formally equivalent to `_map`, but the implementation may perform each application of *f* in a separate parallel thread. External tools invoked by `_run_tool` in different threads may be run simultaneously on different machines. If a runtime error occurs in one thread, the other threads may still run to completion before the evaluation terminates.

A.3.4.6 Special Purpose Functions

```
t_closure _self
```

Unless redefined, the name `_self` always refers to the model in which it textually occurs. In effect, every model imports itself under this name, prior to the first `import` clause that appears explicitly in the SDL program text.

```
t_text
_model_name(t_closure m)
```

The value *m* must be a model; that is, a closure defined by importing an immutable file from the Vesta repository. `_model_name` returns a text value that gives one name for *m* within the repository. If a model with identical contents in an identical directory is present at several locations in the repository, the name returned may be that of any of these locations, not necessarily the one that was actually imported in the current evaluation.

```
t_text
_fingerprint(t_value v)
```
The `_fingerprint` primitive returns a text representation of the given value's *fingerprint*, a 128-bit internal identifier for the value. Fingerprints are chosen so that with very high probability, two different values will always have different fingerprints. A given value may have different fingerprints in different evaluations or when computed at different points in the same evaluation, but the implementation tries to avoid this when practical.

Specifically, a source with a particular absolute name in the Vesta repository always has the same fingerprint, while two sources with different names but with the same value will have the same fingerprint if they were fingerprinted *by content* when inserted into the repository. See the documentation of the **vadvance** program for details on when sources are fingerprinted by name and when by content. A derived value returned by any Vesta primitive other than `_run_tool` has a fingerprint that depends deterministically on the fingerprints of its arguments. Derived values returned by `_run_tool` have either arbitrary unique fingerprints or deterministic content-based fingerprints; see Section A.3.4.8 for details.

These properties make fingerprints useful as version stamps for Vesta evaluations, sometimes more useful than `_model_name`. If *m1, m2* are models with identical contents that reside in identical directories, then `_fingerprint(m1) = _fingerprint(m2)` will often be true even when `_model_name(m1) != _model_name(m2)`.

A.3.4.7 Type Manipulation Functions

```
t_text
_type_of(t_value v)
```
Returns a text value corresponding to the type of the value *v* as shown in Table A.9.

```
t_bool
_same_type(t_value v1, t_value v2) =
   operator==(_type_of(v1), _type_of(v2))
```

```
t_bool
_is_bool(t_value v)
```

Value	Text returned by `_type_of`
true, false	“t_bool”
integer	“t_int”
byte sequence	“t_text”
err	“t_err”
list	“t_list”
binding	“t_binding”
closures	“t_closure”

Table A.9: Text values returned by the `_type_of` primitive for each possible input value.

Returns *true* if *v* is of type t_bool; returns *false* otherwise.

```
t_bool
_is_int(t_value v)
```

Returns *true* if *v* is of type t_int; returns *false* otherwise.

```
t_bool
_is_text(t_value v)
```

Returns *true* if *v* is of type t_text; returns *false* otherwise.

```
t_bool
_is_err(t_value v)
```

Returns *true* if *v* is of type t_err; returns *false* otherwise.

```
t_bool
_is_list(t_value v)
```

Returns *true* if *v* is of type t_list; returns *false* otherwise.

```
t_bool
_is_binding(t_value v)
```

Returns *true* if *v* is of type t_binding; returns *false* otherwise.

```
t_bool
_is_closure(t_value v)
```

Returns *true* if *v* is of type t_closure; returns *false* otherwise.

A.3.4.8 Tool Invocation Function

```
t_binding
_run_tool(
  t_text platform,
  t_list command,
  t_text stdin = "",
  t_text stdout_treatment = "report",
  t_text stderr_treatment = "report",
  t_text status_treatment = "report_nocache",
  t_text signal_treatment = "report_nocache",
  t_int  fp_content = -2,
  t_text wd = ".WD",
  t_bool existing_writable = FALSE)
```

`_run_tool` is the mechanism by which external programs like compilers and linkers are executed from a Vesta SDL program. It provides functionality that is fairly platform-independent. The following description, however, is somewhat Unix-specific (for example, in its description of exit codes and signals).

The *platform* argument specifies the platform on which the tool is to be executed. `_run_tool` selects a specific machine for the given platform. The legal values for *platform* and the mechanism by which a machine of the appropriate platform is chosen are implementation dependent.

The tool to be executed is specified by the *command* argument. This argument is a t_list of t_text values. The first member of the list is the name of the tool (interpretation of the name is discussed below); the remaining members of the list are the arguments passed to the tool as its command line. The tool is executed on the specified *platform* in an environment with the following characteristics:

- The file system is encapsulated so that absolute paths (i.e., those beginning with a Delim) are interpreted relative to `./root`, where '.' is the implicit final parameter to `_run_tool`. Non-absolute paths are interpreted relative to `./root/$wd`, where *wd* is a parameter to `_run_tool`. The interpretation of filenames is discussed in more detail below.

- The environment variables are taken from `./envVars`, where '.' is the implicit final parameter to `_run_tool`.

- The content of standard input is the value of the *stdin* `_run_tool` parameter.

- Treatment of standard output and standard error is specified by the *stdout_treatment* and *stderr_treatment* parameters. These parameters may be one of the t_text values `"ignore"`, `"report"`, `"report_nocache"`, `"value"`, or `"report_value"`. If the treatment is `"ignore"`, any bytes written to the corresponding output stream (stdout or stderr) are discarded. If the treatment is `"report"`, the corresponding output is made visible to the user. If the treatment is `"report_nocache"`, the corresponding output is made visible to the user and, if it is not empty, the evaluator does not cache the result of the `_run_tool` call. If the treatment is `"value"`, the output stream is converted to a Vesta value of type t_text and returned as part of the `_run_tool` result, as described below. If the treatment is `"report_value"`, the output stream is both made visible to the user and also returned as part of the result.

- The *status_treatment* and *signal_treatment* arguments may take on the t_text value `"report"` or `"report_nocache"`. Regardless of their values, the `code` and `signal` fields of the result value will be set as described below. If the value of *status_treatment* is `"report_nocache"`, this `_run_tool` call will not be cached if the result `code` is nonzero; similarly, if *signal_treatment* is `"report_nocache"`, the `_run_tool` call will not be cached if the result `signal` is nonzero. Additionally, in our implementation, a runtool call that is not cached because of its return code or signal is considered a runtime error and halts the evaluation with an error message, unless the -k ("keep going") flag is given on the evaluator command line.

- The *fp_content* argument controls how fingerprints are assigned to any derived files created by the tool execution, including derived files created for stdout or stderr when the value of the *stdout_treatment* or *stderr_treatment* parameter is "value". A value of -1 causes the fingerprints of all such derived files to be computed deterministically from their contents. A non-negative *fp_content* value of *x* causes only those files less than *x* bytes in length to have their fingerprints computed from the file contents; an arbitrary unique fingerprint is chosen for files at least *x* bytes in length. Hence, a value of 0 causes all derived files to be assigned arbitrary fingerprints. Setting *fp_content* to -2 selects a site-dependent default value (set by the [Evaluator]/FpContent configuration variable, in our implementation). The boolean values `TRUE` and `FALSE` are accepted as synonyms for -1 and 0, respectively.

  The cost of fingerprinting a file's contents is non-trivial (approximately 1 second per megabyte on the prototype implementation), but doing so allows for cache hits in cases where two evaluations depend on a value that is identical, but was computed in two different ways.

- The *existing_writable* argument controls whether the tool is permitted to write to files that already exist in its encapsulated file system when it is started. If the argument is TRUE, such files may be opened for writing and written to; if it is FALSE, they may not. For technical reasons in the NFS-based repository implementation, tools will get much better file system performance when *existing_writable* is FALSE. It should be set to TRUE only for tools that require it.

The _run_tool primitive returns a binding that contains the results of the command execution. This binding has type:

```
type run_tool_result = binding [
  code   : int,
  signal : int,
  stdout_written : bool,
  stderr_written : bool,
  stdout : text,
  stderr : text,
  root   : binding
]
```

If *r* is of type run_tool_result, then:

- *r*/code is an integer value that characterizes how the command terminated (i.e., the exit status of the Unix process).
- *r*/signal is an integer value identifying the Unix signal that terminated the process, or 0 if the process exited voluntarily.
- *r*/stdout_written and *r*/stderr_written indicate whether data was written to the stdout and stderr streams, respectively.
- *r*/stdout is defined if and only if the *stdout_treatment* _run_tool parameter is "value" or "report_value", in which case it contains the bytes written to stdout.
- *r*/stderr is defined if and only if the *stderr_treatment* _run_tool parameter is "value" or "report_value", in which case it contains the bytes written to stderr.
- *r*/root is a binding containing all files created by the command that are extant upon exit. See the description under “File System Encapsulation” below for details.

Two fine points relating to the results of _run_tool:

1. If the tool cannot be invoked—for example, because of errors in the parameters to `_run_tool`—the evaluator always prints a diagnostic and halts with a runtime error. However, errors that occur during the execution of the tool are reported in a tool-specific fashion, as discussed under `"status_treatment"` and `"signal_treatment"` above.
2. If `"report_nocache"` is specified as the treatment for an output stream (stdout or stderr) or the exit or signal status, the evaluator will not make a cache entry for the `_run_tool` call if any output is produced on the corresponding output stream or if the exit or signal status is nonzero, respectively. In addition, none of the ancestor functions of the failing `_run_tool` call in the call graph are cached either. Since no cache entries are made, a subsequent re-interpretation of the model will produce the same output (on stdout or stderr). This can be useful for reproducing error messages from a compiler or other external tool that are displayed through the Vesta user interface.

File System Encapsulation:

- When the command process (or any subprocess it creates) executes a Unix system call that includes a file path as a parameter, the file path is translated into a reference into the '.' binding that is the last parameter to `_run_tool`.
- The path is interpreted relative to `./root` if it is absolute (i.e., if it begins with "/"), and relative to `./root/$wd` otherwise, where `$wd` is the value of the *wd* parameter to `_run_tool`. Each component of the path—except possibly the final one—must name a Vesta binding. The interpretation of the final component of the path depends on the semantics of the system call. If the system call expects an extant file, the final component must name a Vesta value of type t_text. If the system call expects an extant directory, the Vesta value must be of type t_binding. If the system call expects an unbound name, the name must not be bound by the binding corresponding to the penultimate path component.
- A file created or modified by the command process (or a subprocess) remains visible in the name space throughout the remainder of the process's execution (or until deleted), just as in a regular file system. This is achieved by modeling file creation, modification, and deletion as a suitable overlaying of `./root`. For example, if the process creates "foo.o" in its working directory, this has the effect of:

  ```
  ./root/$wd += [ foo.o = <bytes of file> ];
  <subsequent execution of the command process>
  ```

- File modification is handled in exactly the same way. For example, if the process opens the existing file “foo.db” in its working directory and writes to it, this has the effect of:

  ```
  ./root/$wd += [ foo.db = <new contents of file> ];
  <subsequent execution of the command process>
  ```

  Note that modification of preexisting files is forbidden if the *existing_writable* argument to `_run_tool` is set to FALSE (its default value).

- File deletions are modeled similarly, but the files are removed from the context using the binding difference (-) operator, instead of added using the binding overlay (+) operator.

- When the command process exits, the accumulated effects of the file creations and deletions it has performed are returned as part of the `_run_tool` result (in `r/root`). In this binding, the names of files deleted by the tool are bound to *false*. Such names correspond either to files that existed in `./root` before the tool was invoked, or to files created and subsequently deleted by the tool.

  Thus, if `./root` represents the state of the file system visible to the command process at the time it is launched, then the state of the file system when it exits can be described as:

  ```
  ./root ++ r/root
  ```

  So, if the invoker of `_run_tool` wanted to update `./root` to reflect the changes made by calling `_run_tool`, the code might look like this:

  ```
  r = _run_tool( <suitable parameters> );
  new_fs = ./root ++ r/root;
  . += [ root = new_fs ];
  ```

  After the last assignment, names in `./root` bound to *false* are files that were deleted by the tool. Here is a recursive function for removing such files:

  ```
  remove_deleted(b: binding): binding
  {
    res: binding = [];
    foreach [ n = v ] in b do
      res += if v = false then [] else
        if _is_binding(v)
  ```

```
          then [ $n = remove_deleted(v) ]
          else [ $n = v ];
    return res;
  };
```

A.3.4.9 Diagnostic Functions

```
t_value
_print(t_value v, t_int deps = 0, t_bool verbose = FALSE)
```

Print the value *v* to standard output followed by a newline, and return *v*. What gets printed depends on *v*'s type. If *v* is of type t_err, `ERR` is printed. If *v* is of type t_bool, `TRUE` or `FALSE` is printed. If *v* is of type t_int, its decimal value is printed.

The printed representation of a t_text value is `<file 0x`*XXXXXXXX*`>` if *verbose* is false and the text is represented by a backing file, in which case *XXXXXXXX* is the file's hexadecimal identifier. Otherwise, it is the text value's contents enclosed in double quotes.

The printed representation of a t_list value containing the values $v_1, v_2, \ldots, v_k$ is `<` $p_1, p_2, \ldots, p_k$ `>`, where p_i denotes the printed representation of the value v_i.

The printed representation of a t_binding value containing the (name, value) pairs $(n_1, v_1), (n_2, v_2), \ldots, (n_k, v_k)$ is `[` $n_1 = p_1, \ldots, n_k = p_k$ `]`, where again p_i denotes the printed representation of the value v_i.

The printed representation of a t_closure value is `<Model` *name*`>` if the closure is represented by a model, in which case *name* is a name for the model file in the repository. Otherwise, if *verbose* is true, it is the complete list of formals, body, and context; if not it is simply `<Closure>`.

If *deps* is greater than zero, the value's dependencies are also printed. In the current implementation, values of 1 and 2 provide different levels of detail. This feature is meant for debugging the evaluator itself.

Typically, `_print` is used for debugging purposes, and its result is ignored. However, it is important to remember that `_print` is a function, not a statement. Hence, one cannot simply write:

```
  _print(v);
```

inside a function body. Instead, the call to `_print` must be used in a functional way, such as:

```
  dummy = _print(v);
```

Note also that efficient implementations of the Vesta language will cache function results and re-use those cached results whenever it is safe to do so. Calls to the _print function itself are never cached. However, the _print function's side effect of printing to the terminal is not repeated whenever the call to _print is skipped due to a higher level hit on the function cache.

```
t_bool
_assert(t_bool cond, t_value msg)
```

If the value *cond* is *true*, return *true*. Otherwise, print the value *msg* as with the _print primitive, then terminate the evaluation with a runtime error. As a diagnostic aid, our implementation allows the context of a false assertion and/or a stack trace to be printed as well, if selected by command-line options to the evaluator. Note that, like _print, _assert is a function, not a statement.

A.4 Concrete Syntax

A.4.1 Grammar

Models:

```
Model       ::= Files Imports Block
```

Files Clauses:

```
Files       ::= FileClause*
FileClause  ::= files FileItem*;
FileItem    ::= FileSpec | FileBinding
FileSpec    ::= [ Arc = ] DelimPath
FileBinding ::= Arc = ‘[’ FileSpec*, ‘]’
```

Import Clauses:

```
Imports     ::= ImpClause*
ImpClause   ::= ImpIdReq | ImpIdOpt
ImpIdReq    ::= import ImpItemR*;
ImpItemR    ::= ImpSpecR | ImpListR
ImpSpecR    ::= Arc = DelimPath
ImpListR    ::= Arc = ‘[’ ImpSpecR*, ‘]’
ImpIdOpt    ::= from DelimPath import ImpItemO*;
ImpItemO    ::= ImpSpecO | ImpListO
ImpSpecO    ::= [ Arc = ] Path [ Delim ]
ImpListO    ::= Arc = ‘[’ ImpSpecO*, ‘]’
```

Paths and Arcs:

```
DelimPath   ::= [ Delim ] Path [ Delim ]
Path        ::= Arc { Delim Arc }*
Arc         ::= Id | Integer | Text
```

Blocks and Statements:

```
Block       ::= ‘{’ Stmt*; Result; ‘}’
Stmt        ::= Assign | Iterate | FuncDef | TypeDef
Result      ::= { value | return } Expr
```

Assignment Statements:

```
Assign      ::= TypedId [ Op ] = Expr
Op          ::= AddOp | MulOp
AddOp       ::= +  |  ++  |  -
MulOp       ::= *
```

Iteration Statements:

```
Iterate     ::= foreach Control in Expr do IterBody
Control     ::= TypedId | ‘[’ TypedId = TypedId ‘]’
IterBody    ::= Stmt | ‘{’ Stmt+; ‘}’
```

Function Definitions:

```
FuncDef     ::= Id Formals+ [ TypeQual ] Block
Formals     ::= ( FormalArgs )
FormalArgs  ::= TypedId*,
              | { TypedId = Expr }*,
              | TypedId { , TypedId }* { , TypedId = Expr }+
```

Expressions:

```
Expr        ::= if Expr then Expr else Expr | Expr1
Expr1       ::= Expr2 {  =>  Expr2 }*
Expr2       ::= Expr3 {  ||  Expr3 }*
Expr3       ::= Expr4 {  &&  Expr4 }*
Expr4       ::= Expr5 [ { == | != | < | > | <= | >= } Expr5 ]
Expr5       ::= Expr6 { AddOp Expr6 }*
Expr6       ::= Expr7 { MulOp Expr7 }*
Expr7       ::= [ UnaryOp ] Expr8
UnaryOp     ::= -  |  !
Expr8       ::= Primary [ TypeQual ]
```

```
Primary     ::= ( Expr ) | Literal | Id | List
              | Binding | Select | Block | FuncCall
```

Binary operators with equal precedence are left-associative.

Literals:

```
Literal     ::= ERR | TRUE | FALSE | Text | Integer
```

Lists:

```
List        ::= < Expr*, >
```

Bindings:

```
Binding     ::= ‘[’ BindElem*, ‘]’
BindElem    ::= SelfNameB | NameBind
SelfNameB   ::= Id
NameBind    ::= GenPath = Expr
GenPath     ::= GenArc { Delim GenArc }* [ Delim ]
GenArc      ::= Arc | $ Id | $ ( Expr ) | % Expr %
```

Binding Selections:

```
Select      ::= Primary Selector GenArc
Selector    ::= Delim | !
```

Function Calls:

```
FuncCall    ::= Primary Actuals
Actuals     ::= ( Expr*, )
```

Type Definitions:

```
TypeDef     ::= type Id = Type
TypedId     ::= Id [ TypeQual ]
TypeQual    ::= : Type
Type        ::= any | bool | int | text
              | list [ ( Type ) ]
              | binding ( TypeQual )
              | binding [ ( TypedId*, ) ]
              | function { ( TypedForm*, ) }* [ TypeQual ]
              | Id
TypedForm   ::= [ Id : ] Type
```

A.4.2 Ambiguity Resolution

The grammar as given above is ambiguous. We resolve the ambiguity as follows.

The Vesta parser accepts a modified grammar in which the > token is replaced by two distinct tokens: GREATER in the production for Expr4 and RANGLE in the production for List. The modified grammar is unambiguous and can easily be parsed by an LL(1) or LALR(1) automaton.

The Vesta tokenizer is responsible for disambiguating between GREATER and RANGLE wherever > appears in the input. It does so by looking ahead to the next token after the >. If the next token is one of

```
- ! ( ERR TRUE FALSE Text Integer Id < [ {
```

then the > is taken as GREATER; otherwise, it is taken as RANGLE.

Why is this solution reasonable? Inspection of the grammar shows that in a syntactically valid program, the next token after GREATER must be one of those in the list above. The next token after RANGLE must be one of the following:

```
: * + ++ - == != < GREATER <= >= && || =>
; do , ) then else RANGLE ] % / \ ! (
```

These sets overlap in the tokens `-`, `!`, `(`, and `<`. Because we have chosen to resolve these cases as GREATER, it is impossible to write certain syntactically valid programs containing RANGLE. However, any such program can be rewritten by replacing every List nonterminal by (List), yielding a semantically equivalent program in which the closing > of the List is correctly resolved as RANGLE. Moreover, we claim (without presenting a proof) that any program in which RANGLE is followed by `-`, `!`, `(`, or < must have a runtime type error, due to the paucity of operators defined on the list type, so in practice such programs are never written.

A.4.3 Tokens

Table A.10 gives a BNF description of the tokens of the language. The token classes Delim, Integer, Id, and Text, and the individual tokens in the classes Punc, TwoPunc, and Keyword, serve as terminals in the BNF of earlier sections.

We define Newline as an ASCII new line sequence, either CR, LF, or CRLF. NonNewlineChar is any ASCII character other than CR and LF. CommentBody is any sequence of ASCII characters that does not contain '*/'. Tab is the ASCII TAB character.

The ambiguities in the token grammar are resolved as follows. The tokenizer interprets the program as a TokenSeq. It scans from left to right, repeatedly matching the longest possible Token beginning with the next unmatched character. The

```
TokenSeq   ::= Token*
Token      ::= Integer | Id | Text | Punc
             | TwoPunc | Keyword | Whitespace
             | Comment

Delim      ::= /  |  \

Integer    ::= DecimalNZ Decimal*
             | 0 Octal* | 0 { x | X } Hex+
Decimal    ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
DecimalNZ  ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Octal      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Hex        ::= Decimal
             | A | B | C | D | E | F
             | a | b | c | d | e | f

Id         ::= { Letter | Decimal | IdPunc }+
Letter     ::= A | B | C | D | E | F | G | H | I | J | K
             | L | M | N | O | P | Q | R | S | T | U | V
             | W | X | Y | Z
             | a | b | c | d | e | f | g | h | i | j | k
             | l | m | n | o | p | q | r | s | t | u | v
             | w | x | y | z
IdPunc     ::= .  |  _

Text       ::= " TextChar* "
TextChar   ::= Decimal | Letter | Punc | Escape
Punc       ::= ˜ | ‘ | ! | @ | # | $ | % | ˆ | & | * | ( | )
             | _ | - | + | = | ‘{’ | ‘[’ | ‘}’ | ‘]’ | : | ;
             | ‘|’ | ’ | , | < | . | > | ? | / | Space
Escape     ::= \ EscapeChar
EscapeChar ::= n | t | v | b | r | f | a | \  | "
             | Octals |  Hexes
Octals     ::= Octal [ Octal [ Octal ] ]
Hexes      ::= { x | X } Hex [ Hex ]

TwoPunc    ::= ++ | == | != | <= | >= | => | || | &&

Keyword    ::= binding | do | else | ERR | FALSE | files
             | foreach | from | function | if | in | import
             | list | return | then | type | TRUE | value

Whitespace ::= ‘ ’ | Tab | Newline

Comment    ::= // NonNewlineChar* Newline
            |  ‘/*’ CommentBody ‘*/’
```

Table A.10: BNF for the tokens of the Vesta SDL.

tokens Whitespace and Comment are discarded after matching; other tokens are passed on for parsing by the main grammar. When a string of characters matches both Integer and Id, it is tokenized as Integer. When a string matches both Keyword and Id, it is tokenized as Keyword.

A.4.4 Reserved Identifiers

Here are the Vesta SDL's reserved identifiers; they should not be redefined:

```
_append _assert _bind1 _defined _div _elem _find _findr
_fingerprint _head _is_binding _is_bool _is_closure
_is_err _is_int _is_list _is_text _length _list1 _lookup
_map _max _min _mod _model_name _n _par_map _print
_run_tool _same_type _self _sub _tail _type_of _v
```

Acknowledgements

The Vesta project has extended over many years, and there have been many contributors and supporters.

We begin by thanking those who participated in the development of Vesta-1, including Bob Ayers, Mark R. Brown, Sheng-Yang Chiu, John Ellis, Chris Hanna, and Paul McJones.

Butler Lampson was one of the primary designers of the Vesta-2 function cache and weeder. He also contributed to the initial design of the system modeling language and repository. Butler, Jim Horning, and Martín Abadi helped design the evaluator's fine-grained dependency algorithm. Together with Chris Hanna, Jim also participated in the design of the system modeling language and the initial implementation of the evaluator.

Bill McKeeman's suggestions helped us to make the modeling language syntax simpler and more readable. Our fingerprint package is based on code and ideas from 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 a draft of this report. 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, while Cynthia Hibbard proofread a draft of the entire manuscript and provided numerous corrections. 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 large user community, and Walker Anderson and Joford Lim led the group's initial evaluation of Vesta. Matt Reilly and Ken Schalk were the prime movers in carrying the Araña Vesta effort through to adoption and production use. Now that we have moved on to other activities, Ken has become the primary maintainer of the entire Vesta system. Ken and Matt have also done the bulk of the work on the Linux port.

Bibliography

[1] Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 83–91, May 1996.

[2] AccuRev, Inc. Accurev. http://www.accurev.com/i_prod.html.

[3] Larry Allen, Gary Fernandez, Kenneth Kane, David Leblang, Debra Minard, and John Posner. ClearCase MultiSite: Supporting geographically-distributed software development. In *Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers*, pages 194–214, 1995. Available as volume 1005 in *Lecture Notes in Computer Science*, Springer-Verlag.

[4] Atria Software, Inc., 24 Prime Park Way, Natick, MA 01760. *ClearCase Concepts Manual*, 1992.

[5] Dave Belanger, David Korn, and Herman Rao. Infrastructure for wide-area software development. In *Proceedings of the 6th International Workshop on Software Configuration Management*, pages 154–165, 1996. Available as volume 1167 in *Lecture Notes in Computer Science*, Springer-Verlag.

[6] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[7] Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber. A simple and efficient implementation for small databases. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 149–154. The Association for Computing Machinery, November 1987.

[8] BitMover, Inc. Bitkeeper. http://www.bitkeeper.com/.

[9] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[10] Andrei Broder. Some applications of Rabin's fingerprinting method. In R. Capocelli, A. De Santis, and U. Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[11] Mark R. Brown and John R. Ellis. Bridges: Tools to extend the Vesta configuration management system. SRC Research Report 108, Digital Equipment Corporation, Systems Research Center, June 1993. http://gatekeeper.research.compaq.com/pub/compaq/SRC/research-reports/abstracts/src-rr-108.html.

[12] Todd Brunhoff. *makedepend(1) manual page*.

[13] Sheng-Yang Chiu and Roy Levin. The Vesta repository: A file system extension for software development. SRC Research Report 106, Digital Equipment Corporation, Systems Research Center, June 1993. http://gatekeeper.research.compaq.com/pub/compaq/SRC/research-reports/abstracts/src-rr-106.html.

[14] Computer Associates. CCC/Harvest. http://www.cai.com/products/ccm/.

[15] Connectathon test suites. http://www.connectathon.org/nfstests.html.

[16] Jacky Estublier and Rubby Casallas. The adele configuration manager. In Walter Tichy, editor, *Configuration Management*, volume 2 of *Trends in Software*, pages 99–134. John Wiley and Sons Ltd., 1994. Available at http://www-adele.imag.fr/Les.Publications/BD/adele1994est.html.

[17] S. I. Feldman. Make—A program for maintaining computer programs. *Software—Practice and Experience*, 9(4):255–265, April 1979.

[18] Glenn Fowler. A case for make. *Software—Practice and Experience*, 20(S1):S1/35–S1/46, June 1990.

[19] Free Software Foundation. GNU lesser general public licence. http://www.fsf.org/licenses/lgpl.html.

[20] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210. The Association for Computing Machinery, December 1989.

[21] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[22] Dick Grune, Brian Berliner, and Jeff Polk. *cvs(1) manual page*. Free Software Foundation.

[23] Carl A. Gunter. Abstracting dependencies between software configuration items. *ACM Transactions on Software Engineering and Methodology*, 9(1):94–131, January 2000.

[24] Christine B. Hanna and Roy Levin. The Vesta language for configuration management. SRC Research Report 107, Digital Equipment Corporation, Systems Research Center, June 1993. http://gatekeeper.research.compaq.com/pub/compaq/SRC/research-reports/abstracts/src-rr-107.html.

[25] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. The Vesta approach to software configuration management. SRC Research Report 168, Compaq Computer Corporation, Systems Research Center, March 2001. http://gatekeeper.research.compaq .com/pub/compaq/SRC/research–reports/abstracts/src–rr–168.html.

[26] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 311–320, June 2000.

[27] Allan Heydon and Greg Nelson. The Juno-2 constraint-based drawing editor. SRC Research Report 131a, Digital Equipment Corporation, Systems Research Center, December 1994. http://gatekeeper.research.compaq.com/pub/compaq/SRC/research-reports/abstracts/src-rr-131a.html.

[28] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanaraynan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[29] Juno-2 home page. http://www.research.compaq.com/SRC/juno-2/.

[30] David B. Leblang and Robert P. Chase, Jr. Computer-aided software engineering in a distributed workstation environment. *SIGPLAN Notices*, 19(5):104–112, May 1984.

[31] David B. Leblang, Robert P. Chase, Jr., and Gordon D. McLean, Jr. The DOMAIN software engineering environment for large-scale software development efforts. In *Proceedings of the 1st International Conference on Computer Workstations*, pages 266–280, San Jose, CA, November 1985. IEEE Computer Society, IEEE Computer Society Press. ISBN 0-8186-0649-5, IEEE Catalog Number 85CH2228-5.

[32] Roy Levin and Paul R. McJones. The Vesta approach to precise configuration of large software systems. SRC Research Report 105, Digital Equipment Corporation, Systems Research Center, June 1993. http://gatekeeper.research.compaq.com/pub/compaq/SRC/research-reports/abstracts/src-rr-105.html.

[33] J. MacDonald, P. N. Hilfinger, and L. Semenzato. PRCS: The project revision control system. In *Proceedings of the 8th International Workshop on System Configuration Management*, pages 33–45, 1998. Available as volume 1439 in *Lecture Notes in Computer Science*, Springer-Verlag, and at http://citeseer.nj.nec.com/279027.html.

[34] Boris Magnusson and Ulf Asklund. Fine grained version control of configurations in COOP/orm. In *Proceedings of the 6th International Workshop on System Configuration Management*, pages 31–48, 1996. Available at http://citeseer.nj.nec.com/magnusson96fine.html.

[35] Timothy Mann. Partial replication in the Vesta software repository. SRC Research Report 172, Compaq Computer Corporation, Systems Research Center, August 2001. http://gatekeeper.research.compaq.com/pub/compaq/SRC/research–reports/abstracts/src–rr–172.html.

[36] Timothy Mann, Andrew Birrell, Andy Hisgen, Charles Jerian, and Garret Swart. A coherent distributed file cache with directory write-behind. SRC Research Report 103, Digital Equipment Corporation, Systems Research Center, June 1993. http://gatekeeper.research.compaq.com/pub/compaq/SRC/research-reports/abstracts/src-rr-103.html.

[37] Peter Miller. Aegis. http://www.canb.auug.org.au/ millerp/aegis/aegis.html.

[38] Modula-3 home page. http://www.research.compaq.com/SRC/modula-3/html/home.html.

[39] Greg Nelson, editor. *Systems Programming with Modula-3*. Series in Innovative Technology. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991.

[40] John Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer USENIX Conference*, pages 247–256, Anaheim, CA, Summer 1990.

[41] Perforce Software. Perforce. http://www.perforce.com/perforce/products.html.

[42] M. O. Rabin. Fingerprinting by random polynomials. Report TR–15–81, Department of Computer Science, Harvard University, 1981.

[43] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE–1(4):364–370, December 1975.

[44] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implemention of the Sun network filesystem. In *Proceedings of the Summer USENIX Conference*, pages 119–130, June 1985.

[45] Kenneth C. Schalk. Vesta SDL Programmer's Reference. http://doc/sdl-ref/.

[46] Telelogic AB. Telelogic CM Synergy (formerly Continuus/CM). http://www.telelogic.com/products/synergy/.

[47] W. Tichy. Design, implementation, and evaluation of a revision control system. In *Proceedings of the 6th International Conference on Software Engineering*, pages 58–67. IEEE Computer Society Press, 1982.

[48] W. Tichy. RCS—A system for version control. *Software—Practice and Experience*, 15(7):637–654, July 1985.

[49] TRUE Software. TRUEchange (formerly known as Aide-De-Camp or ADC). http://www.truesoft.com/.

[50] André van der Hoek, Antonio Carzaniga, Dennis Heimbigner, and Alexander L. Wolf. A testbed for configuration management policy programming. *IEEE Transactions on Software Engineering*. To appear. Available from http://www.ics.uci.edu/ andre/.

[51] Vesta Home Page. http://www.vestasys.org.