105

The Vesta Approach to Precise Configuration of Large Software Systems

Roy Levin
Paul R. McJones

June 14, 1993

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

The Vesta Approach to Precise Configuration of Large Software Systems

Roy Levin
Paul R. McJones

June 14, 1993

Authors' Abstract

The problems of software configuration and release management limit the size of systems that we can build efficiently. Today's large systems strain the capabilities of traditional development tools. The Vesta system provides a novel repository and system builder that emphasize complete yet manageable descriptions of software components. These facilities enable Vesta to eliminate much of the manual and error-prone drudgery of system construction without enforcing a particular methodology on its users. This paper presents an overview of Vesta, followed by a series of detailed examples that illustrate how Vesta's facilities simplify the development of large systems. These examples are drawn from a year's use of Vesta by a group of about 25 researchers developing a rapidly changing software system of over 1.4 million source lines. That experience clearly demonstrates the power and practicality of the Vesta approach and its advantages over conventional tools.

Contents

List of Figures

1 Introduction

Organizations that develop software inevitably face the considerable problems of configuration and release management. They must arrange their evolving body of software and documentation so that new versions can be produced, tested, and distributed efficiently without compromising the integrity of previously completed versions. Too often, the available management tools simply cannot cope with large and complex systems. When the tools break down, people must intervene manually, and their inevitable errors are generally difficult to detect and tedious to correct.

The Vesta configuration management system addresses this problem with a novel integrated storage system (repository) and system builder. In this paper we show how we used these Vesta facilities to describe system components and their interconnections. We illustrate our method with concrete examples and explain how it is able to cope with the essential problems of configuration and release management.

Those problems become evident only when systems grow beyond a certain size. So, to validate our approach, we used the Vesta system to organize and manage a large body of software that was under active development in 1991–92 at Digital Equipment Corporation's Systems Research Center (SRC). That software included about 4400 modules comprising 1.4 million lines of source code, mostly written in Modula-2+ [Rovner *et al.*], but with some C and assembly language. The code included many different kinds of subsystems, such as a micro-kernel operating system, a comprehensive set of user libraries, software development tools, and a substantial collection of application programs. New versions of these components were being produced frequently by the 25 full-time developers, and changes in the interfaces between major components occurred often, necessitating frequent integration. The development tools (e.g., compilers) were evolving concurrently with the software that was built with them. Nearly every component was targeted to run on multiple operating systems and machine architectures.

Such a setting clearly presents a substantial challenge for a configuration management system, and the traditional Unix software development tools were inadequate. Their deficiencies became the requirements for Vesta's configuration management facilities:

1. *Human work should be commensurate with the size of a change.* In Vesta, the amount of effort required for a developer to propagate a change is proportional to the size of the change, not the size of the affected body of code. We emphasize that this requirement addresses *conceptually* small changes; in this sense both "change the local variable `x` to `y` in this line of C-code" and "rebuild the system for a different instruction set architecture" are small changes. Notice that many conventional development tools limit the kinds of conceptually small changes that are feasible. For

example, it may be difficult to rebuild a large system consistently with a new version of a compiler, or to rebuild a collection of related applications with a new subroutine library.

We clearly distinguish the human and machine work required to effect a change. Requirement 1 talks about human work, but says nothing about machine work. Of course, the amount of machine work must be reasonable for the particular change. It makes sense to recompile every module of a system if the compiler version changes, but not if a local variable name changes.

2. *Tools should support programmers working individually and cooperatively.* Vesta supports development of large systems without complicating the creation of small programs. The system-building descriptions are concise; that is, the size and complexity of the descriptions the individual programmer must understand or write scale with the size and complexity of the *individual* piece of software being developed, not the entire system.

3. *Software construction recipes should be complete and self-contained.* Any object (e.g., executable program, documentation) constructed by Vesta can be traced back to a collection of source objects. Also, the construction of any object can be reproduced by Vesta, if necessary.

These requirements led us to adopt a particular methodology in organizing our descriptions of software construction (in Vesta terminology, *system models*, or *models* for short [Lampson and Schmidt]). The methodology has the following characteristics:

- *Modularity.* System models are modular, not monolithic. An individual working on a single component of a large system should, in the main, be able to carry out development and unit testing by manipulating only the system model of that component (requirement 2). But since a system description must be complete (requirement 3), it follows that models necessarily refer to each other.

- *Isolation of construction algorithm and environment.* System models completely define both construction algorithms and construction environments, but keep them carefully separated. The *construction algorithm* defines the set of pieces that comprise a particular component and the rules for assembling them. That assembly also incorporates externally defined components, which are provided by the *construction environment.* Both must be specified precisely if the system description is to be complete (requirement 3). Keeping their specification carefully separated follows from requirement 2, since individual developers need to perform unit testing of their components, while integrators need to test collections of components together, and the environments for constructing these two sorts of tests necessarily differ.

- *Extensive parameterization.* The construction algorithm for each component of a system is highly parameterized. It binds as few decisions as possible, leaving most of the choices to be bound by the environment in which the construction occurs. By accepting many parameters, the construction algorithm becomes usable in a wide variety of circumstances, and a conceptually small change generally translates into a change in a parameter value (requirement 1).

The remaining sections of the paper present detailed examples of this methodology. Section 2 gives an overview of the essential properties of the Vesta repository and builder on which the methodology depends. Section 3 introduces the basic forms of Vesta system models and gives concrete examples of each. Section 4 illustrates how system models are combined to build up large systems from their components. Section 5 addresses scaling issues, showing how to construct a large, consistent development environment with Vesta models. Section 6 concludes the paper with an evaluation of our approach.

2 Vesta Fundamentals

In this section, we give an overview of selected aspects of Vesta that are necessary for understanding the subsequent examples. We don't attempt to cover any topic in depth. Companion papers [Chiu and Levin] [Hanna and Levin] [Brown and Ellis] provide more comprehensive discussions of Vesta's repository and system-builder.

2.1 Repositories, Objects, Packages, and System Models

Vesta stores *objects*[1] in *repositories.* Every object is either a *source* object or a *derived* object. The Vesta system regards source objects as atomic entities that it cannot construct from other objects. Derived objects are just the opposite—Vesta constructs them mechanically from other objects (and, significantly, using other objects). All objects—source and derived—possess unique, machine-oriented names (UIDs) and are immutable, meaning that the binding between a unique name and a value, once established, never changes.

A Vesta *repository* is a specialized file system that stores and names Vesta objects. Source objects have user-sensible, hierarchical names similar to Unix paths, but which include versioning information (see below). Derived objects have no user-sensible names; they can be named only by their UIDs. (For details on the representation and management of Vesta objects, see [Chiu and Levin].)

A Vesta repository may be either *public* or *private.* A private repository belongs to a single user, meaning that only that user can create new objects in

[1]Vesta objects resemble files in many ways, although, as we shall see, there are significant differences [Chiu and Levin]. For the purposes of this paper, however, the reader can treat the two terms as synonymous.

it. Any user can create objects in a public repository. All users can read objects in any repository.

A Unix file system provides a naming hierarchy using directories. Each directory holds a collection of named entities, each of which is either a file or another directory. A Vesta repository is slightly more complicated; it consists of *packages* and *system models*, each of which participates in the naming hierarchy in addition to performing other functions. We'll briefly describe each of these concepts.

A *package* is the unit of software conveniently worked on by one person at a time. This obviously isn't a very formal definition; operationally, it means that Vesta repositories provide checkout/checkin on packages, not individual files. (We'll explain this is some detail in section 2.5.) Vesta packages are versioned; each version is described by a *system model*, which is a source object that can name other source objects. So, a Vesta *package version* consists of a collection of named source objects.

Once created, a package version is immutable (since it is described by a system model, which is a Vesta object, and all Vesta objects are immutable). Thus, we can't change a package version, but we can create new ones.

Package versions are naturally grouped into *package families*, which reflect the evolution of the software unit embodied by the package. Each repository has a flat name space of package families; each family has a tree-structured name space that resembles RCS [Tichy], but uses a mixture of textual and numerical naming elements. For example, suppose we have a repository named `/vesta/proj` that contains a package family named `text`. Then the following are all package versions in this family:

```
/vesta/proj/text.4
/vesta/proj/text.4.levin.17
/vesta/proj/text.8.for_vax.1.bugfix.2
```

For the purposes of this paper, we needn't delve any further into the interpretation of these names. More details can be found in [Chiu and Levin].

A *system model* has three parts: a *directory clause*, an optional *import clause*, and a *building part*. The directory clause binds simple, human-sensible names to source objects identified by UIDs, analogously to a traditional file system directory. The import clause binds simple names to the unique names of other package versions; that is, it enables one system model to refer to another, somewhat like a link in a Unix file system. Together, the directory and import clauses define a naming environment in which the building part is interpreted. The building part contains the rules for constructing derived objects. It is written in a functional programming language (the Vesta language [Hanna and Levin]) that is based on the typed λ-calculus.

The directory clause refers to source objects that "belong" to a package version, in the same sense that the files in a Unix directory belong to that directory. Any of these source objects may itself be a system model, called a *submodel*, in which case it introduces another level in the naming hierarchy, just like a Unix subdirectory. Two different system models may refer to the same UID in their directory clauses; the Unix analogy is multiple hard links. This situation arises often within a package family, since successive versions of a package frequently have many source objects in common.

The import clause refers to other package versions. These references cut across the normal naming hierarchy, much as symbolic links do in Unix. Consider the following two package versions and their directory and import clauses:

> `/vesta/proj/text.4` (bound to some UID, say *uid-1*) is a system model that begins:

```
  DIRECTORY
    Text.def = uid-2,
    Text.mod = uid-3
  ...
```

> `/vesta/proj/stdio.7` (bound to some UID, say *uid-4*) is a system model that begins:

```
  DIRECTORY
    Stdio.def = uid-5,
    Stdio.mod = uid-6
  IMPORT
    text.v (*/vesta/proj/text.4*) = uid-1
  ...
```

These two package versions comprise six distinct source objects, identified by the six UIDs. Their natural user-sensible names are:

`/vesta/proj/text.4/MODEL`	for	*uid-1*
`/vesta/proj/text.4/Text.def`	for	*uid-2*
`/vesta/proj/text.4/Text.mod`	for	*uid-3*
`/vesta/proj/stdio.7/MODEL`	for	*uid-4*
`/vesta/proj/stdio.7/Stdio.def`	for	*uid-5*
`/vesta/proj/stdio.7/Stdio.mod`	for	*uid-6*

However, the following are also valid user-sensible names:

`/vesta/proj/stdio.7/text.v`	for	*uid-1*
`/vesta/proj/stdio.7/text.v/Text.def`	for	*uid-2*
`/vesta/proj/stdio.7/text.v/Text.mod`	for	*uid-3*

The directory and import clauses both introduce names that can be referenced in the building part of the model, as we'll see in the next section.

2.2 Dependency and Parameterization

Because it is a functional programming language, the Vesta language conveniently expresses the essence of the system construction process: the application of software tools to objects according to a dependency description. We represent each tool (e.g., a C compiler, a linker) used by a system model as a Vesta function[Brown and Ellis]. The function's arguments are the tool's inputs, which are typically one or more Vesta objects (files) and other parameters corresponding to switches or options (e.g., to enable optimization or runtime range checks). The function's result is typically a Vesta derived object that can serve as the input to another function. So, a fragment of (the building part of) a system model that compiles and links a two-module program might look like this:

```
M2$Prog( (M2$Compile(Main.mod), M2$Compile(Subs.mod)) )
```

`M2$Compile` and `M2$Prog` are Vesta functions that invoke the Modula-2+ compiler and linker, respectively. This fragment invokes the compiler twice, combines the results in a two-element list, and passes the list to the linker. The value of the fragment therefore depends on the two source files (`Main.mod` and `Subs.mod`) and the two functions (`M2$Compile` and `M2$Prog`).

Let's expand this fragment according to the methodology described in section 1. We want to parameterize the construction algorithm of this hypothetical program, so we introduce a function[2], as shown in figure 1.

```
01  DIRECTORY
02    Main.mod = /v/0haM08002P.Zay/S94f.73.sample,
03    Subs.mod = /v/0haM08002P.Zay/S100a.4.sample,
04  IN  {
05      build = FUNCTION M2$Prog, M2$Compile IN
06          M2$Prog( (M2$Compile(Main.mod), M2$Compile(Subs.mod)) );
07
08      other stuff
09      }
```

Figure 1: System model fragment with a `build` function

We still have a fragment, but one that is closer to a real Vesta model. Note the directory clause, which binds the names `Main.mod` and `Subs.mod` to unique identifiers (whose format is typical, but doesn't concern us here[3]). The building

[2]In this example, and many that follow, we include line numbers for convenient reference in the subsequent text. These numbers don't appear in real system models.

[3]Unique identifiers are intended for machine, not human, consumption, so although they are visible in this example, users don't actually type or manipulate them directly. Section 2.5 explains how unique identifiers are introduced into models.

part is a *binding*, in which the name `build` is bound to a function produced by evaluating the `FUNCTION` expression. We'll examine bindings in more detail in the next section.

Here, the `build` function has two parameters, `M2$Compile` and `M2$Prog`.[4] In the real-life examples in subsequent sections, we'll see that the set of parameters of a typical function is much larger. Modula-2+ modules often import tens of interfaces, meaning that an invocation of `M2$Compile` often depends on tens of values that the environment must supply. So, if `Main.mod` were a typical Modula-2+ program, it might import a dozen interfaces, and if we had to write explicitly the values for those interfaces as additional parameters to the compilation, our system model would become unwieldy. Furthermore, we would also have to add their names to the formal parameter list of the `build` function, thereby making the system model even bulkier.

Clearly, we need a way to suppress details that are easily inferred. The Vesta language provides two *defaulting* mechanisms for this purpose. First, any actual parameters omitted from a function call are supplied by looking up the corresponding formal parameters in the naming environment of the call. Second, some or all of the formal parameter list in a `FUNCTION` expression may be replaced by "`...`", which indicates that any free variable in the body of the `FUNCTION` expression that is not bound in the lexical scope enclosing the `FUNCTION` expression is to be treated as an implicit formal parameter. So, we would typically write lines 5–6 in the fragment in figure 1 as:

```
05      build = FUNCTION ... IN
06          M2$Prog( (M2$Compile(Main.mod), M2$Compile(Subs.mod)) );
```

Notice that, as a consequence of these defaulting rules, any interface imported by `Main.mod` is now implicitly a formal parameter of `build`. That is, since the lexical environment of this function binds only the source file names (in the `DIRECTORY` clause), all the *defaulted* parameters to the compilation and linking become formal parameters of `build`, along with the compiler and linker themselves.

2.3 Bindings

During the process of constructing substantial software systems, the naming environment can grow to be quite large. Since Vesta system models are complete and self-contained (requirement 3), they necessarily include large collections of names. As a result, the Vesta language has to provide mechanisms for conveniently manipulating sets of named values. We call a set of <name, value> pairs a *binding*. Technically, a binding is a (partial) function from names to values; that is, a name may appear at most once in a binding. Bindings are

[4]This is a deliberate oversimplification. The truth comes at the end of section 2.3.

first-class values in the Vesta language; in particular, a binding may be (and often is) returned as the result of a function call.

The value of each of the following expressions is a binding:

```
{ A = 3, optimize = FALSE }

{ Double = FUNCTION x IN PLUS(x,x),
  Triple = FUNCTION x IN PLUS(PLUS(x,x),x) }

{ pair = { first = 1, second = 3 }, something_else = "abc" }
```

The following expressions evaluate to the same value:

```
LET { x = 3, y = 4 } IN { sum = PLUS(x, y) }
```

and

```
{ sum = 7 }
```

The following expressions evaluate to the same value:

```
LET { Z = FUNCTION x IN { x2 = PLUS(x,x),
                          x3 = PLUS(PLUS(x,x),x) } }
IN Z(7)
```

and

```
{ x2 = 14, x3 = 21 }
```

If `B1` and `B2` are bindings, the construct {`B1,B2`} produces a binding that is the union of `B1` and `B2`, with the members of `B2` taking precedence when the same name appears in both bindings. If `B1` and `B2` are bindings, the construct {`B1;B2`} produces a binding equivalent to {`B1, LET B1 IN B2`}. If `B` is a binding and `N` is an identifier, the construct `B$N` produces the value that is paired with `N` in `B`. We use these constructs extensively in the examples in subsequent sections.

The following five expressions evaluate to the same value:

```
LET { f = FUNCTION x IN { x2 = PLUS(x,x),
                          x3 = PLUS(PLUS(x,x),x) } } IN
    LET f(7) IN PLUS(x2, x3)
```

and

```
LET { f = FUNCTION x IN { x2 = PLUS(x,x),
                          x3 = PLUS(PLUS(x,x),x) }; f(7) } IN
    PLUS(x2, x3)
```

and

```
LET { f = FUNCTION x IN { x2 = PLUS(x,x),
                          x3 = PLUS(PLUS(x,x),x) } } IN
    PLUS(f(7)$x2, f(7)$x3)
```

and

```
LET { x2 = FUNCTION ... IN PLUS(x,x),
      x3 = FUNCTION ... IN PLUS(PLUS(x,x),x) } IN
  LET { x = 7 } IN
      PLUS(x2(), x3())
```

and

```
35
```

Notice, therefore, that when we wrote `M2$Compile` in the fragments in section 2.2, we were actually writing an expression, not a simple identifier. In reality, then, the `build` function in that section has `M2` as a formal parameter, not `M2$Compile` and `M2$Prog`.

2.4 Modula-2+ programs

Since most of the source code we developed using Vesta is written in Modula-2+ [Rovner *et al.*], the characteristics of that language naturally affected the structure of the system models we wrote. For the purposes of these descriptions, Modula-2+ is identical to Modula-2 [Wirth]. It has formal interfaces (called *definition modules*) which are compiled into an intermediate form. The compiled representation of an interface is needed (read) by the compiler when it compiles a module that imports (uses) the interface. Such a module may be another interface, an implementation module, or a main program module. The compiler must also read the compiled interface when compiling an implementation module that exports (implements) the interface. The linker accepts a compiled main program plus a collection of compiled implementation modules and checks that they were all produced using a consistent set of interfaces.

Since two Modula-2+ modules can refer to each other only through a Modula-2+ interface, we can see that the system model fragment in figure 1 isn't quite complete—it should include an interface. In the fragment that appears in figure 2 we've added one. To make it look more like the real system models we will encounter later, we've given a name to the result returned by `Prog`, and exploited some of the language mechanisms for manipulating bindings to tidy things up a bit.

On line 7 this model fragment compiles the Modula-2+ interface named `Subs.def`, producing a single-element binding. By convention, the Modula-2+ compiler attaches the suffix "`.d`" to compiled interface names, that is, on line 7 it constructs the binding

```
01  DIRECTORY
02      Main.mod = uid-1,
03      Subs.mod = uid-2,
04      Subs.def = uid-3,
05  IN  {
06      build = FUNCTION ... IN LET M2 IN
07          LET Compile(Subs.def) IN
08              { Sample = Prog( (Compile(Main.mod),
09                                Compile(Subs.mod)) ) };
10
11      other stuff
12      }
```

Figure 2: System model fragment with a Modula-2+ interface

```
{ Subs.d = compiled version of Subs.def }
```

The two compilations on lines 8–9 both need to refer to this compiled interface, since `Main.mod` imports it and `Subs.mod` exports it. The `LET` expression on line 7 places `Subs.d` in the lexical environment where it can be found by the two subsequent compilations. `Subs.d` is therefore an implicit (i.e., defaulted) parameter to these compilations.

2.5 Where new versions come from

To conclude our overview of Vesta fundamentals, we briefly consider how new package versions come into existence.

In most source-code control systems (e.g., `SCCS` [Rochkind], `RCS` [Tichy], and their descendants), versions of source files are created by a *checkout-checkin* process. Roughly speaking, a new version of a source file is created by copying some existing version from a public directory into a private working area (checkout), editing it one or more times to produce a new version, and copying the result back to the public directory (checkin). Most systems combine some concurrency control with either or both of the copying actions in order to reduce the possibility of chaos and lost edits.

Vesta also has a checkout-checkin process, but it differs in several significant ways:

- The unit of checkout-checkin is a package version, not a single file. Recall that a package version is completely described by a system model, which lists all the source objects that comprise the package. So, in Vesta, checkout-checkin involves copying a directory tree rather than a single file.

- Vesta manages derived files as well as source files. Traditional source-code control systems, true to their name, manage only source code, leaving the user to manage the naming of derived objects manually. In Vesta, derived objects don't have user-sensible names; they are managed entirely "under the covers" by Vesta (that is, they can be named only with UIDs). The user works exclusively with source files.

- Versioning is available to the programmer during a development session, that is, while files are checked out. We've seen that package families contain multiple package versions, organized in a tree. This arrangement has obvious benefits for the shared public versions. However, programmers also find it convenient to retain intermediate versions during development sessions. Vesta extends the package versioning scheme to include these intermediate versions.

To understand the practical consequences of these differences, let's walk through a development session using Vesta. Assume that our public repository is named `/vesta/proj` and that it contains a package family named `text` in which we want to create a new version. By browsing, we discover that version 4 of the package (i.e., `/vesta/proj/text.4`) is the one we want to start from; we propose to create `/vesta/proj/text.5`, which doesn't yet exist. Suppose that the system model named `/vesta/proj/text.4` has the form shown in figure 3.

```
DIRECTORY
    Text.def = uid-1,
    Text.mod = uid-2
IN  {
    the building part; not relevant just now
    }
```

Figure 3: System model at the start of a development session

We initiate a *checkout session* by invoking Vesta's *Checkout* operation and specifying

- the version we intend to create (`/vesta/proj/text.5`),

- the version we intend to start from (`/vesta/proj/text.4`),

- the private repository to hold the intermediate versions that we create during the checkout session (call it `/vesta/levin`), and

- the (Unix) working directory in which we will perform our editing (call it `/udir/levin/text`).

When Checkout completes, the name `/vesta/proj/text.5` will have been reserved for us and the working directory `/udir/levin/text` will have been suitably initialized, as discussed below.

Already things look quite different from conventional source-control systems. Vesta evaluates only immutable system models, which reside in repositories. Yet while a package is checked out, many versions of only transient interest are created. These versions reside in the private repository, which also holds the derived objects produced by evaluation of these versions. The working directory holds only mutable files that correspond to the source objects of the package. Let's look at this in more detail.

After the Checkout operation, our working directory contains a file corresponding to each source object in `/vesta/proj/text.4`. The package has three source objects—two files named in the `DIRECTORY` and the system model itself—so `/udir/levin/text` will contain three files: `Text.def`, `Text.mod`, and `MODEL`. These are ordinary, mutable, Unix files. This package doesn't have any submodels, but if it did, the Checkout operation would construct subdirectories of the working directory for each submodel in the natural way.

So far, only the public repository and working directory have been involved. Now suppose that we alter `/udir/levin/text/Text.mod`. We use an ordinary Unix text editor to make this change; no special Vesta facilities are involved. Next we want to evaluate a Vesta model that resembles `MODEL`, but which refers to our freshly edited file. Accordingly, we invoke the Vesta *Advance* operation, which performs the following actions:

- For each file `F` in the working directory that has changed, copy `F`'s contents to the private repository and assign it a new unique identifier (UID), then alter the `MODEL` file so that the `DIRECTORY` clause binds the name `F` to the new UID.
- Copy the contents of the altered `MODEL` file to the private repository and assign it a new unique identifier (UID), then create a new package name in the private repository and bind it to this UID.

In this case, the result of the Advance operation is the new package named `/vesta/levin/text.5.checkout.1`, which appears in figure 4. Notice that a new UID has been assigned to `Text.mod`, reflecting our change to the working directory file of the same name.

In short, Advance creates a new (immutable) package version in the private repository whose content matches the working directory. Naturally, Advance also copes with file creations and deletions in the working directory, edits to the `MODEL` file itself, and changes to subdirectories. For example, if we edited the file `/udir/levin/text/Text.mod` again, created a new file named `/udir/levin/text/Text.doc`, and then invoked Advance again, we would get a new package version named `/vesta/levin/text.5.checkout.2` with the system model shown in figure 5.

```
DIRECTORY
    Text.def = uid-1,
    Text.mod = uid-3
IN  {
    the building part; not relevant just now
    }
```

Figure 4: System model following the Advance operation

```
DIRECTORY
    Text.def = uid-1,
    Text.mod = uid-5,
    Text.doc = uid-6
IN  {
    the building part; not relevant just now
    }
```

Figure 5: System model after a second Advance operation

We can ask Vesta to perform the *Evaluate* operation on any package version in a repository, for example, on `/vesta/levin/text.5.checkout.1` or `/vesta/levin/text.5.checkout.2`. Since these are immutable "snapshots", no subsequent changes to the working directory can alter their contents. Most commonly, we perform alternating Advance and Evaluate operations, first creating a new immutable version, then attempting to build the derived files that it describes. Vesta provides a combined *Advance-and-Evaluate* operation for this common case.

Now let's imagine that we've finished our changes and have successfully evaluated the system model in figure 5 to build and test it. We're ready to end our checkout session, so we invoke the *Checkin* operation and specify

- the public repository name previously reserved (`/vesta/proj/text.5`), and
- the version in the private repository that is to be bound to the reserved name (`/vesta/levin/text.5.checkout.2`).

Notice that the working directory plays no part in the Checkin operation. Checkin binds the reserved package name `/vesta/proj/text.5` to the same UID as `/vesta/levin/text.5.checkout.2`. Also, it copies to the public repository all the source objects created as part of this session whose UIDs appear in the checked-in system model (that is, *uid-5* and *uid-6*). The intermediate versions we created in the private repository still exist, should we need to go back and look at them, but they don't clutter up the public repository, which many users share[5].

Before leaving this subject, we need to consider one other way in which system models are frequently modified. Models commonly import other models, and, as we have seen, these imports specify particular versions. Often a new version of a package must be created solely to reference one or more new versions of imported packages. That is, in this new system model, the only change will be in the import clause.

Just as the Advance operation performs a specialized editing operation on the directory clause of a system model, so the *Revise-imports* operation selectively modifies the import clause. Of course, we could modify the import clause with an ordinary text editor if we wished, but Revise-imports automates some tedious aspects of the editing process. The automation is provided by a set of user-supplied rewriting rules, which can be predicated on various properties of packages, including version number, date of checkout or checkin, author, package name, etc. The details aren't particularly relevant here; what matters is:

- The import clause changes only because of an explicit, user-invoked action, so the environment in which a package is developed (which is provided by the set of imports, as we will see) is stable by default. No blunder by another user can "sneak in" unnoticed; each user explicitly decides when to tolerate an environment change.
- If a user wants to accommodate frequent environmental changes (e.g., because he is part of a group that is cooperatively developing related packages), tools like Revise-imports can ease the process of updating the explicit version in import clauses.

Contrast the Vesta approach with the standard Unix tools, in which connections between components are frequently represented weakly (e.g., by search paths through the file system). No immutability guarantees exist, and the user cannot prevent changes from occurring in the components that his package imports. Consequently, reproducible construction is impossible to assure. Vesta, on the other hand, makes all connections immutable, so reproducibility is ensured. Only by constructing new versions through explicit user actions can change be introduced.

3 Basic Vesta system models

Our methodology for defining packages groups them into three major classes: *applications*, *libraries*, and *umbrellas*.

[5]For pragmatic reasons, Vesta does provide a way to recover the disk space consumed by these private versions, so, strictly speaking, they aren't immutable. However, Vesta prevents the names from ever being reused, so there is no possibility of subsequent confusion with other versions.

- An *application package* provides one or more executable programs that can be invoked directly by a user, typically from a shell. An interesting special case is an application that is used for program construction, such as a preprocessor or stub generator.

- A *library package* provides at least one (often more than one) interface and at least one implementation of each interface. An important sub-case is a client/server library package, in which the implementation is divided into two parts, a client library and a server program, and the interfaces are sometimes grouped into two or more sets as well, reflecting different kinds of clients (e.g., ordinary client programs, server administration tools).

- An *umbrella package* groups a set of related packages (generally libraries, applications, or other umbrellas), in some systematic way. That is, an umbrella usually defines how a collection of system components are *integrated* into a larger, consistent group. Thus, an umbrella typically provides a convenient environment for constructing and testing the components together. Two important umbrellas deserve mention, namely `building-env` and `release`. `building-env` brings together and constructs a consistent collection of tools, libraries, interfaces, and default option settings. `release` builds a large collection of application programs in a single construction environment, which is provided by a version of `building-env`. We'll look at these two umbrellas in more detail in section 5.

In this section, we examine application and library package models; in section 4, we see how umbrella models aggregate them.

3.1 Application packages

An application package provides one or more executable programs that a user can invoke directly, typically from a shell. Thus, the building part of an application package's system model will typically invoke a compiler one or more times to produce Vesta derived objects that correspond to conventional binary object files, then combine these derived objects into one or more executables. The running example of section 2 developed parts of a system model for a hypothetical application package; let's now switch to a real one. Refer to figure 6.

Although this system model is more complicated than our hypothetical example, its broad outlines are the same. The body of the `build` function (line 8) constructs a single executable (a program called `Hanoi`) by compiling a Modula-2+ main program (`Hanoi.mod`) and linking it with a standard shared library (`SL-ui`). Line 10 defines a roughly analogous function (`doc`) that constructs the documentation using a document compiler. Both `build` and `doc` are invoked from the `test` component, which establishes a suitable environment via the imported `building-env` package (line 12).

```
01  DIRECTORY
02      Hanoi.mod = uid,
03      Hanoi.mss = uid
04  IMPORT
05      building-env.v (*/vesta/proj/building-env.22*) = uid
06  IN  {
07      build = FUNCTION ... IN
08          LET M2 IN { Hanoi = Prog( (Compile(Hanoi.mod), SL-ui) ) },
09
10      doc = FUNCTION ... IN { Hanoi.doc = Scribe$Compile(Hanoi.mss) },
11
12      test = LET building-env.v$Env-default IN {
13          build(),
14          doc()
15          }
16      }
```

Figure 6: System model for an application package

Let's relate this system model to the three methodological properties listed in section 1.

- *Modularity.* The description of the environment needed to build `Hanoi` doesn't appear directly in the system model. Instead, a version of the `building-env` umbrella is explicitly imported on line 5. So, the `Hanoi` system model, while complete and precise, is nevertheless quite compact, since the details of the construction environment are packaged elsewhere.

- *Isolation of construction algorithm and environment.* The construction algorithm is embodied in the functions on lines 7–10, and doesn't refer to a specific version of the construction environment. Instead, the items needed from the environment are (implicit) formal parameters to the function. The specific environment provided by the imported `building-env` model is used only in the `test` component on line 12.

- *Extensive parameterization.* We see that `build` and `doc` are parameterized in the same way as the example of figure 2. They are invoked from the `test` component for individual testing purposes. However, they can also be invoked from other system models. In particular, the `release` model, which builds all the applications using a single `building-env`, invokes these functions. Notice that the isolation of the construction algorithm and environment makes this possible.

It's also worth noting how extensive use of defaulting reduces the size of the environment's description. As a result, even though many name scopes are quite large (e.g., the `Env-default` binding contains over 1000 names), the system model is short and contains only essential material. However, this brevity comes at a price; we can't easily discover by perusing the source text of the model what names are defined in a scope and where their associated values came from. Vesta must provide tools (such as browsers) to help users answer these questions conveniently.

3.2 Library packages

SRC has several hundred library packages, each of which provides one or more Modula-2+ interfaces and at least one implementation of each interface. A single module frequently implements more than one interface and, of course, libraries often depend on other libraries. This interdependency immediately raises a structuring issue: how best to group this large collection of named pieces for the convenience of the application packages that need to use them.

There is no single correct answer, but we adopted an approach that works reasonably well, given the size of our system. Our libraries export about 1500 compiled interfaces, and we put them all into a flat name space. We group the implementations of those interfaces into a small set of libraries (about 10) that represent major abstraction layers, chosen so that most applications need only mention a single one. The names of these libraries conventionally begin with `SL-` (e.g., `SL-ui`) and are added to the same name space with the compiled interfaces. This name space forms the bulk of the `Env-default` binding provided by the `building-env` model.

To make it straightforward for the `building-env` umbrella to assemble this name space, each library package supplies two functions, named `intfs` and `impls`, which the `building-env` system model invokes. Consider the library package named `quadedge` (figure 7), which is part of a set of graphics facilities in the standard library `SL-ui`.

The two functions `intfs` and `impls` simply compile the package interfaces and implementations, respectively, and return bindings containing the results. The `test` component invokes a function in a submodel that does the actual work of testing (figure 8). Of course, there's no requirement to isolate the testing code in a submodel, but doing so conveniently separates the construction algorithms and environment, especially when the testing portion of the system model is substantial.

Evidently, there's more going on here than in the `Hanoi` example. Let's work top-down. The `test` component of the system model (figure 7, line 16) invokes the function `do-tests` (figure 8, lines 30–36) in the submodel to perform the actual testing. This function's body invokes the function `do-a-test` twice with different parameters; each parameter is a source object defined in the `DIRECTORY` clause (lines 20–21). Looking at line 25, we discover that the parameter has the suggestive name `test-script`, so we can reasonably expect it to contain the input that drives some sort of testing program.

```
DIRECTORY
    OctEdge.def  = uid,
    OctEdge.mod  = uid,
    QuadEdge.def = uid,
    QuadEdge.mod = uid,
    tests        = uid,
IN  {
    intfs = FUNCTION ... IN
        LET M2 IN {
            Compile(OctEdge.def),
            Compile(QuadEdge.def) },
    impls = FUNCTION ... IN
        LET M2 IN {
            Compile(OctEdge.mod),
            Compile(QuadEdge.mod) };
    test = tests$do-tests(),
    }
```

Figure 7: System model for a library package

Next, let's look at the body of `do-a-test`. On line 27, we see the construction of an executable named `Tester`; it closely resembles the construction of the `Hanoi` program. `Tester` is then used on line 28, where we see a call of the function `Shell$Sh`.

`Shell$Sh` provides carefully-regulated access to the Unix shell. `Shell$Sh` treats its argument as a shell script and executes it in a restricted Unix environment. In effect, the script can access files in only a single directory, and `Shell$Sh` arranges that the names in that directory correspond to the set of Vesta names in the scope at the point of its invocation[6]. In this case, the shell script is a rather simple one; it executes a single program named `Tester` and gives it a file named `test-script` as its (standard) input. Both of these names must therefore be in the environment in which `Shell$Sh` is invoked, and we see that they are. `Tester` is bound on line 27, and `test-script` is a formal parameter (line 25) whose value is bound when the function is applied (lines 35 and 36).

To summarize so far, we see that the submodel builds a custom test program (`Tester`) and runs it with inputs that are Vesta source objects mentioned by the

[6]This limitation prevents arbitrary side-effects, which would compromise Vesta's functional language, and helps ensure reproducibility. The shell script is allowed to read and write its artificially created working directory or any subdirectory. `Shell$Sh` extracts any files that the script writes into a binding, whose names are the file names, and returns the binding as the result of the function application. In this example, we don't care about the result value, for reasons that will become clear shortly.

```
18  DIRECTORY
19      Tester.mod          = uid,
20      OctEdgeTest.testin  = uid,
21      QuadEdgeTest.testin = uid,
22  IMPORT
23      building-env.v (* /vesta/proj/building-env.23 *) = uid,
24  IN  {
25      do-a-test = FUNCTION test-script, ... IN
26          LET {
27              Tester = LET M2 IN Prog( (Compile(Tester.mod), SL-ui) ) }
28          IN  Shell$Sh("Tester < test-script");
29
30      do-tests = FUNCTION ... IN
31          LET {
32              building-env.v$Env-build-with-overrides(
33                  { quadedge = {intfs = intfs, impls = impls} } ) }
34          IN  {
35              do-a-test(OctEdgeTest.testin),
36              do-a-test(QuadEdgeTest.testin) },
37      }
```

Figure 8: System model for the `tests` submodel of `quadedge`

submodel's `DIRECTORY` clause. Let's now look more closely at the construction of `Tester` (line 27). It appears very similar to the body of the `build` function of an application model; a main program is compiled and linked with a shared library (`SL-ui`). But how is the environment for this construction established? Look at lines 32–33, which invoke the function `Env-build-with-overrides` in the imported `building-env` package. If this were `Env-default`, it would be familiar; we saw this binding used as the environment for the testing of the `Hanoi` program. But here the situation is more complicated, because we want our test program to be linked not with a standard version of `SL-ui`, but with one that includes the version of `quadedge` that we want to test. Lines 32–33 construct a custom environment containing this non-standard `SL-ui` for the use of the test program. The ability to do this entirely from the system model of the `quadedge` package is an essential and powerful feature of Vesta, and it's important that we understand how it works.

Each version of the `building-env` model imports particular versions of each library package, which it uses to construct the environment it supplies. So, in particular, the version of `building-env` imported by the `quadedge` package for testing purposes imports specific versions of library packages. Recall that as part of the construction algorithm for the `Env-default` binding, the `building-env` system model invokes the `intfs` and `impls` functions from each such package. Among these packages is a version of the `quadedge` package, which we noted was part of the `SL-ui` library. But this version isn't what we want to test—it is an older, stabler version, and we want to substitute the current version we are working on. That's what the function `Env-build-with-overrides` does. It accepts a parameter that is used to override selected library packages. Look at line 33. The parameter is a binding containing a single element named `quadedge`. This name matches one of the library package names that the `building-env` model knows about. The `Env-build-with-overrides` function therefore constructs an environment using the value we have bound to `quadedge` instead of the `building-env` model's wired-in value. Since `Env-build-with-overrides` expects the wired-in value to supply two functions (`intfs` and `impls`), the value bound to `quadedge` on line 33 must be a binding that includes those two names—and we see that indeed it is. Finally, look at the values bound to those names. They are the values of the self-same identifiers in the current scope. By the defaulting rules, we can determine that these names were bound in the main system model for the `quadedge` package (figure 7), and passed as implicit formal parameters through the call of `test$do-tests()`.

`Env-build-with-overrides` is a bit subtle, but the ability to override packages in the standard environment without modifying the `building-env` model provides great power and convenience. The developer of `quadedge` can change the package and test the change in a consistent environment by altering only the `quadedge` model. Note that the test program might invoke other interfaces in `SL-ui` whose implementations use the `QuadEdge` interface. If the `QuadEdge` interface in this version of the package differs from the standard one (i.e., the version wired into `building-env`), then the appropriate modules of those other packages that are incorporated in `SL-ui` by the `building-env` model will be automatically recompiled. If the interface differences are source-compatible, the recompilations will succeed and the resulting `SL-ui` library will be the same as if the new `quadedge` package had been wired into the `building-env` model. It's important to understand that this is *not* the same as simply substituting the implementation modules of the new `quadedge` package for the ones in the `SL-ui` library included in `Env-default`. Such a simple substitution doesn't guarantee to produce a consistently compiled result; using `Env-build-with-overrides` does. In fact, a small change to the `QuadEdge` interface might produce substantial recompilation of library modules in many packages, if the interface is widely used.

4 Umbrella packages

An umbrella package groups a set of related packages (libraries, applications, and/or other umbrellas) in some systematic way, and provides a convenient environment for constructing them. An umbrella commonly produces an executable and associated documentation, much like an application package. This occurs when a complex application program (e.g., a compiler, or sophisticated text editor) is built from several smaller pieces that are developed in parallel by several programmers. Since a package is Vesta's unit of individual work, it is natural to divide a large application into several packages with an umbrella package to bring them together and build the application.

In the preceding sections, we saw that application and library packages can be built both for testing purposes and as part of an umbrella (like **`release`** or **`building-env`**). An umbrella package must serve the same two constituencies (its own testing and higher-level umbrellas) plus an additional one: its component packages. We want a component package "under" an umbrella to be testable without modification of the umbrella; hence the umbrella package itself must supply facilities that enable unit testing of the component.

In fact, we saw an example of this in the previous section. The **`building-env`** umbrella provided a function, **`Env-build-with-overrides`**, that enabled one of the component package models, **`quadedge`**, to perform unit testing in a consistent environment that included its own interfaces and implementations. This paradigm—a construction function that permits selective overrides—is common in Vesta umbrellas. We've seen how it is used by the component packages of an umbrella. Now let's look in more detail at the umbrella itself.

For our example, we'll use the umbrella package that constructs the core of the Vesta system itself. (Naturally, Vesta is built using Vesta!) First, we must understand Vesta's overall implementation structure. The bulk of the implementation resides in a server, but client programs access Vesta's core facilities through a collection of interfaces implemented by a library. This client library and the server communicate using remote procedure calls (RPC). Consequently, the (umbrella) system model for the **`vesta`** package must build two bundles of code—the server and the client library—plus the interfaces used by client programs.

We will look at the consequences of this client/server division shortly, but initially we'll focus on the relationship between the umbrella package and the components it imports. To emphasize this aspect of the umbrella model, we present it first omitting everything that relates to the server construction (figure 9).

By definition, an umbrella package groups a set of related packages. We see these on lines 4–6, which import the three component packages (named **`vesta-base.v`**, **`vesta-eval.v`**, and **`vesta-rep.v`**) that together provide the Vesta facilities. Because of the client/server division, each of these component packages provides two pairs of functions. One of these pairs is already familiar to us—it's the customary **`intfs`** and **`impls`** functions that we saw in section 3.2, which are used to build a client library. The other pair is for constructing the server, which we'll look at shortly.

The functions **`build-intfs`** and **`build-impls`**, defined on lines 14–21, appear to collect the interfaces and implementations of the component packages

```
DIRECTORY
IMPORT
    vesta-base.v (*/vesta/proj/vesta-base.9*)      = uid,
    vesta-eval.v (*/vesta/proj/vesta-eval.36*)     = uid,
    vesta-rep.v (*/vesta/proj/vesta-rep.29*)       = uid,
    building-env.v (*/vesta/proj/building-env.72*) = uid,
IN  {
    LET {
        default-pkgs = {
            vesta-base = vesta-base.v,
            vesta-eval = vesta-eval.v,
            vesta-rep = vesta-rep.v },
        build-intfs = FUNCTION ... IN {
            vesta-base$intfs();
            vesta-rep$intfs();
            vesta-eval$intfs() },
        build-impls = FUNCTION ... IN {
            vesta-base$impls(),
            vesta-rep$impls(),
            vesta-eval$impls() },
        }
    IN  {
        intfs-with-overrides = FUNCTION pkg-overrides, ... IN
            LET { default-pkgs, pkg-overrides } IN build-intfs(),
        impls-with-overrides = FUNCTION pkg-overrides, ... IN
            LET { default-pkgs, pkg-overrides } IN build-impls();

        intfs = FUNCTION ... IN intfs-with-overrides({}),
        impls = FUNCTION ... IN impls-with-overrides({}),

        };

    build-client = FUNCTION ... IN LET intfs() IN impls();

    test =
        LET building-env.v$Env-default IN {
            build-client();
            },
    }
```

Figure 9: Client-related parts of the `vesta` system model

and build them in the obvious way. However, because we want the umbrella to permit these functions to be overridden, as we did with the `quadedge` package in section 3.2, these functions actually do something slightly subtle. Look carefully at the identifiers used to refer to the component packages in the bodies of these functions. They are `vesta-base`, `vesta-eval`, and `vesta-rep`. By default, these are bound to the imported package versions named `vesta-base.v`, `vesta-eval.v`, and `vesta-rep.v`, respectively, but that default may be overridden. Let's see how.

First, lines 10–13 establish the default package names by constructing a binding named `default-pkgs`. Second, lines 33 and 35 merge this binding with a second one named `pkg-overrides`, and invoke `build-intfs` and `build-impls` in an environment containing the result. So, the functions `intfs-with-overrides` and `impls-with-overrides` resemble the function `Env-build-with-overrides` that we encountered earlier—they construct something using a set of wired-in default package versions, but permit selective overriding through a parameter.

Notice the two functions defined on lines 37–38, which build the default versions by invoking the two more general functions with an empty binding as the set of packages to override. Note also the `build-client` function and the `test` component (lines 53–60), which simply arrange to compile the default versions of the interfaces and implementations.

This paradigm recurs in most umbrellas. A set of component packages appear in the `IMPORT` clause, each bound to a name of the form `foo.v`. A binding named `default-pkgs` binds `foo` to `foo.v` for each such import. A `build-with-overrides` function taking a parameter `pkg-overrides` has as its body a `LET` expression in which the binding {`default-pkgs,pkg-overrides`} is opened. Finally, all the real work of construction uses a name of the form `foo`, not `foo.v`, to refer to a component package. For convenience, it is also customary to provide a parameterless function that invokes the `build-with-overrides` function with an empty binding as the parameter.

Now that we've seen how the overriding machinery works, let's flesh out the system model above to include the server-related portions. Refer to figures 10 and 11. The major new portions are lines 22–29 and 40–50. We now can see that the second pair of functions provided by each component package builds a collection of interfaces and their implementations for inclusion in the server. Thus, lines 22–29 are analogous to lines 14–21; they simply build this second set of interfaces and implementations. Lines 40–50 construct the server itself, and most of this should now be familiar. Line 42 handles the possibility of overrides, and lines 45–50 construct an executable program, much as we saw in sections 3.1 and 3.2. (We'll return to the new wrinkles on lines 46 and 50 in a little while.) The server evidently consists of a main program `VestaDriver.mod`, the pieces taken from the three component packages, and a standard library `SL-basics`. The `test` component builds the server (line 59) in a standard environment imported explicitly for the purpose (lines 7 and 57), using a convenient function

(line 54) that invokes `server-with-overrides` with an empty set of overrides.
Now let's look at an outline of the system model for one of the component packages, say `vesta-eval`, to see how it would exploit the `vesta` umbrella's override feature for testing purposes. The model appears in figure 12. Lines 1–19 resemble things we've already seen. Lines 20–25 construct the parameter that will be used to override the parts of the client and server contributed by the `vesta-eval` package. Lines 27–33 do the real work. Line 28 builds the server, overriding the `vesta-eval` component. Lines 29–33 build a test program using the Vesta client library constructed with the `vesta-eval` portion

```
01  DIRECTORY
02      VestaDriver.mod = uid,
03  IMPORT
04      vesta-base.v (*/vesta/proj/vesta-base.9*)      = uid,
05      vesta-eval.v (*/vesta/proj/vesta-eval.36*)     = uid,
06      vesta-rep.v (*/vesta/proj/vesta-rep.29*)       = uid,
07      building-env.v (*/vesta/proj/building-env.72*) = uid,
08  IN  {
09      LET {
10          default-pkgs = {
11              vesta-base = vesta-base.v,
12              vesta-eval = vesta-eval.v,
13              vesta-rep = vesta-rep.v },
14          build-intfs = FUNCTION ... IN {
15              vesta-base$intfs();
16              vesta-rep$intfs();
17              vesta-eval$intfs() },
18          build-impls = FUNCTION ... IN {
19              vesta-base$impls(),
20              vesta-rep$impls(),
21              vesta-eval$impls() },
22          build-server-intfs = FUNCTION ... IN {
23              vesta-base$server-intfs();
24              vesta-eval$server-intfs();
25              vesta-rep$server-intfs() },
26          build-server-impls = FUNCTION ... IN {
27              vesta-base$server-impls(),
28              vesta-eval$server-impls(),
29              vesta-rep$server-impls() },
30          }
```

Figure 10: Complete `vesta` system model (part 1)

```
31      IN  {
32          intfs-with-overrides = FUNCTION pkg-overrides, ... IN
33              LET { default-pkgs, pkg-overrides } IN build-intfs(),
34          impls-with-overrides = FUNCTION pkg-overrides, ... IN
35              LET { default-pkgs, pkg-overrides } IN build-impls();
36
37          intfs = FUNCTION ... IN intfs-with-overrides({}),
38          impls = FUNCTION ... IN impls-with-overrides({});
39
40          server-with-overrides = FUNCTION pkg-overrides, ... IN
41              LET {
42                  default-pkgs, pkg-overrides;
43                  build-intfs(); build-server-intfs() }
44              IN  {
45                  vesta = M2$Prog {
46                      pieces = (
47                          M2$Compile(VestaDriver.mod),
48                          build-server-impls(),
49                          SL-basics ),
50                      M2-enable-gc = TRUE } },
51          };
52
53      build-client = FUNCTION ... IN LET intfs() IN impls(),
54      build-server = FUNCTION ... IN server-with-overrides({}),
55
56      test =
57          LET building-env.v$Env-default IN {
58              build-client(),
59              build-server()
60              },
61      }
```

Figure 11: Complete `vesta` system model (part 2)

overridden by the local `intfs` and `impls`. The result, then, of evaluating the `test` component is a binding containing two executables, `server` and `tester`, built to be consistent with each other and using the locally-defined pieces of `vesta-eval`. We observe the same property that we noted with the `quadedge` package—a new version of a component package can be built and tested without modifying the umbrella package that includes an older version of the component.

We conclude our examination of the `vesta` umbrella by returning to a detail of the construction of the server on lines 45–50 of figure 11. Notice that the `Prog`

```
DIRECTORY
    EvalTester.mod = uid,
    entries for all vesta-eval interface and implementation modules
IMPORT
    vesta.v (*/vesta/proj/vesta.47*)                  = uid,
    building-env.v (*/vesta/proj/building-env.71*) = uid
IN {
    intfs = FUNCTION ... IN
        { compile all client interfaces },

    impls = FUNCTION ... IN
        { compile implementations of client interfaces },

    server-intfs = FUNCTION ... IN
        { compile interfaces for use with the server only },

    server-impls = FUNCTION ... IN
        { compile server implementation modules };

    override = {
        vesta-eval = {
            intfs        = intfs,
            impls        = impls,
            server-intfs = server-intfs,
            server-impls = server-impls } };

    test = LET building-env.v$Env-default IN {
        server = vesta.v$server-with-overrides(override),
        LET vesta.v$intfs-with-overrides(override) IN {
            tester = M2$Prog( (
                M2$Compile(EvalTester.mod),
                vesta.v$impls-with-overrides(override),
                SL-basics) ) } }
    }
```

Figure 12: System model of the `vesta-eval` package

function is passed a binding of two values named `pieces` and `M2-enable-gc`. This illustrates how a normally defaulted parameter is overridden. The function `Prog` actually has a long list of parameters that control the linking process. In this case, we want to supply (non-default) values for two of them: `pieces`, the list of things to be linked together, and `M2-enable-gc`, a Boolean switch. We could use a positional list if we knew these were the first two parameters, but instead we use a binding to exploit Vesta's parameter defaulting rules (recall section 2.2) and achieve the approximate effect of keyword parameters.

5 The `release` and `building-env` umbrellas

We've seen that an umbrella model constructs a software subsystem by combining lower-level components. An umbrella model also offers a selective-override feature, which the component models use to build testing versions of the subsystem. In this section, we'll look at two substantial umbrella models that exploit these structuring conventions. These models also illustrate how our description methodology scales naturally to accommodate large systems.

5.1 The `release` umbrella

Umbrellas may import other umbrellas. For example, the `release` model, which builds a large collection of application programs using a single, consistent `building-env`, imports the `vesta` model and invokes its `build-server` function. As we have seen, the `vesta` model is a rather small umbrella package; the `release` model is considerably larger, but not conceptually more difficult to understand. Refer to figure 13.

```
DIRECTORY
IMPORT
    building-env.v (*/vesta/proj/building-env.76*) = uid,
    bootfile.v (*/vesta/proj/bootfile.20*)         = uid,
    bitmapeditor.v (*/vesta/proj/bitmapeditor.8*)  = uid,
    copy.v (*/vesta/proj/copy.5*)                  = uid,
    vesta.v (*/vesta/proj/vesta.55*)               = uid,
    119 additional imported packages...
IN  {
    LET building-env.v$Env-default IN {
        bootfile.v$build(),
        bitmapeditor.v$build(),
        copy.v$build(),
        vesta.v$build-server(),
        119 additional similar lines...
        }
    }
```

Figure 13: The `release` model

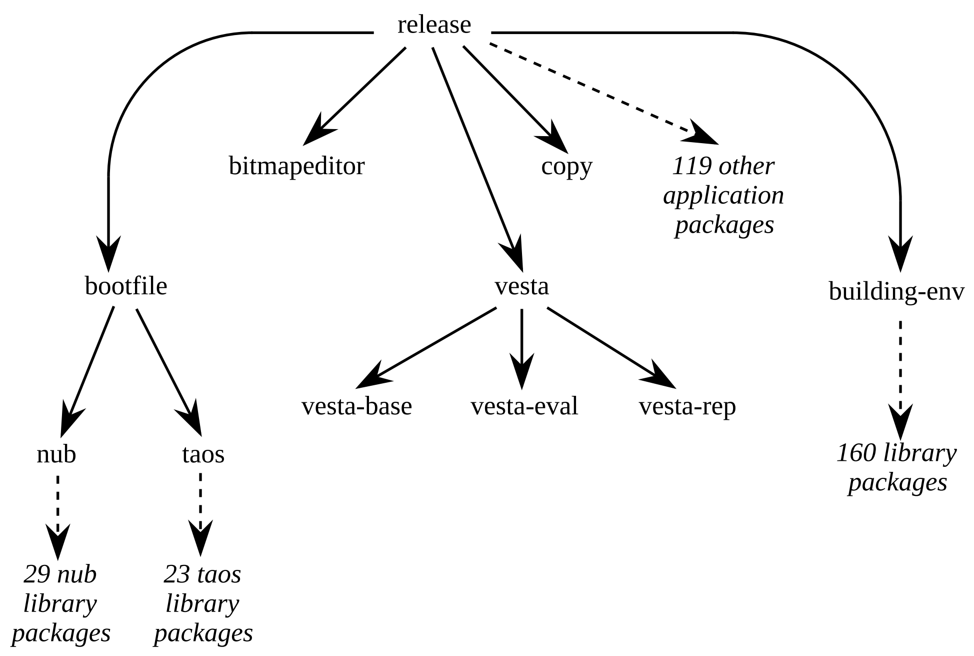

Figure 14: Import graph of the `release` system model

Note that a building function is invoked in each of the imported models. Some of the imported packages, like `bitmapeditor` and `copy`, are simple applications of the kind we examined in section 3.1. Others, like `vesta`, are themselves umbrellas. Figure 14 shows the import structure of the `release` model and some of its more interesting imported components.

Of particular interest in this picture is the `bootfile` model, which builds the operating system image for SRC's Topaz system [McJones and Swart]. This operating system has a micro-kernel structure; it includes a small kernel (called the Nub) and a larger server program (called Taos) that executes much as an ordinary application would. The `bootfile` model reflects this structure; it is an umbrella that imports two other umbrellas, `nub` and `taos`, which build the kernel and operating system server from individual components.

The `release` model is our most comprehensive umbrella, in the sense that it forms the apex of our entire software pyramid.

5.2 The `building-env` umbrella

We have already encountered some facilities of the `building-env` model, notably the `Env-default` binding and `Env-build-with-overrides` function. The `building-env` model has additional, unique characteristics that we need to investigate further.

Until now, we've ignored how particular versions of construction tools, such as the compilers and linkers, were specified. We mentioned that `M2` is a binding containing the `Compile` and `Prog` functions, but we didn't say where these functions came from and how the binding was constructed.

We don't simply treat the tools as binaries. Requirement 3 of section 1 compels us to construct the environment, including the tools, from complete and precise descriptions in terms of source objects. We therefore define the tools as we would any other program—by writing the Vesta description that expresses how they are constructed. So, when we look into the `building-env` model, we find the function shown in figure 15.

```
build-bridges = FUNCTION ... IN {
    AS    = BRIDGE( as-bridge$build() ),
    C     = BRIDGE( c-bridge$build() ),
    M2    = BRIDGE( m2-bridge$build() ),
    Shell = BRIDGE( shell-bridge$build() ),
    }
```

Figure 15: Bridges built by the `building-env` system model

`as-bridge`, `c-bridge`, `m2-bridge`, and `shell-bridge` are packages imported by `building-env`. (Vesta uses the term *bridge* for a group of related tools that support a particular programming language [Lampson and Schmidt].) Each is essentially an application package, and therefore has a `build` function that builds an executable[7]. The `build-bridges` function constructs each executable and passes it to the built-in Vesta primitive `BRIDGE`, which launches the bridge executable, then obtains (via an RPC call), a Vesta binding of the functions it implements. This binding is returned as the result of the `BRIDGE` function[8].

By the way, bridges can be overridden just like any other package. Consequently, the testing portion of the system model for `m2-bridge` can use the function `Env-build-with-overrides` to build an entire consistent set of libraries and tools using, say, new versions of the compiler and linker. The contents of that environment can be compared with another version of the environment, if

[7] Of course, our models can accommodate a tool for which only the binary form is available. Construction from source is preferable, because the construction can be repeated in a slightly different environment if necessary. But if only the binary is available, the `build` function simply returns the binary directly as its result; that is, the binary is a source object in the Vesta sense (see section 2.1).

[8] The workings of Vesta bridges fall outside the scope of this paper; see [Brown and Ellis] for a complete discussion.

desired, as part of a regression test. Furthermore, the entire comparison can be included in the `test` component of the bridge model.

Since the `build` function of each bridge package builds an executable, it must have a suitable construction environment. Furthermore, as we saw in section 3.1, the invoker of the `build` function is responsible for defining that environment. Thus, the `building-env` model, in which the `build-bridges` function appears and is invoked, must itself supply the environment for constructing the bridges. This environment is delivered by a pre-existing `building-env`, that is, an earlier version of the `building-env` package. Thus, a `building-env` model imports another `building-env` model for building the bridges.

Keeping these two construction environments separate is essential, particularly during cross-platform software development. The bridges execute tools that run on the *host* platform—the one on which the development is occurring. The software being built by a Vesta model is intended to run on a *target* platform, which may have a different instruction set, or operating system, or both, from the host. The `building-env` model must keep these two environments separated, or chaos will result. We'll see how the separation is achieved shortly.

An importer of `building-env` may need to use both environments. For most situations, the target environment is all that is necessary, and it is this environment that the `Env-default` binding and `Env-build-with-overrides` function construct. Recall that this environment contains some facilities (e.g., the bridges) that have been built to execute in the host environment and produce results for the target environment. Some software packages require the use of specialized tools that are not part of `building-env`, and hence must be able to obtain an appropriate host environment in order to specify the construction of the tool. Let's look at an example.

In figure 16 we've altered the `Hanoi` example of section 3.1 by eliminating the source file `Hanoi.mod` and introducing instead a computation that produces a Modula-2+ program by applying a private preprocessor (`my-pp`) to a source file named `Hanoi.pp-in`. Note the change in line 16 to invoke `apply-pp`, which in turn invokes `build-pp`. Line 9 constructs the preprocessor in a fashion that is, by now, quite familiar. However, we're interested in how the environment for this construction is established, which happens on lines 7 and 8.

The environment established by the `test` component is, as usual, the target environment; that is, it contains suitable interfaces and libraries for the environment in which the software being constructed will ultimately execute. Buried in the `Env-default` binding, however, is another binding that is the imported `building-env` used to construct the bridges. This binding is called `Env-for-bridges` to emphasize its role; it is an environment suitable for construction of software destined to execute *as part of the Vesta evaluation* of `Hanoi`. In other words, it is the host environment. Consequently, on line 8, we see the `Env-for-bridges$Env-default` binding being opened as the environment for the tool construction on line 9.

Although this example illustrates how the separate host and target environ-

```
DIRECTORY
    Hanoi.pp-in = uid,
    SpecialPP.mod = uid,
IMPORT
    building-env.v (*/vesta/proj/building-env.22*) = uid
IN  {
    build-pp = FUNCTION Env-for-bridges IN
        LET Env-for-bridges$Env-default IN {
            my-pp = M2$Prog( (M2$Compile(SpecialPP.mod),SL-basics) ) };

    apply-pp = FUNCTION source, ... IN
        LET build-pp() IN Shell$Sh("my-pp source")))$stdout;

    build = FUNCTION ... IN
        LET M2 IN {
            Hanoi = Prog( (Compile(apply-pp(Hanoi.pp-in)), SL-ui) ) };

    test = LET building-env.v$Env-default IN build(),
    }
```

Figure 16: Tool construction inside an application's system model

ments are used in a Vesta model, we can extract some other lessons. First, the **Hanoi** model got only a few lines longer when we added the preprocessor complexity, yet it still *completely* captures what's going on. Second, the environment of tool construction is available to any package that needs it. The tools don't have to be placed in the **building-env** model; private tools like **my-pp** are easily introduced in suitably smaller scopes. Third, it isn't necessary to construct a full-fledged Vesta bridge package in order to execute a simple tool. The **Shell** bridge can be used for this purpose, just as it was used for testing in figure 8. (Incidentally, if we look back at that example, we see that it ignores the distinction between the host and target environments. It constructs the test program in the target environment, then executes it with the **Shell$Sh** function, implicitly assuming that the host and target platforms are the same. This is indeed true for the particular version of **building-env** that it imports, so the model isn't buggy, but a similar model that differs only in the version of **building-env** it imports could easily fail to work.)

Let's return now to the **building-env** model to see how it keeps the host and target environments separate. We've already seen three pieces of this model: the **Env-default** binding, the **Env-build-with-overrides** function, and the **Env-for-bridges** binding that is part of **Env-default**. Now let's consider the model as a whole (figures 17 and 18).

```
01  DIRECTORY
02  IMPORT
03      building-env.v = uid,   (* host environment *)
04      (* bridges *)
05      m2-bridge.v = uid,
06      as-bridge.v = uid,
07      etc.
08      (* library packages *)
09      text.v = uid,
10      quadedge.v = uid,
11      etc.
12  IN  {
```

Figure 17: The `building-env` system model (part 1)

Once again, we see the principle of extensive parameterization at work. The heart of the `building-env` model is the function `Env-build`, which constructs an environment for a specified target architecture and operating system while allowing selective overrides of component packages. `Env-default` and `Env-for-bridges$Env-default` provide convenient access to common special cases, in which some of these parameters are defaulted. But `Env-build` provides the full generality.

Let's exploit this generality. Referring to lines 14–15 and lines 46–47, we see that the `Env-default` binding constructed on line 50 is targeted for a VAX running Ultrix. Suppose we needed to build this target environment on an Alpha AXP running OSF/1. If the (host) environment provided by `Env-for-bridges$Env-default` isn't suitable, we simply replace line 34 with

```
34a  LET Env-for-bridges$Env-build({}, "AXP", "OSF1") IN
34b      build-bridges();
```

Observe how Vesta's ability to describe environments completely and encapsulate them with bindings makes this a tiny change to the source text of the model, even though it has a dramatic effect on the actions required to construct the system.

5.3 Software distribution with Vesta

We've seen that Vesta models emphasize complete description in terms of source objects. Derived objects are managed automatically by Vesta, which is why they don't have user-sensible names (see sections 2.1 and 2.5). But sometimes users have to deal with derived objects in a more explicit way. An obvious example is the distribution of a software release.

```
LET {
    default-inst-set = "VAX",
    default-platform = "Ultrix",
    default-pkgs = {
        m2-bridge = m2-bridge.v,
        as-bridge = as-bridge.v,
        text = text.v,
        quadedge = quadedge.v,
        etc.
        },
    build-bridges = FUNCTION ... IN {
        M2 = BRIDGE( m2-bridge$build() ),
        AS = BRIDGE( as-bridge$build() ),
        etc.
        },
    }
IN  {
    Env-build = FUNCTION pkg-overrides, inst-set, platform IN {
        Env-for-bridges = building-env.v;
        LET {default-pkgs, pkg-overrides} IN {
            (* Build the bridges in the host environment *)
            LET Env-for-bridges$Env-default IN build-bridges();
            (* Build the interfaces for the target environment *)
            text$intfs();
            quadedge$intfs();
            etc.
            (* Build the libraries for the target environment *)
            SL-basics = M2$Lib( (text$impls(), etc.) );
            SL-ui = M2$Lib( (quadedge$impls(), etc., SL-basics) ),
            etc.
            }
        };

    Env-build-with-overrides = FUNCTION pkg-overrides IN {
        Env-build(pkg-overrides, default-inst-set, default-platform)
        };

    Env-default = Env-build-with-overrides( {} ),
    },
}
```

Figure 18: The `building-env` system model (part 2)

Imagine that we have completed the production of a consistent collection of software that we now wish to distribute. We want to distribute the collection of derived objects to a customer, but without the source from which it was produced. Furthermore, the customer may not use Vesta; indeed, the customer may not be doing software development at all.

Making a software distribution with Vesta is a two-step process. First, we need to acquire a precise description of the objects, in particular the derived objects, to be distributed. Second, we need to transfer these objects to the distribution medium.

Let's consider an example. Suppose that we wanted to distribute the `Hanoi` program (figure 6) with its documentation, but without revealing the sources or the means of constructing them. We apply a Vesta tool named `vcapture` to the `Hanoi` system model. `vcapture` invokes the Vesta evaluator on the `Hanoi` system model, looks at the resulting value, and writes a new system model (figure 19).

```
DIRECTORY
    Derived0001 = uid,
    Derived0002 = uid
IN  {
    test = {
        Hanoi = Derived0001,
        Hanoi.doc = Derived0002
        },
    }
```

Figure 19: "Backstop" system model for Hanoi

The UIDs in the `DIRECTORY` name the two derived objects that the `Hanoi` model in section 3.1 constructs. Notice that evaluating this mechanically-produced model gives the same result as evaluating the original, but without invoking any construction tools. We call such a model a *backstop*.

The backstop handles the first step of the distribution process. The second step is a straightforward transfer of this model and the objects in its `DIRECTORY` to a distribution medium, such as a Unix `tar` tape, or perhaps just a publicly accessible directory in an ordinary file system. This can be done with ordinary Unix tools.

Backstops can be produced for any Vesta model, not just for simple application packages. A backstop of `release` contains an entire collection of consistently constructed applications. A backstop of `building-env` constitutes a snapshot of a consistent software development environment (which is roughly equivalent to the union of the Unix `/usr/include` and `/usr/lib` directories, plus the subset of `/bin` that is used as software construction tools). Distributing this backstop conveniently transfers the development environment to the customers without exposing the sources that produced it. Furthermore, the developers can selectively include sources if they wish; they simply alter the original `building-env` model to include the appropriate sources in the value it produces when evaluated by `vcapture` to give the backstop.

If the customer who receives the backstop also uses Vesta, he simply puts the backstop into a (public) repository where it can be imported by his other system models. If not, the customer uses a simple utility to move the derived objects into appropriate places in the customer's file system, following the naming structure defined by the backstop model that is part of the software distribution.

6 Evaluation

Let's evaluate how Vesta addresses the three requirements stated in the first section.

Human work should be commensurate with the size of a change. The examples of sections 3–5 clearly show how a conceptually small change can be introduced by a small amount of editing of system models, even when the effect of the change, measured in the amount of mechanical work required to construct the resulting system, is very large. A single change to a fundamental interface may require editing one source line, but may trigger a complete reconstruction of the building environment.

We have considerable evidence that Vesta users appreciate and exploit this flexibility and power:

- In the course of one year, over 200 private versions of the `building-env` package were constructed. In many software development shops of comparable size, a consistent build of a couple of hundred library packages can't be done without major human effort. With Vesta, it occurred every day or so. Obviously, the users found it both easy and useful.

- The changes that users were willing to consider, such as low-level interface changes, were not constrained by the limitations of the software construction tools. We had repeated examples of:

  - rebuilding the entire environment to enable/disable execution profiling of libraries;
  - changing compiler versions to accommodate an incompatible object file format;
  - retargeting the entire operating system to a new hardware platform;
  - rebuilding the entire base of applications for a different operating system.

Conventional tools don't support most of these changes at all, or, at best, can barely be coaxed into doing them with a lot of manual labor.

Tools should support programmers working individually and cooperatively. Vesta's support for extensive parameterization makes it possible for a developer both to work independently and to combine her work with that of others. We saw how the overriding paradigm is used to enable a developer to test new components in a precisely controlled environment without affecting any package but her own. The paradigm applies uniformly from the simplest umbrella packages for a small project to the entire building environment, with hundreds of packages and dozens of developers.

We have evidence to support the utility of the overriding paradigm:

- Nearly every umbrella package in Vesta exhibits the facilities discussed in sections 4 and 5. Obviously, these mechanisms enable a development process that users find attractive.
- New public versions of the `building-env` package were announced at least weekly (over 60 in the space of one year). Each version represented a consistent integration of nearly 200 packages, many under active development. Yet, the responsibility for the weekly release of `building-env` was entrusted to a single person, who spent less than one-quarter of his time on it. Most of the problems that normally show up during conventional integration testing were exposed earlier in the development process by individual developers using the overriding mechanism to test in the context of the complete system rather than a test jig. Furthermore, individual developers could test at their own pace—no need to submit tentative changes to a mechanical "nightly build" as was the norm in the heyday of batch systems and, regrettably, is still common today.
- Even though new releases of `building-env` occurred frequently, the person responsible for the release rarely encountered pressure if a release was delayed. Again because of the overriding paradigm, individual users could construct custom versions of the environment they needed, without waiting for a centrally managed weekly release.

Software construction recipes should be complete and self-contained. Vesta system models completely describe components and the environment(s) in which they are constructed, both in terms of immutable source objects. This approach isolates developers from each others' changes unless and until they choose to share components. Equally important, it also isolates them from changes to the tools in the environment until they consciously decide to upgrade.

As evidence of the value of this approach, we observed that the SRC community of 25–30 users frequently had 10–15 versions of `building-env` in active use, of which two or three often differed only in the version of the Modula-2+

compiler (which was under active development at the time). Clearly, they valued the ability to control precisely the contents of their software construction environment. Traditional system-builders (e.g., `make` [Feldman]) either lack the facilities that enable this mode of operation, or support it so weakly that considerable discipline is required to achieve the desired degree of isolation. Vesta replaces human discipline with automation.

While the evidence supports the utility of our methodology for system models, we know of some things that won't scale well to larger systems. The name spaces used for package names and for (compiled) interfaces are flat. For SRC's purposes, this wasn't a problem—each contained under 2000 names. However, scaling up by a small factor would require imposing some structure on these name spaces. This, in turn, would add some complexity to the models that import `building-env` to establish an environment for construction. We anticipate that the added complexity would be small—a line or two to open a nested name scope. The umbrella packages would also need to add a line or two so that the overriding paradigm would continue to work. We don't see any technical difficulties here, but we haven't actually performed this change to see what would happen.

Finally, we reaped considerable benefits from the clear separation of the Vesta language and the conventions for applying it. The language itself is small and simple, with few features that directly stem from the desire to describe large software systems. The hard problem was to devise a workable collection of conventions and principles for using the language for software development. The examples in preceding sections demonstrate how we solved that problem. We expect the Vesta language and many of the principles we devised to work just as well in other programming shops, although the details of their application would naturally differ.

In conclusion, then, we believe that Vesta's novel language and repository facilities deliver significant advantages to software developers by drastically reducing the complexity of building and evolving large interrelated collections of software. We believe that our experience demonstrates that Vesta's configuration management abilities surpass those of many other systems.

7 Acknowledgements

Butler Lampson provided essential guidance during the design, development, and deployment of the system at SRC. The authors especially thank the many SRC researchers who unselfishly submitted to the inevitable inconvenience of using a prototype system in order to validate the Vesta approach.

References

[Brown and Ellis] Mark R. Brown and John R. Ellis. “Bridges: Tools to Extend the Vesta Configuration Management System.” Research Report 108, Digital Equipment Corporation Systems Research Center, June 1993.

[Chiu and Levin] Sheng-Yang Chiu and Roy Levin. “The Vesta Repository: A File System Extension for Software Development.”. Research Report 106, Digital Equipment Corporation Systems Research Center, June 1993.

[Feldman] S. I. Feldman, “Make—A Program for Maintaining Computer Programs”, *Software—Practice and Experience.* 9(4), April 1979.

[Hanna and Levin] Christine B. Hanna and Roy Levin. “The Vesta Language for Configuration Management.” Research Report 107, Digital Equipment Corporation Systems Research Center, June 1993.

[Lampson and Schmidt] Butler W. Lampson and Eric E. Schmidt. “Practical use of a polymorphic applicative language.” *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, January 1983.

[McJones and Swart] Paul R. McJones and Garret F. Swart. “Evolving the UNIX system interface to support multithreaded programs.” *Proceedings of the Winter 1989 USENIX Conference*, February 1989.

[Rochkind] Marc J. Rochkind. “The Source Code Control System.” *IEEE Transactions on Software Engineering.* SE-1, Issue 4, December 1975.

[Rovner *et al.*] Paul Rovner, Roy Levin, and John Wick, “On Extending Modula-2 for Building Large Integrated Systems”, Research Report 3, Digital Equipment Corporation Systems Research Center, January 1985.

[Tichy] Walter F. Tichy. “RCS—A system for version control.” *Software—Practice and Experience*, 15,7, July 1985.

[Wirth] Niklaus Wirth. Programming in Modula-2 (2nd edition). Springer-Verlag, New York, 1982.