Vesta User Manual

Last modified on Mon Jan 20 14:02:04 PST 1992 by mbrown
modified on Tue May 7 15:04:23 PDT 1991 by glassman
modified on Tue Dec 18 16:33:40 PST 1990 by hanna
modified on Thu Dec 13 17:13:33 PST 1990 by levin

January 22, 1992

Contents

1. Introduction

1.1 Vesta's goals and approach

Vesta seeks to simplify large-scale programming – the construction and evolution of large software systems. This includes the construction of small programs that rely upon a rich collection of libraries.

Programmers describe a software system to Vesta using a programming language that is modular and precise:

- The modularity allows one programmer to build upon the work of another, without understanding all of the details.
- The precision makes the construction of a system reproducible. Dependencies on changing source files, environment variables, and search paths are not allowed to creep in.
- The precision also allows Vesta, not programmers, to be in charge of all derived files (e.g. objects and images.) The programmer expresses his system entirely in terms of source files, and never names object files. Manual object file management is a big source of complexity in large-scale software development.

The Vesta language supports parameterization, so the programmer can describe how a system is built for multiple platforms from a single set of source files.

Vesta works with existing preprocessors and compilers. A compiler that's designed to take advantage of Vesta can give more functionality and improved performance.

The Vesta repository stores Vesta programs and source files containing programs written in conventional programming languages (C, Fortran,

etc.) The repository maintains the immutability of sources: Once a source file has been checked into the repository, it cannot be modified. Instead, a new version can be created.

The Vesta repository is designed so that source files have path names, as in conventional file systems. This storage model is familiar to users. The Vesta storage model also allows a Vesta repository to be "mounted" as a read-only file system, giving conventional programming tools access to sources.

Vesta is designed to work with multiple Vesta repositories, so that different groups can share work. Vesta repositories are designed to be replicated to give distant groups faster access and higher availability.

Vesta is motivated primarily by some recognized difficulties in the development of large systems: control of parallel development, version management, inter-component consistency control, bootstrapping, and porting. Vesta greatly reduces these difficulties. At the same time, Vesta is designed to accommodate small and large programs, single- and multi-person projects. It is useful both during routine development and when making sweeping changes to a system.

At SRC, Vesta is used routinely to build consistent versions of applications and libraries containing over 1,000 implementation modules, including SRC's Unix-like operating system (Taos). Vesta's precise descriptions enable reconstruction of entire collections of software automatically to incorporate incompatible changes – without disturbing users of that software.

1.2 Overview

If you are or plan to become a Vesta user, this manual is for you. It documents the user-visible capabilities of the Vesta server and the Vesta environment for building Modula-2+ programs at SRC ("buildingenv".)

This manual does not document the many Vesta tools, each of which has its own manpage. It does introduce some of the tools.

This manual does not document the Modula-2+ interfaces that a Vesta tool would call. These Modula-2+ interfaces contain comments that serve as documentation.

This is a thick document, and we don't expect you to read all of it before starting to use Vesta. Chapter 3, page 15 is a tutorial that will help you become a Vesta user. Before working through the tutorial, read Chapter 2, page 5, which covers the basic Vesta concepts that you need to know.

1.3 Getting help

Vesta may surprise you from time to time. For instance, a build may take longer than you expected, or you may have difficulty understanding an error message, or you may find some part of this manual unintelligible.

Don't hesitate to call on the members of the Vesta group when you have a problem. We have a selfish reason for wanting to help: We are looking for ways to make Vesta easier to use, and helping users is what makes us aware of the problem areas.

The first people to contact in case of problems are:

Bob Ayers	ext. 2153
Mark R. Brown	ext. 2207
Roy Levin	ext. 2218

Only if Bob, Mark, and Roy are unavailable should you contact:

Sheng-Yang Chiu	ext. 2236
Chris Hanna	ext. 2197

If you prefer, you can send one or all of us an electronic message, or you can post a message to src.vesta. Even if you manage to diagnose a problem without help from someone in the Vesta group, it is worth letting us hear about it via a message or posting.

2. Basic concepts

2.1 Objects

Vesta stores *objects*. Objects resemble files in a Unix file system, but in Vesta they have some properties not associated with Unix files. (Sometimes we lapse and call objects files.)

An object is either source or derived. A source object is one that Vesta cannot mechanically derive. Typically it contains source code typed in by a user. A derived object is one that Vesta can mechanically derive from other objects, for instance by running a compiler.

Vesta assigns a unique identifier, or UID, to every object it stores. The binding of UID to object is immutable, meaning that, whenever Vesta is capable of delivering an object corresponding to a particular UID, it delivers the same object.

For source objects, the association between UID and object is not just immutable, it is practically immortal: If Vesta ever delivers an object corresponding to a particular UID, it will continue to deliver the same object indefinitely thereafter. Vesta uses differencing techniques in the representation of source objects to reduce the additional storage required for multiple versions. We don't expect it to be necessary to delete any publicly-accessible Vesta source objects.

Derived objects cannot be kept indefinitely, because they are so large. We keep the space used by derived objects under control via manually-initiated garbage collections.

Vesta UIDs play a role analogous to inode numbers in a Unix file system. They are an implementation feature that you should be aware of but should not have to think about very often. They show through to you in today's Vesta only because today's Vesta user interface is primitive.

2.2 Repositories

Vesta repositories have two major roles: to store objects, and to give them sensible names. The UIDs that Vesta assigns to objects are too cumbersome for users to type or read, so Vesta repositories provide a hierarchical naming scheme for the source objects they store.

To a user, a repository appears as a subgraph of the ordinary, hierarchical file system name space. The name of the root of the graph is the repository name. Typical examples of repository names in use at SRC are /vesta/proj and /user/mcjones/vesta/private.

The name space within a repository is organized into a set of package families. Each package family is a tree of package versions. For example, /vesta/proj/text.3 names a particular package version in the text package family in the repository named /vesta/proj. Within a package family, the versions on the "trunk" have the form <familyname>.<number>, like text.3. Sprouting from the trunk are branches (sometimes called "forks"), with versions on the branch having names of the form <branchname>.<number>. So, text.3 is a version on the trunk, and text.3.bugfix.0 is a version on a branch emanating from the trunk, and text.3.bugfix.0.checkout.2 is a version on a second-level branch. Family trees may branch arbitrarily, but deep branching is unusual.

The value of a package version is a particular kind of source object called a *system model*. A system model has two parts, a directory and a construction specification. For now we'll ignore the construction specification and describe the directory.

The *directory* of a system model is a mapping from simple names to source objects. For instance, a directory might associate the name Text.mod with a source object containing Modula code.

You name a source object by combining the name of a package version's system model with a simple name from the directory of that model. The name /vesta/proj/text.3.bugfix.0/Text.mod corresponds to the object you find by going to the repository /vesta/proj, then to the system model text.3.bugfix.0 within that repository, then to the object associated with Text.mod in that model's directory.

A system model that's the value of a package version is called a *root model*. Not all models are root models. A system model is a source object that can be named in the directory of another model, producing a model

hierarchy. A model that's named in the directory of another model is called a *submodel.* For instance, the system model /vesta/proj/text.3 might contain a testcases submodel, /vesta/proj/text.3/testcases. This submodel might contain the test program TestText.mod, the full name of which would be /vesta/proj/text.3/testcases/TestText.mod.

Vesta does not permit a root model to also be a submodel. This restriction is designed to keep package families from sharing objects in an unstructured way. But some sharing between package families is often necessary. Therefore a system model usually has an *imports section* in addition to its directory. The imports section associates simple names, like text.v, with root models from other package families. Imports play a similar role to symbolic links in a Unix file system.

Since both the directory and the imports section of a model map simple names to source objects, we often refer to them collectively as the directory of a model.

2.3 System models

System models serve two roles. We've already seen that a model includes a directory which maps simple names to source objects. A system model also contains a construction specification that describes exactly how derived objects are to be produced from source objects. The directory is actually an integral part of the construction specification. Thus the entire model is really a construction specification; it was convenient to discuss the directory separately in the previous section.

A system model is a program written in the Vesta language, a functional programming language with a very simple type system. Vesta programs typically compute values that are programs and parts of programs, such as executable program images and object code files. But Vesta programs can also construct other sorts of derived objects, such as Postscript files produced by document compilers. The Vesta language is documented in Chapter 4, page 33.

To work productively in the Vesta environment, you need to write models that fit together with other models. The Vesta language does not impose a set of conventions to ensure this; as a writer of models, you have to exercise discipline. Chapter 7, page 121 describes the conventions for SRC's models.

2.4 Public and private repositories

Vesta defines two distinct repository types, public and private. A public repository holds package versions that multiple users share and potentially alter. The packages they contain have indefinite useful lifetimes; they are essentially archival. By contrast, a private repository is altered by a single user, and generally holds package version whose useful lifetimes are considerably shorter, say minutes to weeks.

2.5 Checkout sessions

To create a new package version in a public repository, you create what Vesta calls a *checkout session*. In effect, you declare your intention to provide a new system model to be bound to a particular package version; that is, you reserve a package version name.

For example, suppose you want to develop a new version of the text package, which lives in the public repository named /vesta/proj. Suppose that text.3 is the highest-numbered checked-in version on the trunk of the text family, and that nobody has reserved (checked out) text.4. Using UseVesta or vmake, you could create a checkout session for /vesta/proj/text.4.

A checkout session does more than reserve a name for you. Vesta expects that you'll need to create one or more intermediate versions of the text package before you've got one that you're willing to place in the public repository under the name text.4. You use a private repository for this purpose. For example, to checkout /vesta/proj/text.4, you might supply /user/bovik/vesta/private/text.4 as the package version in the private repository.

What's the purpose of this private package name? It's the basis of a series of names for the intermediate versions you'll build in the course of making your change to the text package. The names of those versions are under the control of the user-interface tool; the UseVesta tool would name them

```
/user/$USER/vesta/private/text.4.checkout.0
/user/$USER/vesta/private/text.4.checkout.1
/user/$USER/vesta/private/text.4.checkout.2
```

and so on. As you create new system models for the text package, they will be bound to successively numbered package version names. (You'll find out how to create models in section 2.7, page 10.)

Eventually, you complete your changes to the text package and want to make the result publicly available. You end the checkout session by asking your user-interface tool to perform a Checkin operation. You specify the package version (in your private repository) that is to be checked in (typically the last in the above sequence), and Vesta binds the reserved name /vesta/proj/text.4 to that model. It also binds the "root" of the above sequence, /user/bovik/vesta/private/text.4, to the same model.

So, package names can be in either a *bound* or an *unbound* state. They are bound when they have an associated system model. They are unbound when they are reserved as part of a checkout session. In the latter case, they come in pairs, or "partners" – one in a public repository and one in a private repository. You can use repository browsing tools (see section 3.2, page 16) to determine which names are bound and unbound and, in the latter case, what the partnership relationships are.

2.6 Disconnected sessions

A Vesta checkout session is fine for creating a new package or developing an existing one, but seems a bit heavyweight for exploratory development that you may wish to discard. To support such exploratory work, Vesta provides disconnected sessions.

A disconnected session involves only a private repository. You initiate it by specifying a "root" package name, much as you did for a checkout session. Suppose you decide to write a program to help you keep track your wine collection. You might begin a disconnected session with the package name /user/bovik/vesta/private/wine.1. As with the checkout session, your user-interface tool helps you to create a sequence of package versions derived from this root name, e.g.,

```
/user/$USER/vesta/private/wine.1.disconnected.0
/user/$USER/vesta/private/wine.1.disconnected.1
/user/$USER/vesta/private/wine.1.disconnected.2
```

and so on.

Disconnected sessions work well for exploratory programming when you're unsure that the result will ever need to be checked in to a public repository. But what if you subsequently change your mind? Vesta provides a means of converting a disconnected session into a checkout session, enabling you to say "I want to reserve /vesta/proj/wine.1 and operate as though I had checked it out to /user/bovik/vesta/private/wine.1, even though I've already created a bunch of related stuff."

2.7 Creating new package versions

You use the UseVesta tool to create new versions. UseVesta offers a rather extensive collection of facilities, which are described in detail in Chapter 8, page 139. We'll just cover a few of the basics here.

We can think of the outer loop of the program development cycle as "checkout", develop, "checkin". "Develop" is then the inner loop, which consists of "edit", "build", "run", "debug". You use your favorite tools to edit or otherwise create source files, then employ UseVesta to do the "build" step. UseVesta can also initate the programs you build, or you can arrange to run them from a shell. You use your favorite debugger for the "debug" step of the cycle.

UseVesta associates an ordinary Echo directory, called the *session directory*, with each checkout or disconnected session that you create. You do all your editing in this directory (and its subdirectories). You can edit with a text editor, with standard source transformation tools like sed, awk, and trans, or with specialized source transformers embedded as commands in UseVesta.

When you've completed your edits, you push UseVesta's **Advance & Eval** button. As the name suggests, this button combines two operations: Advance and Eval (each of which have separate buttons as well).

2.8 Advance

The Advance operation compares the state of your session directory with the state of the most recently created system model for the checkout (or disconnected) session. It creates a new system model that reflects the

present state of the session directory by incorporating each difference it finds (e.g., updated file, deleted file, created file).

For example, suppose that you have just begun a checkout session rooted at /user/bovik/vesta/private/text.3. As part of the Checkout operation, UseVesta created /user/bovik/vesta/private/text.3.checkout.0 and bound it to a copy of /vesta/proj/text.2 (which is the default behavior). It also made (mutable) copies in your session directory of all the source files mentioned in this model and its submodels. Now you use a text editor to modify Text.mod in the session directory. When your edits are complete, you push Advance. Advance notices that Text.mod has changed, so it creates a new source object in your private repository containing the bytes of the changed Text.mod. Advance then creates a new system model by copying the old model and binding Text.mod to the new source object in the directory portion. Finally, Advance copies this model into the repository, making it the value of /user/bovik/text.3.checkout.1.

Advance maintains a fairly intuitive mapping between the session directory's names and the most recent package version in your private repository. Ordinary Vesta source files have their names taken from the **DIRECTORY** clause of system models that name them. System models correspond to directories, which contain a file named **MODEL** that holds their text representation. The root model corresponds to the session directory itself; each submodel corresponds to a subdirectory of the session directory. Imported models (i.e., the ones named in the IMPORT clauses) are ignored; i.e., they aren't represented at all in the session directory.

2.9 Eval

The Eval operation interprets a system model as a Vesta program, which usually means that it constructs a piece of software. Thus, Eval is roughly equivalent to typing "make" in the world of Makefiles. The result of Eval is typically a collection of derived objects, which are stored directly in Vesta repositories; they don't appear in any session directory.

2.10 The building environment

In a Modula-2+ development environment based on *make*, the environment of a *make* invocation is defined by various search paths

(sequences of directories.) The compiler looks up imported interfaces using one search path, and the linker searches for libraries containing compiled implementations using a second path. *make* locates the compiler and linker themselves using a third path. You choose an environment by specifying these paths.

In Vesta, the environment of Modula-2+ interfaces, libraries, and tools is described by a system model. This allows Vesta to maintain the environment in many versions. The *buildingenv* family contains the versions of SRC's Modula-2+ environment. You choose an environment for a program you are developing by importing a specific version of buildingenv in the model describing your program.

In a *make*-based environment, the directories on a search path are mutable. The people sharing an environment may adopt conventions that limit the sorts of updates allowed to these directories, to reduce consistency problems. Vesta system models, like all Vesta sources, are immutable, which aids in keeping them internally consistent.

A version of buildingenv includes:

- compiled public interfaces;
- libraries that implement these interfaces;
- bridges, i.e., coherent groupings of language-specific tools;
- general-purpose tools, such as Bag and trans;
- default option settings for the bridges and tools, e.g., to enable garbage collection.

The default environment produced by buildingenv includes all of the interfaces imported by most application and library packages. It also includes a small collection of shared libraries (with names like SL-basics and SL-ui) that export these interfaces. The small number of libraries reduces the complexity of constructing applications, since you don't need to remember the names of a large number of libraries. You should usually be able to build an application by importing a single library.

The central facilities for compiling and linking programs reside in bridges. Vesta provides bridges for four languages: M2+, C, assembly, and shell (sh). The bridges export functions like `Compile` and `Prog` whose purpose is pretty obvious, and also export some functions whose purpose is more

obscure. Bridge functions may take a wide variety of options as parameters, corresponding to the command line switches you customarily include in Makefiles. Fortunately, buildingenv supplies default values for all these switches, so you don't have to think about most of them.

Since buildingenv is versioned, a particular version defines a particular set of bridges and therefore a particular set of compilers and other translation tools. The whole collection is under your explicit control, since you explicitly choose the version of buildingenv to build your package. Others may build new, even incompatible versions of the environment, including the compilers, without affecting your package version.

3. Tutorial

3.1 Becoming a Vesta user

To use Vesta, you must first be a Taos user. Next you need a Vesta server on your workstation.

Before you start a Vesta server for the very first time, run the vestanewuser program by typing

```
ff> vestanewuser
```

to a shell. This command configures some initial directories (associated with you, not your workstation) so that the Vesta tools will work properly. You can safely execute vestanewuser more than once – the second execution will have no effect.

Now you can start a Vesta server on your workstation by typing:

```
ff> startvesta
```

You can only have one Vesta server on your workstation; if you run startvesta a second time, it exits with the message

```
Sorry. There is already a Vesta server in place.
```

You'll probably want to add the startvesta command to your .login file, to start a server every time you boot and log in. If you log into your workstation remotely, your .login will try to start a second Vesta server, and you'll see the message above. Don't add the startvesta command to your workstation's echo.rc file, or you will experience access control problems. Soon, the Vesta server will be installed on all Taos machines and started automatically, like vulibrarymgr and dpserver are today.

3.2 Browsing the repository

3.2.1 Browsing a package version

Vesta repositories and the package versions that comprise them are part of the Echo name space, so you can browse them with standard tools (e.g., ls, cd). The standard public repository at SRC is named /vesta/proj. Try typing:

```
ff> cd /vesta/proj/random.4

ff> ls -F
MODEL*          Random.mod*      RandomPerm.mod*
Random.def*     RandomPerm.def*  tests/
```

What you see is similar to a directory containing a library being developed using make, but there are some obvious differences. First, the directory does not contain any object files, library files, or executables. This is consistent with the Vesta philosophy that derived objects don't have user-visible global names. Second, there is no Makefile in the directory. The MODEL file replaces the Makefile. Third, all of the files appear to be executable. This is done on purpose, but can be confusing.

In a Unix file system, there isn't any visible content to a directory other than the names it contains. In Vesta, however, system models do double duty as directories and as instructions for building software. You just used ls to view a model named /vesta/proj/random.4 as a directory. Now, to see it as a complete system model, cat the MODEL file:

```
ff> cat MODEL
(*  Copyright 1990 Digital Equipment Corporation.              *)
(*  Distributed only by permission.                            *)
(*  random.v                                                   *)
(*  Last modified on Tue Oct 29 13:03:58 PST 1991 by mbrown    *)
(*       modified on Mon Jun 18 16:24:40 PDT 1990 by levin     *)

DIRECTORY
  Random.def ~ /v/0haM08002P.Zay/S1bb.1.random,
  Random.mod ~ /v/0haM08002P.Zay/S1bb.2.random,
  RandomPerm.def ~ /v/0haM08002P.Zay/S1bb.3.random,
```

```
  RandomPerm.mod ~ /v/0haM08002P.Zay/S1bb.4.random,
  tests ~/v/0haM08002P.Zay/M1450.16.random,
IN {

  owner ~ "mbrown";

  intfs ~ LAMBDA ... IN
    LET M2 IN {
      Compile(Random.def);
      Compile(RandomPerm.def),
      },

  impls ~ LAMBDA ... IN
    LET M2 IN {
      Compile(Random.mod),
      Compile(RandomPerm.mod),
      };

  test ~ {} (* tests$outputs() *) ,

}
```

This undoubtedly looks rather foreign to you. Chapter 4, page 33 will describe the language of system models; for now, we'll stick to browsing.

The test program for the random package is located in the tests submodel, which you can list in the usual way:

```
ff> ls -l /vesta/proj/random.4/tests
total 15
-r-xr-xr-x  1 vestasrv      954 Oct 29 13:23 MODEL
-r-xr-xr-x  1 vestasrv    13449 May  8  1990 RandomTest.mod
lr-xr-xr-x  0 vestasrv        0 Nov  1 17:41 buildingenv.v ->
                                    /vesta/proj/buildingenv-vax.66
```

Notice that the file buildingenv.v in the tests subdirectory is a symbolic link to /vesta/proj/buildingenv-vax.66. When a model is viewed as a directory, its imports are viewed as symbolic links.

You can use Ivy to browse the content of a package. If you have been following the tutorial in a typescript, the working directory of your

typescript is /vesta/proj/random.4 (the typescript banner says "/vesta/.proj.vr/J/random.4".) You can visit any file in the random.4 package by middle-pressing the Win menu anchor, then picking the file you want. Pick MODEL to get a text editor window on the model.

Notice that the DIRECTORY portion of the MODEL file contains a listing of all the top-level files in the package. You can visit any file by left-clicking on (or selecting all the characters of) its name, then picking one of the GetSelected commands from the Util menu. For GetSelectedNear you can bypass the menu by typing option-G.

You can also view a Vesta source file via its UID. For example, note that the UID /v/0haM08002P.Zay/S1bb.2.random corresponds to /vesta/proj/random.4/Random.mod. You can left-click on (or select all the characters of) a UID and type option-G to open a window on that file. Try it.

3.2.2 Browsing the current release

What if you know the name of an interface, but don't know the name of the package family that contains it? The vxref (Vulcan cross reference) application answers questions of this sort by consulting files built from SRC's current Modula-2+ software release. For instance, if you know the name of an interface you want to examine, vxref will give you the full path name of the interface:

```
ff> vxref Random definition
Random    definition /vesta/proj/random.4/Random.def

ff> vxref Random implementation
Random    implementation /vesta/proj/random.4/Random.mod
```

Assuming that you've run these vxref queries in an Ivy typescript, you can visit either file by left-clicking on the path name and typing option-G.

See the vxref(1) manpage for more complete information on the capabilities of vxref. Ivy's MpIndex command has a subset of vxref's functionality, but is based on ee(1).

3.2.3 Browsing package families

While standard tools work adequately for browsing particular package versions, a specialized tool works best for examining family trees, since they have considerably more information than can be encoded in an ordinary directory. ManageReps is such a tool. (It is so named because it provides a variety of facilities for creating, altering, and examining repositories.)

Start up ManageReps by typing

```
ManageReps &
```

to a shell. Deiconize the resulting window.

ManageReps works on one repository at a time. When you start it up for the first time, it looks at /vesta/proj, which is what you usually want. Notice that it is displaying a subwindow with a two-column listing of the package families in /vesta/proj.

Suppose you want to look at the vestarep family tree. Select the word "vestarep" anywhere on the screen (for instance, near the bottom of the /vesta/proj subwindow) and click one of the two **Show** buttons. Clicking the upper **Show** button creates a new subwindow for viewing vestarep; you can get rid of the subwindow by clicking its **Shut** button. Clicking the lower **Show** button that's part of the /vesta/proj subwindow replaces the contents of that subwindow with a listing of the vestarep family; you can restore the previous contents of the subwindow by clicking its **Prev** button.

In either case the family tree for the vestarep package is displayed; as of 28 November, 1990, the family had ten members:

```
vestarep.0
vestarep.1
vestarep.2
vestarep.3
vestarep.4
  vestarep.4.chiu.0
  vestarep.4.chiu.1
  vestarep.4.chiu.2
vestarep.5
vestarep.6
```

Note that the sequence of versions isn't strictly linear; a branched line of development occurred at vestarep.4. You can use ManageReps to find out more about it. Select vestarep.4 (selecting the entire line, with a double-click, will do) and click either of the two Show buttons again to produce:

```
Bound state = Bound
  Creator     = chiu
  Known at    = Wed Nov  7 15:29:20 PST 1990
  Bound at    = Wed Nov  7 16:15:38 PST 1990
Came from   = /-/com/dec/src/user/chiu/vesta/rep/vestarep.4
Partner     =
Self        = /vesta/proj/vestarep.4
Session Dir =
Value       = /v/0haM08002P.Zay/M191.21.vestarep
Comment     = another attempt at the first version of vestarep for echo
```

Known at says when the package version was checked out, **Bound at** says when it was checked in. You can ignore the other fields for now.

Now select vestarep.4.chiu.0 and click Show:

```
Bound state = Bound
  Creator     = chiu
  Known at    = Wed Nov 14 15:47:10 PST 1990
  Bound at    = Wed Nov 14 15:57:15 PST 1990
Came from   = /-/com/dec/src/user/chiu/vesta/rep/vestarep.4.chiu.0
Partner     =
Self        = /vesta/proj/vestarep.4.chiu.0
Session Dir =
Value       = /v/0haM08002P.Zay/M239.2.vestarep
Comment     = fixed bug in DataStore.ReadUID for deriveds
```

Also, select vestarep.5 and click Show:

```
Bound state = Bound
  Creator     = chiu
  Known at    = Wed Nov 14  9:23:06 PST 1990
  Bound at    = Mon Nov 26 11:18:20 PST 1990
Came from   = /-/com/dec/src/user/chiu/vesta/rep/vestarep.5
```

```
Partner     =
Self        = /vesta/proj/vestarep.5
Session Dir =
Value       = /v/0haM08002P.Zay/M237.fa4.vestarep
Comment     = see README file in the model for the change log
```

Observe that the time-sequence of checkouts is vestarep.4, vestarep.5, vestarep.4.chiu.0, but the checkin time-sequence is vestarep.4, vestarep.4.chiu.0, vestarep.5. It seems likely that the branch was created to repair a bug in vestarep.4, which was discovered after vestarep.5 was already under development. The comment on vestarep.4.chiu.0 supports this interpretation.

3.3 Creating a private repository

Now that you know how to browse, let's write a first program. First, however, you need to create a private repository. In the remainder of this chapter, we assume that your user name is "bovik".

Click on the **Create Rep..** button in ManageReps. Fill in the Repository Name field with the private repository name '/user/bovik/vesta/private'. (If you really hate this name you can choose another one starting with '/user/bovik/vesta', but UseVesta will work less smoothly if you insist on being creative with the name. Most people only ever need one private repository, and the vestanewuser program created the directory /user/bovik/vesta for you.) If you are at SRC, fill in the Repository's Home Replica field with the string 'src'. Click Create Private Rep.

Later, after you have created some packages in this repository, you may want to browse it. To direct ManageReps' attention to this repository, depress the left mouse button on the **Switch Rep~** menu anchor, drag to the **To /user/<user>/vesta/private** command, and release the button. You can try this now if you wish, although there won't be much to see.

3.4 Writing a first program with UseVesta

To get acquainted with the UseVesta tool and Vesta itself, let's write a trivial program.

3.4.1 Creating a Place for Session Directories

Recall from section 2.7, page 10 that with UseVesta you use ordinary Echo directories to perform your edits. Many users find it handy to have a standard subdirectory of their home directory in which all of these short-term directories appear. We'll suppose you want to do the same, so go to a shell and type

```
mkdir ~/wds
```

to create this directory.

3.4.2 Starting a session

You're going to write a "Hello, world" program. Begin by creating a session directory in which to edit:

```
mkdir ~/wds/hello
```

Now create a UseVesta instance with that directory as its sesssion directory by typing

```
cd ~/wds/hello
UseVesta &
```

to a shell. Then wait for the UseVesta icon to appear, and open it.

When you open the UseVesta window, you'll notice several types of trailing punctuation in the UseVesta control panel buttons. Ordinary, single-function buttons have no trailing punctuation. Menu anchors have the "~" suffix. Buttons that unconditionally produce pop-up forms end with "..". Finally, buttons that conditionally produce a pop-up end with ",,". If you hold down the Option key when invoking a ",," button, the pop-up occurs; otherwise, a default behavior is invoked without further interaction.

In order to work on the "Hello, world" program you need a Vesta session. Since you don't plan to save the "Hello, world" program in the public repository, you need a disconnected session.

Depress mouse-left on **Cmds˜** menu anchor, drag down to **Create Disconnected..**, and let up. A dialog will appear at the bottom of the UseVesta window. Supply a package name, hello.1, in the first line of the dialog. Finally click **Create Disconnected** in the dialog to create the session.

3.4.3 Writing the program

Using a text editor, create the file Hello.mod in your session directory (/wds/hello) containing

```
(* Last modified on Fri Nov  1 12:20:03 PST 1991 by bovik *)
SAFE MODULE Hello;
  IMPORT Stdio, Wr;
BEGIN
  Wr.PutText(Stdio.stdout, "Hello, world\n");
END Hello.
```

3.4.4 Making a system model

UseVesta can fill in a system model template from the contents of a session directory. Standard templates exist that follow the SRC conventions for application and library packages.

Holding down on the Option key, invoke the **Recreate Models,,** command in the **Misc˜** menu. This opens the Recreate Models dialog. Change the **Template =** field to Application.template, since Hello.mod is an application program. Turn on the boolean **Ignore Existing Model Body**. Then click the **Recreate** button. This creates the file **MODEL** in your session directory. Open this file with a text editor; you see

```
DIRECTORY
  Hello.mod ˜ ,
IMPORT
  buildingenv.v ˜ <uid>,
IN   {
  owner ˜ "replace string between quotes with name of owner",

  build ˜ LAMBDA ... IN
```

```
    LET {
      M2;
      (*GENERATE-BEGIN SortedDefs*)
      (*GENERATE-END SortedDefs*)
      }
    IN {
      program_name ~ Prog ((
        (*GENERATE-BEGIN Mods*)
      Compile(Hello.mod),
        (*GENERATE-END Mods*)
        (* suitable libraries *)
        )),
      };

  test ~
    LET buildingenv.v$Env-default() IN build(),

  }
```

Unfortunately, you now have some hand editing to perform on this generated model.

Insert a SRC timestamp line as a comment at the top of the model. You can delete the comment lines with **GENERATE** in them (see Chapter 8, page 139 for an explanation of situations in which they are useful.) You should change "program name" to "hello", and fill in your name as the value of 'owner'.

You need to pick the correct value for the buildingenv model, which is the most recently bound member of the buildingenv-vax family. Go to a shell and type

```
vs buildingenv-vax
```

You'll get a bunch of output, the first line of which will look similar to

```
Package /vesta/proj/buildingenv-vax.66:
```

This is the information you need. Change the line

```
buildingenv.v ~ <uid>,
```

to something like

```
buildingenv.v (*/vesta/proj/buildingenv-vax.66*) ~ ,
```

deleting the UID and using the version number you got as output from vs.

Finally, you need to specify the library to link into your program. Recall that buildingenv delivers a small set of libraries corresponding to major "abstraction levels" in the environment. For a simple program with no need for windows, the library **SL-basics** is adequate. Replace the comment **(* suitable libraries *)** with the name **SL-basics**.

You've produced a usable system model, which should look similar to this:

```
(* Last modified on Wed Oct 30  9:30:25 PST 1991 by bovik *)
DIRECTORY
  Hello.mod ~ ,
IMPORT
  buildingenv.v (*/vesta/proj/buildingenv-vax.66*) ~ ,
IN   {
  owner ~ "bovik",

  build ~ LAMBDA ... IN
    LET {
      M2;
      }
    IN {
      hello ~ Prog ((
        Compile(Hello.mod),
        SL-basics
        )),
      };

  test ~
    LET buildingenv.v$Env-default() IN build(),

  }
```

Tell your text editor to save the **MODEL** file. Click the UseVesta **Advance** button. If you are not using Ivy to do your editing, when the Advance the command has completed, re-read the **MODEL** file into your editor (if you are using Ivy, the window will reset automatically.) You'll see that Advance has supplied UIDs for Hello.mod and buildingenv.v.

3.4.5 Building the program

You're ready to build. Left-click **Eval,,**. UseVesta reports that the evaluation has begun. The first Vesta evaluation in a new server is slower than later ones, because Vesta has to read in caches and start bridges (e.g. the Vulcan compiler.) After about a minute, Vesta will report "compiling Hello.mod", then "binding...", and finally "making executable...". After less than a two minutes have gone by, UseVesta should report "Evaluation successful".

3.4.6 Running the program

OK, you've built the program; now you'd like to test it. Left-click **Run,,** and a dialog will appear. Click **Run** in the dialog to run the program. A new terminal window "Run: hello" will appear, and stdout from the running program will be displayed there.

Another way to test a program is to copy the executable from your Vesta repository to a file in the ordinary Echo name space. With a small change to the model, Vesta will perform the copying during evaluation.

Return to the editor window on your MODEL file, and replace the lines

```
test ~
  LET buildingenv.v$Env-default() IN build(),
```

with

```
ship ~ LAMBDA dir, ... IN
  CopyM2ProgToFile(hello, CAT((dir, "/bin/hello")));

test ~
  LET buildingenv.v$Env-default() IN {
    build();
    ship("/tmp");
    },
```

Now, when the test component of the model is evaluated, Vesta will copy the 'hello' program into the file named /tmp/bin/hello. Tell your editor to save the **MODEL** file.

Now click **Advance & Eval**. Because you haven't changed Hello.mod, Vesta knows not to recompile it or to relink the executable 'hello'. In less than a minute, the evaluation completes successfully. Now type

```
/tmp/bin/hello
```

to a shell to see if the program is working correctly.

A pitfall to avoid: The Advance operation generally alters the **MODEL** file, since it must rewrite the **DIRECTORY** portion whenever it discovers a modified or newly-created file in the session directory. So, if you leave the **MODEL** file in a text editor window and invoke Advance, the window contents will not reflect the current contents of the file. UseVesta and vmake will reset Ivy windows to bring them up to date; if you use another editor, you'll have to do this manually.

3.4.7 Debugging the program

The UseVesta **Run** dialog contains options that allow you to debug your program.

If you've copied a program to /tmp, here's an incantation for starting to debug it. Go to a typescript running a shell and follow this transcript:

```
ff> vdebug /tmp/bin/hello &
[1] 19457
pid 19457:
ff> vulcan -stdio -debug 19457
Using bridge /v/0haM08002P.Zay/D13be.FWNAepCfFkv-Gsqu.vulcan
Connected to self_19457
Target is stopped; generating stack trace...
thread 083bdbdb0H, id 1, help trap
"Initial Pause"
<_TPSpecialStub + 017eH (078cc86e6H)>:
stopped
<1>
```

Notice that the vdebug command prints out a number (the pid of the process created by vdebug) and you must feed this same number to the vulcan command.

See the Vulcan Loupe documentation (readdoc vulcanloupe) for information about debugger commands.

If you are using Ivy, you should know that the Vulcan debugger can display the current location in the current source file whenever you execute a frame or control command. Type

```
(TlLoad.Import "Loupe")
```

to your ivyserver typescript, and

```
ivy TRUE
```

to your Vulcan Loupe typescript. (This feature was announced in message /usr/spool/news/src/vulcan91b/3160 from Michael Sclafani.) If you like this behavior, you can enable it once and for all by putting the ivyserver command in your .ivyConfig/UserProfile.tl and the Loupe command in your .louperc.

3.4.8 Abandoning the session

When you are finished with the "Hello, world" program, you should abandon the disconnected session by picking **Abandon** from UseVesta's **Cmds~** menu. Abandoning the session will also delete the files from your session directory, so you must confirm the operation by clicking the **Confirm Abandon** button in the confirmation dialog that pops up. Then delete the UseVesta window.

3.5 A checkout session

Now that you know about the inner loop of software development in Vesta, let's look at the outer loop. You'll check out an existing package, make a small change, and check it in again.

The package you'll use is in the public repository /vesta/tutorial, and is called 'easter'. Use ManageReps to look at this package family. If you are looking at this document online, just select "/vesta/tutorial" in this document and pick **To selected name** from the /tt Switch Rep~ menu.

Notice the highest bound package version in the family. As of this writing it is /vesta/tutorial/easter.2, but a newer version could have been introduced since this tutorial was written. You are going to make a new version of the program on a branch off the trunk of the family tree. You will name your package version easter.$HIGH.bovik.0, where "$HIGH" is a place-holder for the version number of the highest bound package version.

3.5.1 Checkout

Begin your check-out session much as you began your disconnected session:

```
ff> mkdir ~/wds/easter

ff> cd ~/wds/easter

ff> UseVesta &
```

Now you need to direct UseVesta's attention to the /vesta/tutorial public repository. To do this, use the **Change Context (reps, etc)..** command in the **Misc~** menu, which brings up a dialog with fields for the public and private repository, and other things. The private repository should already be set correctly (see section 3.4.2, page 22). Fill in /vesta/tutorial for the public repository. Click **Set**.

Now, click the **Checkout..** button at the left side of the control panel. A dialog appears. Fill in the package version name (easter.$HIGH.bovik.0) you want to reserve (check out). Click **Checkout**. UseVesta will make the reservation, create a package named easter.$HIGH.bovik.0.checkout.0 in your private repository, bind it to the same UID as /vesta/tutorial/easter.$HIGH, and set up a session directory corresponding to this system model. When UseVesta reports that the checkout has completed successfully, look in the sesion directory. You should find files named **MODEL** and Easter.mod. (You will also find a collection of dot-files created by UseVesta; ignore them.)

3.5.2 Changing the package

Once the checkout session is underway, the inner loop of software development is similar to the case of a disconnected session. In section 3.4.3, page 23, you began an entirely new program. This time, you'll modify an existing one, which, after all, is the more usual case. So, you won't have to generate a system model from scratch; there's already one from the previous version of the package.

The real work of the Easter program is in the procedure Compute in Easter.mod. If you look at it for a minute or two, you'll discover that there's no immediate reason to believe that the day it computes is a Sunday. So, it might be worth including an independent check to that effect. This is the change you are to make.

You could code your own algorithm for computing the day of the week, or you could use the functions ToTime and FromTime, which you can find in the TimeConv module in the current release in /vesta/proj. Use the browsing techniques you learned in section 3.2.2, page 18 to find the source for TimeConv.def.

The procedure ToTime ignores the 'dayOfWeek' and 'dayOfYear' fields of a TimeConv.T record, so you can fill in all the other fields and use ToTime to compute a Time.T. Then you can use FromTime to compute the day of the week. (This technique does not work for years before 1970, because of the way the type Time.T is defined.)

After completing your edits, use **Advance & Eval** to build a test version. Notice that the **MODEL** stores the executable as /tmp/bin/easter. Run it to see if your change works. If not, use the debugger (as in section 3.4.7, page 27 or whatever debugging technique you prefer, then edit Easter.mod and iterate as necessary.

3.5.3 Checkin

Now that you're satisfied with the modifications you've made, you can complete the checkout session. Click **Checkin..** and note the dialog that appears. There is a field to fill in a brief comment about the package version. (If you'd like to put in a longer comment, you may find it easier to dismiss the checkin dialog and use the **Change Comment..** command in the **Cmds˜** menu. It contains a multi-line, scrollable comment area.)

After filling in the comment, click **Checkin**. UseVesta will copy the files into the public repository and bind the reserved name /vesta/tutorial/easter.$HIGH.bovik.0 to your final system model. Finally, UseVesta deletes the source files in your session directory; you can always find them in the repository.

3.6 Learning to do more with Vesta

Now that you've worked your way through some tutorial sessions, you have a basic familiarity with Vesta. Of course, you'll want to work on more sophisticated programs than the examples of the preceding sections. This is the end of the tutorial material, but there's quite a bit of reference material on Vesta that should help you use Vesta for real work.

To learn about the other facilities of UseVesta, read /vesta/out/rls/doc/printdoc/UseVesta.doc. To learn more about ManageReps, read /vesta/out/rls/doc/printdoc/ManageReps.doc.

Use 'apropos vesta' to get a list of all the Vesta tools. You may be interested in vmake, a command-line interface to much of the UseVesta functionality. The vs command gives a useful subset of the ManageReps functionality.

To learn how to construct a Vesta model for a library package, or for a package that includes server programs, consult chapter 7, page 121.

To learn about the Vesta language, read chapter 4, page 33.

To learn more about the Modula-2+ bridge and the other available bridges, see chapter 5, page 83.

To learn more about repositories, including the tools that manage the storage occupied by repositories, see chapter 6, page 115.

To learn about things that are missing from Vesta at present, consult chapter 10, page 181.

4. The Vesta Language

4.1 Language Overview

The Vesta language is a special-purpose functional language, intended to permit a precise, modular description of a software system. The system description is called a *model*. When evaluating a model, the Vesta evaluator constructs (compiles and links) the software system described by the model.

Because the Vesta language is functional, it lacks the notion of a variable. You apply functions to values to yield more values, then apply functions to these values, and so on.

Because the Vesta language is special-purpose, it does not include some traditional features of general-purpose functional languages, such as explicit support for recursion, data abstraction, and pattern matching.

Each Vesta value has an associated type. This type is not statically expressed in the Vesta program but instead is dynamically carried with the value at runtime. Runtime checks prevent the evaluation of a model from breaching the Vesta language's type system.

4.2 Language Syntax

4.2.1 Grammar Notation

The syntax is presented using an extended BNF, which can be summarized by the following rules:

- Characters that appear literally are surrounded by double-quotes. e.g. **"LAMBDA"**.

- Alternatives are separated by vertical bar, e.g., `":" | ","`.
- Parentheses are used for groups.
- `[ X ]` means zero or one occurrence of `X`, i.e., empty or `X`.
- `*( X )` means zero or more repetitions of `X`, i.e., empty, `X`, `X X`, `X X X`, etc.
- `+( X )` means one or more repetitions of `X`, i.e., `X`, `X X`, `X X X`, etc.
- `**( X Y)` means zero or more repetitions of `X` separated by `Y`, i.e., empty, `X`, `X Y X`, `X Y X Y X`, etc.
- `++( X Y)` means one or more repetitions of `X` separated by `Y`, i.e., `X`, `X Y X`, `X Y X Y X`, etc.

4.2.2 Lexical Elements

Character Set

The character set includes:

1. upper and lower case letters:

   ```
   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
   ```

2. digits:

   ```
   0 1 2 3 4 5 6 7 8 9
   ```

3. graphic characters:

   ```
   ! " # $ % & ' ( ) # + , - . / : ;
   < = > ? @ [ \ ] ^ _ ` { | } ~
   ```

4. the space character
5. the formatting characters: newline, backspace, horizontal tab, vertical tab, form feed, and carriage return
6. the control characters with values `\000` through `\007` and `\016` through `\037`, which only appear in comments and strings.

Whitespace and Comments

The whitespace characters are space, horizontal tab, vertical tab, newline, carriage return, formfeed, and comments. Whitespace is not significant in a model except when it is used to separate tokens or when it appears in string constants. Whitespace is required to separate a keyword, identifier, or number from a following keyword, identifier, or number.

A comment is an arbitrary character sequence that begins with (* and ends with *). Comments may be nested.

Identifiers

```
N          ::=   letter *( alpha | digit )
alpha      ::=   letter | "_" | "-" | "."
letter     ::=   <an upper- or lower-case letter: A-Z, a-z>
digit      ::=   <a digit: 0-9>
```

Identifiers are used as names in models. All characters in an identifier are significant. Case is also significant in identifiers; identifiers differing only in use of corresponding upper and lower case characters are considered different.

Reserved Words

Reserved words have the same syntactic structure as identifiers, but are used for special purposes in the grammar and so may not be used as identifiers. To aid a user in distinguishing reserved words from identifiers, reserved words are all upper-case. The reserved words are:

```
DIRECTORY     IF          IN        LET
ELSE          IMPORT      LAMBDA    THEN
```

Numeric Literals

```
number     ::=   decimalnum | hexnum
decimalnum ::=   +( digit )
hexnum     ::=   digit *( digit | hexdigit) "H"
hexdigit   ::=   "a" | "b" | "c" | "d" | "e" | "f"
```

Literal numbers represent non-negative integers. A literal integer is a sequence of digits with a suffix to indicate hex numbers.

A literal decimal number uses the digits `0` through `9`.

A literal hex number uses the digits `0` through `9` and `a` through `f` and is suffixed by "H". To avoid ambiguity, hex numbers must begin with `0` through `9`.

Text Literals

```
text      ::=   quote *( non-quote | special ) quote
quote     ::=   <the character ">
non-quote ::=   <any printing character other than " or \>
special   ::=   "\" specchar | "\" octal octal octal
specchar  ::=   "n" | "t" | "b" | "r" | "f" | "\" | quote
octal     ::=   "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
```

A text literal is a sequence of zero or more characters or escape sequences enclosed in double quotes. A text literal cannot extend over the end of a line.

The following table describes what the special characters in a text literal describe:

`\n`	**linefeed**
`\t`	**tab**
`\r`	**carriage return**
`\f`	**form feed**
`\b`	**backspace**
`\\`	**backslash**
`\"`	**double quote**

A \ followed by exactly three octal digits specifies the character whose ASCII code is that octal value. A \ followed by anything other than an escape sequence or an octal value is an error.

UID Literals

```
uid    ::= uid1 | uid2
```

```
uid1   ::= "/v/" word65 "/" kind1 word16 "." word16 "." word64
uid2   ::= "/v/" word65 "/" kind2 word16 "." word64 "." word64

kind1  ::= "S" | "M"
kind2  ::= | "D" | "L"
word16 ::= +( digit | hexdigit )
word64 ::= +( letter | digit | "_" | "-" )
word65 ::= +( letter | digit | "_" | "-" | "." )
```

UIDs name objects in the Vesta repository. In general, a user need not be concerned with the syntax of a UID literal since UID literals are created by the Vesta system; at most, a user will need to copy a UID literal from one model to another. The syntax is given here only for completeness.

In a Vesta model UIDs can appear only as UID literals; they cannot be constructed, such as from their substrings.

Operators, Separators, and Terminators

There are only two operators:

```
$     -
```

The separator characters are:

```
( )     { }     ,     ;
```

As the grammar shows (page 37), semicolon and comma may be used as either separators or terminators. The semantic specifications ignore the possibility of terminators; that is, a trailing punctuation character has no effect on the semantics. (If we had a pretty-printer, it would preserve trailing punctuation for you.)

4.2.3 Grammar

```
model     ::=   directory
```

```
(* expression *)
E          ::=   N
               | literal
             (* field selection *)
               | E "$" N
             (* list (sequence of value) *)
               | "(" [ ++( E "," ) [ "," ] ] ")"
             (* binding (set of name-value pair) *)
               | "{" [ ++( bind-gp ";" ) [ "," | ";" ] ] "}"
             (* name introduction *)
               | "LET" E1 "IN" E2
             (* function definition *)
               | "LAMBDA" [ ++( N "," ) [ "," ] ] ["..."] "IN" E
             (* function application *)
               | E1 E2
             (* conditional *)
               | "IF" E1 "THEN" E2 "ELSE" E3
bind-gp    ::=   ++( bind-elt "," )
bind-elt   ::=   E | N "~" E

(* directory clause *)
directory  ::=   "DIRECTORY" entries ["IMPORT" entries] "IN" E
entries    ::=   [ ++( entry "," ) [","] ]
entry      ::=   N "~" uid
```

4.2.4 Precedence and Associativity Rules

Here are the classes of operator precedence, from lowest to highest:

```
lowest:   (), {}
          semicolon
          comma
          LAMBDA, LET, IF
          selection ($)
highest:  application
```

Within a binding constructor, comma and semicolon are left-associative among themselves. Section 4.4.5 explains the significance of this rule.

Function application is left-associative, so

```
f g 3
```

means "apply f to g and apply the result to 3".

```
f {g 3}
```

means "apply g to 3 and apply f to the result".

Selection is left-associative, so

```
e1$A$B
```

means "select A from the expression e1, and then select B from the result of the first selection".

Braces are overloaded to distinguish bindings and also to indicate grouping. Parentheses are used only as list constructors, so

```
f (g 3)
```

means "apply g to 3, construct a single-element list containing the result, and apply f to it".

4.3 Language Semantics—Preliminaries

4.3.1 The Specification Language

The semantics are specified semi-formally, using Modula-2+ as the operational specification language. The specification also uses the SRC Text and List interfaces for commonly-used types and procedures.

4.3.2 Type System

The Vesta language is dynamically typed. Types are associated with runtime values instead of language variables and expressions, and so can be determined only at runtime.

The Vesta language is also strongly typed; that is, an ex . model is not permitted to breach the language's type system. Durin ion, all the values that are parameters to primitive functions are type and errors are flagged for values with incorrect types. For a discu the types for each primitive function, see Section 4.5. Bridges may also type-check parameters to bridge functions and identify type errors. Consult each bridge's documentation for a description of the runtime type-checking that it performs.

4.3.3 Scope Rules

The underlying scoping mechanism of the Vesta language is static; that is, free variables in a function get their values from the environment in which the function is defined. Typically, in a statically scoped language, each use of an identifier is associated with the innermost lexically apparent binding of that identifier. Because the Vesta language has binding values, however, the binding associated with the use of an identifier cannot be determined in general until runtime. See Section 4.4.5 for a further discussion of this issue.

In addition to static scoping, dynamic scoping may be used for free variables in a function body. The function signature regulates whether dynamic scoping is used. With dynamic scoping, free variables in a function get their values from the environment in which the function was called. See Sections 4.4.7 and 4.4.8 for a discussion of how to define and apply a function that uses dynamic as well as static scoping for the free variables in its body.

4.3.4 Models and Including Models

A model is the unit of evaluation. A model is composed of a DIRECTORY expression, which is the only top-level expression in the model.

One model may reference another by simply naming the other model. See Section 4.4.10 for details.

4.3.5 The Value Space

In the runtime value space, values are typed. A value is basically a two-element record with a type and the value whose interpretation depends on the type.

There are seven possible types:

```
TYPE Type = (BindingT, BooleanT, ClosureT, IntegerT,
             ListT, OpaqueT, TextT);
```

The Vesta evaluator understands the semantics of values of all of these types except for (most) **OpaqueT** values. The structure of values is described by the following definitions:

```
TYPE
  Value = REF ValueR;
  Values = List.T (* OF Value *);
  ValueR = RECORD
    CASE type: Type OF
    | BindingT:
        belems: List.T;
    | BooleanT:
        bool: BOOLEAN;
    | ClosureT:
        formals: Text.RefArray;
        defaultFormals: BOOLEAN;
        body: REFANY; (* Value or Expr *)
        env: Env;
    | IntegerT:
        int: INTEGER;
    | ListT:
        lelems: Values;
    | OpaqueT:
        opval: ARRAY OF BYTE;
        bridge: Bridge;
    | TextT:
        text: Text.T;
    END;
  END;
```

Typed values are represented as follows:

- If **type** = **BindingT**, then the value has one component, **belems**, which is an (unordered) association list. Each element of the list is a two-element list (**n**, **v**), where **n** is an identifier, and **v** is a **Value**. The identifier components of all the two-element lists are distinct.

- If **type** = **BooleanT**, then the value has one component, **bool**, which is either TRUE or FALSE.

- If **type** = **ClosureT**, then the value has four components: **formals**, **env**, **body**, **defaultFormals**. **formals** is a list of identifiers that constitute the formal parameters (the "domain" of the function). **env** is the static environment of the function definition (see Section 4.3.6). **body** is the function body, which may be represented in one of two ways: either it is a syntactic form (corresponding to the body of a **LAMBDA** expression) or it is an opaque value constructed by some bridge. **defaultFormals** is a boolean flag that controls the scope of name binding for free names in **body**.

- If **type** = **IntegerT**, then the value has one component, **int**, which is a signed number.

- If **type** = **ListT**, then the value has one component, **lelems**, which is an ordered list of heterogeneous elements.

- If **type** = **OpaqueT**, then the **bridge** component identifies the bridge that produced the value (in an implementation-dependent way). The **bridge** component identifies the bridge that knows how to interpret the **opval** component of the value.

  - If the **bridge** component is some bridge other than the Vesta bridge, then the value is a sequence of bytes that is uninterpreted by the evaluator. The **opval** field of these values is not understood by the Vesta evaluator.
  - If the **bridge** component is the Vesta bridge, then the **opval** field either represents one of the primitive functions or constants, or the **ERROR** value. The primitive functions and constants are described in Section 4.5. The **ERROR** value is produced by the Vesta evaluator as a result of an illegal expression, or as a result of the primitive function **ERROR** (see Section 4.5.8).
  - The **ERROR** value is contagious in use. So, for example, the model fragment:

```
LET {err ~ ERROR("")} IN
  (PLUS(1, err), err)
```

evaluates to a list with two elements, each of which is the **ERROR** value. The fragment:

```
LET
  {a ~ ERROR(""),
   b ~ 5}
IN
   b
```

evaluates to 5.

- If **type** = **TextT**, then the value has a single component, **text**, which is a sequence of characters.

(The preceding description of Vesta values does not model one detail: A value that's the result of evaluating a model carries with it the UID of the model. This allows the **DIRECTORY_OF** and **IMPORTS_OF** built-in functions described in Section 4.5.7 to work.)

External Representation

Each value has an external representation as a sequence of characters:

- If **type** = **BindingT**, the binding elements are printed in an undetermined order, enclosed by curly braces, and separated by commas. Each binding element is represented as the element name, followed by a colon, followed by the external representation of the element value. Given the expression:

  ```
  {a ~ 1, b ~ 2; c ~ PLUS(a, b)}
  ```

  an external representation of its value is:

  ```
  {a: 1, b: 2, c: 3}
  ```

- If **type** = **BooleanT**, the external representation is **TRUE** for TRUE, and **FALSE** for FALSE.

- If **type** = **ClosureT**, the external representation is a four-tuple enclosed by angle brackets with the elements separated by commas. The first element of the tuple is the formal parameter names, printed in the same format as a list. The second element of the tuple is the environment, printed in the same format as a binding; the elements of the binding may be partially or completely suppressed and replaced by .... The third element of the tuple is the body. If the body is a syntactic form it will be printed as a string of characters; if it is an opaque value it will be printed with the opaque value format. The last element of the tuple is the defaulting flag; it will be printed as a boolean value. Given the expression:

  ```
  LAMBDA a, b IN PLUS(a, b)
  ```

  an external representation of its value is:

  ```
  < (a, b), {...}, PLUS(a, b), FALSE>
  ```

- If **type** = **IntegerT**, the external representation is the same as the expression syntax for a numeric expression, with an optional "-" sign prepended.

- If **type** = **ListT**, the list elements are printed in order, enclosed by parentheses, and separated by commas. Given the expression:

  ```
  (1, 2, 3, 4)
  ```

  an external representation of its value is:

  ```
  (1, 2, 3, 4)
  ```

- If **type** = **OpaqueT**, the external representation is determined by the bridge that produced the value; see the bridge documentation for further information. The opaque values produced by Vesta are the ERROR value and the primitive functions and constants. The external representation of the ERROR value is **ERROR**; the external representations of the primitive functions and constants are the same as their expression syntax.

- If **type** = **TextT**, one of two possible formats will be used for the external representation. The first format uses the same expression syntax as a text literal. The second format is a UID literal, which names an object in the Vesta repository which contains the text value.

4.3.6 Environments

An environment is a list of bindings. Notationally:

```
TYPE Env = List.T (* OF Value *);
```

Although an element of the **Env** list is specified as an arbitrary typed value, in fact, by construction, it will always be a binding.

A new environment is constructed by one of the following functions:

```
PROCEDURE EnvPush(r: Env; b: Value): Env;
  BEGIN
    ASSERT(b^.type = BindingT);
    List.Push(r, b);
    RETURN r;
  END EnvPush;

PROCEDURE EnvAppend(r1, r2: Env): Env;
  BEGIN
    RETURN List.Append(r1, r2)
  END EnvAppend;
```

The **EnvPush** function takes an environment and a binding; it returns a new environment that is created by pushing the binding on the front of the old environment. The **EnvAppend** functions takes two environments; it returns a new environment created by appending the second list to the first.

There is an initial environment **rInit** that is used in the description of **Lookup** below:

```
VAR rInit: Env;
```

The initial environment provides a standard collection of functions and values that can be used in Vesta models. See Section 4.5 for a description of the elements of the initial environment. Note that the names of the elements in the initial environment are not reserved and so may be redefined.

During the evaluation of a model, both a static and a dynamic environment are constructed. See Section 4.4 for a discussion of how the static and dynamic environments are built, and how they are used during evaluation.

Environments are used to map identifiers to typed values via the following function:

```
PROCEDURE Lookup(s, d: Env; N: Text.T): Value;
  PROCEDURE LookupInner(r: Env): Value;
    VAR
      b: Value;
      be: List.T;
    BEGIN
      IF r = NIL THEN RETURN NIL END;
      b := NARROW(List.First(r), Value);
      be := List.Assoc(b^.belems, N);
      IF be # NIL THEN
        RETURN NARROW(List.Second(be), Value)
      ELSE
        RETURN LookupInner(List.Tail(r))
      END;
    END LookupInner;
  VAR result: Value;
  BEGIN
    result := LookupInner(s);
    IF result = NIL THEN
      result := LookupInner(rInit);
    END;
    IF result = NIL THEN
      result := LookupInner(d)
    END;
    IF result = NIL THEN result := VestaError END;
    RETURN result;
  END Lookup;
```

When looking up a name, the environments are searched in the following order: (1) static environment, (2) initial environment, (3) dynamic environment.

When a name is looked up in an environment, a binding is taken from the front of the environment list, and the name is looked up in the binding.

Because bindings may not have more than one value for the same name, the name will be found once, or not at all. If the name is found, the associated **Value** is returned. Otherwise, the next binding on the list is examined in the same way. If every binding on the environment list is examined and the name is not found, then the name does not exist in that environment.

4.3.7 Parsed Expressions

Parsing is defined by the function **Parse**, whose signature is:

```
PROCEDURE Parse(e: Text.T): Expr;
```

Parse returns an **Expr**, which is a variable record with different fields for each expression class:

```
TYPE
  ExprClass = (ApplyE, BindingE, DirectoryE, IfE, IntegerE,
    LambdaE, LetE, ListE, NameE, SelectionE, TextE, TildeE,
    UidE);
  Expr = REF ExprR;
  Exprs = REF ARRAY OF Expr;
  ExprR = RECORD
    CASE kind: ExprClass OF
    | ApplyE:
        func, actuals: Expr;
    | BindingE:
        belems: REF ARRAY OF Exprs;
    | DirectoryE:
        dinput, import: Exprs;
        dbody: Expr;
    | IfE:
        test, then, else: Expr;
    | IntegerE:
        int: INTEGER;
    | LambdaE:
        formals: List.T (* OF Text.T *);
        defaultFormals: BOOLEAN;
        prog: Expr;
    | LetE:
```

```
      linput, lbody: Expr;
  | ListE:
      lelems: Exprs;
  | NameE:
      name: Text.T;
  | SelectionE:
      expr: Expr;
      sname: Text.T;
  | TextE:
      text: Text.T;
  | TildeE:
      tname: Text.T;
      value: Expr;
  | UidE:
      uid: Text.T;
  END;
END;
```

For convenience, **Expr** includes two variants, **TildeE** and **UidE**, that are not expressions in the language. The tilde form may appear only as an element in a binding or directory expression. The uid form may appear only as the right-hand side of a tilde.

4.4 Language Semantics—Evaluation

Evaluation is defined by a single function **Eval**:

```
PROCEDURE Eval(e: Expr; s, d: Env): Value;
BEGIN
  CASE e^.kind OF
  | ApplyE:     RETURN EvalApply(e, s, d);
  | BindingE:   RETURN EvalBinding(e, s, d);
  | DirectoryE: RETURN EvalDirectory(e, s, d);
  | IfE:        RETURN EvalIf(e, s, d);
  | IntegerE:   RETURN EvalInteger(e, s, d);
  | LambdaE:    RETURN EvalLambda(e, s, d);
  | LetE:       RETURN EvalLet(e, s, d);
  | ListE:      RETURN EvalList(e, s, d);
  | NameE:      RETURN EvalName(e, s, d);
```

```
  | SelectionE: RETURN EvalSelection(e, s, d);
  | TextE:     RETURN EvalText(e, s, d);
  | UidE:      RETURN EvalUID(e, s, d);
  END;
```

4.4.1 Names

```
PROCEDURE EvalName(e: Expr; s, d: Env): Value;
  BEGIN
    RETURN Lookup(s, d, e^.name);
  END EvalName;
```

Names are evaluated by searching for the first occurrence of the same name in the current environments, and returning the value associated with that name. If the name is not found in the environments the **ERROR** value is returned.

4.4.2 Literals

```
PROCEDURE EvalText(e: Expr; s, d: Env): Value;
  VAR result: Value;
  BEGIN
    NEW(result, TextT);
    result^.text := e^.text;
    RETURN result;
  END EvalText;
```

Texts are evaluated by returning the sequence of characters associated with the expression form.

```
PROCEDURE EvalInteger(e: Expr; s, d: Env): Value;
  VAR result: Value;
  BEGIN
    NEW(result, IntegerT);
    result^.int := e^.int;
    RETURN result;
  END EvalInteger;
```

Integers are evaluated by returning the signed number associated with the expression form.

```
PROCEDURE EvalUID(e: Expr; s, d: Env): Value;
  VAR result: Value;
  BEGIN
    IF IsModel(e^.uid) THEN
      result := Eval(Parse(Read(e^.uid)), NIL, NIL);
    ELSE
      NEW(result, TextT);
      result^.text := Read(e^.uid);
    END;
    RETURN result;
  END EvalUID;
```

In **EvalUID**, the functions **Read** and **IsModel** are abstractions of the facilities provided by the Vesta repository. **Read** returns the text associated with the UID supplied as its argument. **IsModel** returns TRUE if its argument names a Vesta model and FALSE otherwise.

When a non-model UID is evaluated, the UID is just read; this returns the text stored in the object named by the UID. When a model UID is evaluated, the UID is read; this returns the text stored in the object named by the UID, and this text is then parsed and evaluated to produce the result.

4.4.3 Selection

```
PROCEDURE EvalSelection(e: Expr; s, d: Env): Value;
  VAR
    b: Value;
    be: List.T;
  BEGIN
    b := Eval(e^.expr, s, d);
    IF b^.type # BindingT THEN RETURN VestaError END;
    be := List.Assoc(b^.belems, e^.sname);
    IF be # NIL THEN
      RETURN NARROW(List.Second(be), Value)
    ELSE
      RETURN VestaError;
```

```
    END;
  END EvalSelection;
```

The syntactic form of name selection is **E$N**. The expression **E** must evaluate to a binding value **b**, and the identifier **N** must be one of the identifiers in the binding value. The result of the expression is the value associated with **N** in the binding value **b**.

4.4.4 Lists

```
PROCEDURE EvalList(e: Expr; s, d: Env): Value;
  VAR
    result: Value;
    i: CARDINAL;
  BEGIN
    NEW(result, ListT);
    result^.lelems := NIL;
    FOR i := 0 TO HIGH(e^.lelems^) DO
      result^.lelems :=
        List.Append1(result^.lelems,
          Eval(e^.lelems^[i], s, d))
    END;
    RETURN result;
  END EvalList;
```

The syntactic form **(E1,...,En)** evaluates to a runtime value whose type is **ListT** and whose value is a **List.T**. The elements of the list value are values resulting from the evaluation of **E1** through **En**.

4.4.5 Bindings

```
PROCEDURE EvalBinding(e: Expr; s, d: Env): Value;
  VAR
    result, v, commavalue, b: Value;
    a: List.T;
    commalist: Exprs;
    i, j: CARDINAL;
    snew: Env;
  BEGIN
```

```
    NEW(result, BindingT);
    result^.belems := NIL;
    snew := s;
    FOR i := 0 TO HIGH(e^.belems^) DO
      commalist := e^.belems^[i];
      NEW(commavalue, BindingT);
      commavalue^.belems := NIL;
      FOR j := 0 TO HIGH(commalist^) DO
        IF commalist^[j]^.kind = TildeE THEN
          v := Eval(commalist^[j]^.value, snew, d);
          VestaMerge(commavalue^.belems,
            List.List1(
              List.List2(commalist^[j]^.tname, v)));
        ELSE
          v := Eval(commalist^[j], snew, d);
          IF v^.type = BindingT THEN
            VestaMerge(commavalue^.belems, v^.belems);
          ELSIF (NUMBER(e^.belems^) = 1) AND
              (NUMBER(commalist^) = 1) THEN
            RETURN v;
          ELSE
            RETURN VestaError;
          END;
        END;
      END;
      VestaMerge(result^.belems, commavalue^.belems);
      snew := EnvPush(snew, commavalue);
    END;
    RETURN result;
  END EvalBinding;
```

There is no requirement that the syntactic form **E** evaluate to a binding value. Curly braces may be used for grouping in addition to being used as the binding constructor. If curly braces are used for grouping, the expression **E** is evaluated and its result is returned. The rest of this section describes evaluation when curly braces are being used as the binding constructor.

A binding evaluates to a runtime value whose type is **BindingT**. As mentioned in Section 4.3.5, the representation of a **BindingT** value is an association list. The elements of the association list come from the values

resulting from the evaluation of the elements of the binding.

The elements of the binding are evaluated in order, from left to right. The final value is built up by using the Vesta **MERGE** operation on the current intermediate result and the result of each element evaluation. Section 4.5.6 specifies the Vesta **MERGE** operation; briefly, **MERGE** combines the element of two bindings, `b1` and `b2`, into one binding, after removing any elements from `b1` for which an element with the same name exists in `b2`.

As shown in Section 4.2.3, the syntax for a binding is:

```
binding  ::= "{" [ ++( bind-gp ";" ) [ "," | ";" ] ] "}"
bind-gp  ::=   ++( bind-elt "," )
bind-elt ::=   E | N "~" E
```

An individual binding element `bind-elt` either has the tilde form `N ~ E` or is a general expression `E`. These two forms are evaluated differently:

- If the binding element has the tilde form `N ~ E`, then the right-hand side of the tilde is evaluated to produce value `v`. The Vesta **MERGE** operation is performed on the intermediate result and the single element binding value `{N: v}` .
- If the individual binding element `E` is a general expression, then the expression is evaluated to produce value `v`. The value `v` must be a `BindingT` type, and the Vesta **MERGE** operation is performed on the intermediate result and this binding value.

A binding group `bind-gp` is a list of comma-separated binding elements. The bindings groups are evaluated in order, from left to right. Each element of a binding group is evaluated in the same environments, and the result of evaluating a binding group is a `BindingT` value. As each binding group is evaluated its value is pushed onto the static environment. The next binding group is evaluated in this new environment.

The binding elements to the left of the ; are available in the scope of the elements to the right of the ;. The following expression:

```
{a ~ 3; b ~ a}
```

evaluates to the binding value:

```
{a: 3, b: 3}
```

As discussed in Section 4.2.4, semicolon has a lower precedence than comma. Thus, the binding:

```
{a ~ 3; b ~ a, c ~ a; d ~ PLUS(b, c)}
```

defines the same value as:

```
{a ~ 3; {b ~ a, c ~ a}; d ~ PLUS(b, c)}
```

The same name may appear more than once in a binding expression, but the name components of a binding value are distinct. The binding expression:

```
{a ~ 3, a ~ 4; {a ~ 5}, a ~ 6}
```

is legal and evaluates to the binding value:

```
{a: 6}
```

As mentioned in Section 4.3.3, the binding of an identifier cannot be determined in general until runtime. Consider the following model:

```
DIRECTORY
IN
  LET
    {a ~ 3}
  IN
    a
```

In this case, a lexical scan of the model shows that the use of **a** in the last line is associated with the binding of **a** in the fourth line of the model. But consider the following model:

```
DIRECTORY
IMPORT
  sub.v ~ <some uid>
```

```
IN
  LET
    {a ~ 3;
     sub.v$f()}
  IN
    a
```

In this case, a lexical scan of the model cannot reveal which binding of a is associated with the use in the last line. The use of a could be associated with the binding of a in sixth line, but if sub.v$f() returns a binding value with an a component, then the use of a could be associated with the binding of a in the seventh line of the model.

4.4.6 LET expressions

```
PROCEDURE EvalLet(e: Expr; s, d: Env): Value;
  VAR b: Value;
  BEGIN
    b := Eval(e^.linput, s, d);
    IF b^.type # BindingT THEN RETURN VestaError END;
    RETURN Eval(e^.lbody, EnvPush(s, b), d);
  END EvalLet;
```

The LET expression adds new name-value pairs to the static environment in which the LET body is evaluated. In the LET expression LET E1 IN E2, it's an error if E1 doesn't evaluate to a binding. Note that the dynamic environment is not modified by the LET expression.

4.4.7 LAMBDA expressions

```
PROCEDURE EvalLambda(e: Expr; s, d: Env): Value;
  VAR result: Value;
  BEGIN
    NEW(result);
    result^.type           := ClosureT;
    result^.formals        := e^.formals;
    result^.env            := StaticSubset(s, e);
    result^.body           := e^.prog;
    result^.defaultFormals := e^.defaultFormals;
```

```
  RETURN result;
END EvalLambda;
```

The function signature is composed of two parts: a (possibly empty) sequence of formal parameter names, and the optional token ... which follows the formals list. In the formals sequence, all the names must be distinct or an error occurs. The ... token signifies *defaulting* of formal parameters. During the application of a function that permits formals defaulting, a free variable in the function body may be bound from the dynamic environment of the function application. See Section 4.4.8 for a specification of function application.

The result of evaluating the **LAMBDA** expression is a closure with four components. The **formals** component is a list of the formal parameter names. The **env** component saves the static environment of the closure, which is not identical to the static environment of the function definition. The **body** component records the body as a syntactic form; no evaluation of the body occurs until the closure is applied. The **defaultFormals** component indicates whether formals defaulting should occur upon application.

The static environment of a closure is a subset of the static environment in which the **LAMBDA** expression defining the closure appears. Only identifiers that appear literally and are free in the body of the **LAMBDA** expression are included in the static environment of a closure. This peculiar-sounding rule makes it possible for the language implementation to cache closures, improving the speed of evaluation.

This rule can give surprising results. A common case involves the Shell bridge (section 5.5, page 106):

```
LET {myProg ~ M2$Prog((M2$Compile(MyProg.mod), SL-basics))} IN
  { DoMyProg ~ LAMBDA arg, ... IN Shell$Sh("myProg arg")$stdout }
```

The only identifier free in the body of DoMyProg is Shell, which is not present in the static environment. Thus the static environment of DoMyProg is empty. But this clearly wasn't the intent; the expression should read

```
LET {myProg ~ M2$Prog((M2$Compile(MyProg.mod), SL-basics))} IN
  { DoMyProg ~ LAMBDA arg, ... IN
      LET {myProg ~ myProg} IN Shell$Sh("myProg arg")$stdout }
```

In this version the static environment of DoMyProg to contains myProg. Notice that arg does not require this treatment, since, even if it were literally present in the body of DoMyProg, it wouldn't be free.

A variation on the same theme occurs within shell scripts. For example:

```
DIRECTORY
  script.sh ~ <UID>,
  some.file ~ <UID>
IN {
  someFunction ~ LAMBDA arg, ... IN Shell$Sh("script.sh arg")
}
```

where the shell script script.sh contains a reference to some.file. The problem is harder to spot, because no mention of some.file appears in the model outside of the DIRECTORY statement. Nevertheless, it is a reference to the static environment and must be adjusted in the same way, namely

```
someFunction ~ LAMBDA arg, ... IN
  LET {some.file ~ some.file} IN Shell$Sh("script.sh arg")
```

Functions are treated as first-class values (also known as higher-order functions). They may be stored in data structures, passed as arguments, and returned as results. Current models make limited use of higher-order functions, but the language mechanisms are available.

There are several syntactic issues a user should be aware of. All of the function signature except the **LAMBDA** keyword may be omitted, so the following is a legal function definition:

```
LAMBDA IN 3
```

When using defaulting, remember to leave a space between the formal parameter list and the defaulting token; otherwise, the signature may not be what was intended. In the following function signature, there is a single formal parameter named `a...`:

```
LAMBDA a... IN
```

4.4.8 Function application

```
PROCEDURE EvalApply(e: Expr; s, d: Env): Value;
  VAR
    f, a, args, b, vbody: Value;
    elet, elist, eapply, ebody: Expr;
    snew, dnew: Env;
    i: CARDINAL;
  BEGIN
    f := Eval(e^.func, s, d);
    IF f^.type # ClosureT THEN RETURN VestaError END;
    a := Eval(e^.actuals, s, d);
    IF a^.type = BindingT THEN
      (* Convert "E1 E2" to "LET E2 IN E1()" and eval *)
      NEW(elist, ListE); NEW(eapply, ApplyE);
      NEW(elet, LetE);
      elist^.lelems := NIL;
      eapply^.func := e^.func;
      eapply^.actuals := elist;
      elet^.linput := e^.actuals;
      elet^.lbody := eapply;
      RETURN Eval(elet, s, d);
    ELSE
      IF a^.type = ListT THEN
        args := a;
      ELSE
        NEW(args, ListT);
        args^.lelems := List.List1(a);
      END;
      (* Ensure actual param list no longer than formals *)
      IF NUMBER(f^.formals^) <
           List.Length(args^.lelems) THEN RETURN VestaError;
      END;
      NEW(b, BindingT);
      b^.belems := NIL;
      (* First bind supplied actuals. *)
      FOR i := 0 TO List.Length(args^.lelems) - 1 DO
        List.Push(b^.belems,
          List.List2(f^.formals^[i],
            List.Nth(args^.lelems, i)));
      END;
```

```
      (* Now bind any remaining formals from 'd' and 's'.
         Error occurs (in Lookup) if any such formal is not
         found. *)
      FOR i := List.Length(args^.lelems)
            TO HIGH(f^.formals^) DO
        List.Push(b^.belems,
          List.List2(f^.formals^[i],
            Lookup(s, d, f^.formals^[i])))
      END;
      (* Construct new environments *)
      IF f^.defaultFormals THEN
        dnew := EnvAppend(s, d);
      ELSE
        dnew := NIL;
      END;
      snew := EnvPush(f^.env, b);
      (* Evaluate closure body in new environments *)
      TYPECASE f^.body OF
      | Expr(ebody): RETURN Eval(ebody, snew, dnew);
      | Value(vbody): (* invoke the appropriate bridge *)
      END;
    END;
  END EvalApply;
```

The Vesta language uses applicative-order evaluation for function applications; that is, the function and argument are evaluated before the function body.

After the function and actuals argument are evaluated, the environments are constructed in which the function body is to be evaluated. Free variables in the function body are bound from these environments. The way that the environments are constructed, and therefore the way the free variables are bound, depends upon the signature of the function.

First, a free variable may be named in the formal parameter list of the function signature. If it is, there must be a corresponding actual (or defaulted actual) to match the formal. Second, a free variable not named in the formal parameter list may be bound from the static environment of the function definition. Last, a free variable not named in the formal parameter list and not bound from the static environment may be bound from the dynamic environment of the function application (if the function signature permits it). If none of these ways provides a binding for the free

variable, it is an error. Each of these options will be discussed next, in the order in which they are attempted by the Vesta evaluator.

- A free variable may be named in the formal parameter list of the function. This case is the problem of binding actuals to formals. Depending upon the type and structure of the actuals, actual arguments may be bound by name or by position. If there is not a corresponding actual for a formal, then the actual is defaulted; that is, an actual argument with the same name as the formal is taken from the current environments.

  If the actuals evaluate to a binding, then the elements of the bindings are bound by name to the corresponding formals. Superfluous actuals in the binding value that are not matched with any formal are ignored.

  If there is a formal name that is not found in the argument binding, the actual value with be defaulted, and the closest value with the same name will be taken from the application environments. If a value with that name is not found in the environments, it is an error. The following examples show argument matching by name; the result of evaluating the expression:

```
LET
  {f ~ LAMBDA a,b,c IN MINUS(PLUS(a,b), c),
   a ~ 3}
IN
  f {c ~ 5, b ~ 4}
```

  is 2.

  If the arguments evaluate to a list, then the list is "taken apart" and each element of the list is bound by position to the corresponding formal. If there are more elements in the list then there are formals, it is an error. The following example shows matching by position; the result of evaluating the expression:

```
LET
  {f ~ LAMBDA a,b IN PLUS(a,b)}
IN
  f (1, 2)
```

  is 3.

If there are more formals then there are actuals in the list, the remaining formals are defaulted; that is, they are matched by name from the application environment. If the required formal is not found in the environment, it is an error. The result of evaluating the expression:

```
LET
  {f ~ LAMBDA a, b, c IN MINUS(PLUS(a, b), c);
   c ~ 2}
IN
  f (3, 4)
```

is 5.

The following fragment shows a common mistake; the function expects one actual parameter, which must be a list:

```
LET
  {f ~ LAMBDA l IN FIRST(l)}
IN
  f (1, 2, 3)
```

This example will produce an error when evaluated, since there are more list elements than formals. The example may be rewritten to produce a legal fragment:

```
LET
  {f ~ LAMBDA l IN FIRST(l)}
IN
  f ((1, 2, 3))
```

The result of evaluating this expression is 1.

If the actual argument produces neither a list nor a binding value, the actual is bound by position to the first formal. If there are no formals, it is an error. If there is more than one formal, the remaining formals are matched by name from the application environments. If the required formal is not found in the environments, it is an error. Consider the following expression:

```
LET
  {f ~ LAMBDA a, b IN PLUS(a, b),
   b ~ 6
IN
  f 5
```

In this case, the value 5 is bound to the formal a, and the value 6 is bound to the formal b.

- A free variable in the function body that is not named in the formal parameter list may be bound from the static environment of the function definition. For example:

```
LET
  {a ~ 7,
   b ~ 8;
   f ~ LAMBDA IN PLUS(a, b)}
IN
  f ()
```

When the function body is executed, the value 7 is bound to the free variable a and the value 8 is bound to the free variable b.

- A free variable in the function body that is not named in the formal parameter list and not bound from the static environment may be bound from the dynamic environment of the function application, if the function signature contains the token ...:

```
LET
  {f ~ LAMBDA ... IN PLUS(a, b),
   a ~ 9,
   b ~ 10}
IN
  f ()
```

When the function body is executed, the value 9 is bound to the free variable a and the value 10 is bound to the free variable b.

If the function signature does not contain the token ..., then free variables in the function body may not be bound from the dynamic environment of the function application. For example, the following program is illegal because x is an undefined free variable when the function f1 is applied:

```
DIRECTORY
IN
  LET
    {f1 ~ LAMBDA ... IN x;
     f2 ~ LAMBDA IN f1()}
  IN
    LET {x ~ 5} IN f2()
```

This program may be rewritten to execute without errors by adding the token ... to the function signature of **f2** as follows:

```
DIRECTORY
IN
  LET
    {f1 ~ LAMBDA ... IN x;
     f2 ~ LAMBDA ... IN f1()}
  IN
    LET {x ~ 5} IN f2()
```

Recursive functions may be written using the ... feature. The following fragment:

```
LET
  {fact ~ LAMBDA x ... IN
             IF EQ(x, 0) THEN
               1
             ELSE
               TIMES(x, fact (MINUS(x, 1))) }
IN
  fact 5
```

computes the factorial of 5.

As the specification in this section shows, if the actuals argument **E2** evaluates to a binding value then the function application:

```
E1 E2
```

is equivalent to the expression:

```
LET E2 IN E1()
```

4.4.9 Conditional expressions

```
PROCEDURE EvalIf(e: Expr; s, d: Env): Value;
  VAR test: Value;
  BEGIN
```

```
    test := Eval(e^.test, s, d);
    IF test^.type # BooleanT THEN
      RETURN VestaError;
    ELSIF test^.bool = TRUE THEN
      RETURN Eval(e^.then, s, d);
    ELSE
      RETURN Eval(e^.else, s, d);
    END;
  END EvalIf;
```

A conditional expression must have both the THEN arm and the ELSE arm. If the expression `E1` does not evaluate to a boolean value it's an error. If the value of `E1` is `TRUE`, the THEN arm is evaluated; otherwise, the ELSE arm is evaluated.

4.4.10 DIRECTORY clause

```
PROCEDURE EvalDirectory(e: Expr; s, d: Env): Value;
VAR
  b, v: Value;
  exprs: Exprs;
  tilde: Expr;
  i: CARDINAL;
BEGIN
  b^.type   := BindingT;
  b^.belems := NIL;
  FOR i := 0 TO HIGH(e^.dinput^) DO
    tilde := e^.dinput^[i];
    v := Eval(tilde^.value, s, d);
    IF (List.Assoc(b^.belems, tilde^.tname) # NIL) THEN
      RETURN VestaError;
    END;
    List.Push(b^.belems, List.List2(tilde^.tname, v));
  END;
  FOR i := 0 TO HIGH(e^.import^) DO
    tilde := e^.import^[i];
    v := Eval(tilde^.value, r);
    IF (List.Assoc(b^.belems, tilde^.tname) # NIL) THEN
      RETURN VestaError;
    END;
```

```
      List.Push(b^.belems, List.List2(tilde^.tname, v));
    END;
    RETURN Eval(e^.dbody, EnvPush(s, b), d);
  END EvalDirectory;
```

The directory expression must be the top-level expression in a model, and may only appear as the top-level expression (see Section 4.2.3). There are two sets of entries before the body of the expression: (1) the entries that appear after the **DIRECTORY** keyword, and (2) the entries that appear after the **IMPORT** keyword.

The only syntactic forms that may appear in the DIRECTORY and IMPORT entries are sequences of tilde expressions separated by commas. The left-hand side of the tilde expression is the name **N** and the right-hand side is a UID constant, which evaluates to **v**. The result of evaluating an entry is the name-value pair **(N, v)**. The same name may not appear more than once in a DIRECTORY expression.

Both DIRECTORY and IMPORT entries are evaluated identically. The result of evaluating the entries in each is a binding which contains one **(N, v)** pair for each entry. The resulting binding is then pushed onto the current static environment, (the dynamic environment is unchanged), and the DIRECTORY body is evaluated in these new environments.

Although DIRECTORY and IMPORT entries are evaluated identically, tools that process models may treat the two differently so it's important to put entries in the proper category. Both sets of entries list sources that may be used in the body of the model. The DIRECTORY entries list sources that are local to the package that the model describes, e.g., local definition modules, local implementation modules, and submodels that describe subcomponents of the package. The IMPORT entries list sources that are non-local to the package, e.g., models of other packages that may be referenced from the local model.

A self-referential model; that is, a model that names itself in a DIRECTORY or IMPORT entry, is not permitted.

4.5 Built-in functions and values

Vesta's collection of built-in functions and values are collected into an environment called the the initial environment (Section 4.3.6.)

The names of built-in functions and values are not reserved. To keep your models as readable as possible you should not redefine the names of built-in functions. One way to avoid redefining them is to include at least one lower-case letter in each identifier you define.

All of Vesta's built-in functions operate on typed values and produce typed values. In many cases, the function domains (and results) are restricted to values of a particular type. To make this clear, the descriptions below often specify the particular type rather than Value, for example

```
PROCEDURE PLUS(l: IntegerT; r: IntegerT): IntegerT;
```

instead of

```
PROCEDURE PLUS(l: Value; r: Value): Value
  where l^.type = r^.type = result^.type = IntegerT
```

The Vesta built-in functions originally had lower case names: **cat** instead of **CAT**. The names were changed to upper case to reduce name conflicts. The language evaluator still accepts the original names, but you should change to the new names as you edit old models.

4.5.1 Boolean functions and values

```
PROCEDURE NOT(r: BooleanT): BooleanT;
```

The result is the logical complement of **r**.

```
PROCEDURE AND(l: BooleanT; r: BooleanT): BooleanT;
```

The result is the value **TRUE** if and only if both **l** and **r** are **TRUE**. This function doesn't necessarily short-cut; it may evaluate both its arguments before testing the first one.

```
PROCEDURE OR(l: BooleanT; r: BooleanT): BooleanT;
```

The result is the value **TRUE** if at least one of **l** or **r** is **TRUE**. This function doesn't necessarily short-cut; it may evaluate both its arguments before testing the first one.

```
TRUE
FALSE
```

The primitive values **TRUE** and **FALSE** represent the eponymous boolean values.

4.5.2 Arithmetic functions

```
PROCEDURE PLUS(l: IntegerT; r: IntegerT): IntegerT;
```

The result is the sum of **l** and **r**.

```
PROCEDURE MINUS(l: IntegerT; r: IntegerT): IntegerT;
```

The result is **l** minus **r**. In the subtraction, **l** is the minuend and **r** is the subtrahend.

```
PROCEDURE TIMES(l: IntegerT; r: IntegerT): IntegerT;
```

The result is the product of **l** and **r**.

```
PROCEDURE DIV(l: IntegerT; r: IntegerT): IntegerT;
```

The result is **l** divided by **r**, truncated towards zero. In the division, **l** is the dividend and **r** is the divisor.

```
PROCEDURE MOD(l: IntegerT; r: IntegerT): IntegerT;
```

The result is the value **MINUS(l, TIMES(r, DIV(l, r)))**.

4.5.3 Functions on Texts

```
PROCEDURE SUBTEXT(
  t: TextT; start: IntegerT; length: IntegerT): TextT;
```

The result is a text consisting of the **length** characters in **t** that begin at the character position **start**.

If **start** >= **LENGTH(t)**, the result is an empty text. If **length** > **LENGTH(source) - start**, the result is a text composed of the characters of **t** from the index **start** to the end of **t**.

```
PROCEDURE FIND(
  t: TextT; start: IntegerT; pattern: TextT;
  ignoreCase: BooleanT): IntegerT;
```

This function finds **pattern** in **t** following **start**. More precisely, the result is the smallest character position **i** such that **i >= start** and **EQ(pattern, SUBTEXT(t,i,LENGTH(pattern)))**. If **ignoreCase** is **TRUE**, then lower case characters are treated as upper case characters in both **pattern** and **t** .

If **LENGTH(pattern)** = **0**, the match occurs with **i = start**. If **start** >= **LENGTH(source)**, or no such index **i** exists, the result is **-1**.

```
PROCEDURE REPLACE(source: TextT; start: IntegerT;
  length: IntegerT; t: TextT): TextT;
```

The result is a text with the same characters as **source** except that the **length** characters of **source** beginning at character position **start** are replaced by the characters of **t**.

If **start** >= **LENGTH(source)**, the result is **CAT((source, t))**. Otherwise, if **length** > **LENGTH(source) - start**, the result is **CAT((SUBTEXT(source, 0, start), t))**.

```
PROCEDURE CAT(x: ListT): TextT;
```

If **x** is an empty list, the result is an empty text. Otherwise, **x** may be a general list structure, but all the leaves of the structure must be of type **TextT**. In this case, the result is constructed by concatenating all of the leaves in a depth-first, left-to-right traversal of the structure.

```
PROCEDURE INTEGER_FROM_TEXT(t: TextT): IntegerT;
```

Converts the text `t` into an integer.

4.5.4 Polymorphic functions

```
PROCEDURE TO_TEXT(x: Value): TextT;
```

Returns a text representation of value `x`.

```
PROCEDURE UNION(x1, x2: Value): Value;
```

`x1` and `x2` may both be lists, in which case the result is the list union of `x1` and `x2`. `x1` and `x2` may both be bindings, in which case the result is a binding consisting of all the members of `x1` and `x2`. It is an error for there to exist `(N, Value)` in `x1` and `(N, Value')` in `x2` such that `NE(Value, Value')`.

```
PROCEDURE DIFF(x1, x2: Value): Value;
```

`x1` and `x2` may both be lists, in which case the result is the list contains all the elements in `x1` not contained in `x2`. `x1` and `x2` may both be bindings, in which case the result is a binding consisting of all the members of `x1` for which a member with the same name does not appear in `x2`. That is, for all `(n, v)`,

```
(n, v) IN DIFF(x1,x2) =>
  ((n, v) IN x1) AND NOT ((n, v') IN x2)
```

```
PROCEDURE IS_SUBSET(x1, x2: Value): BooleanT;
```

`x1` and `x2` may both be lists, in which case the result is `TRUE` if `x2` is a subset of `x1`. `x1` and `x2` may both be bindings, in which case the result is `TRUE` iff, for all `(n, v)`,

```
(n, v) IN x1 => (n, v) IN x2
```

Thus, if x1 and x2 are bindings, IS_SUBSET(x1, x2) implies IS_IN(x1, x2) but not conversely.

```
PROCEDURE LENGTH(x: Value): IntegerT;
```

x may be of type ListT, TextT, or BindingT. The result is always non-negative. When x is a list, the result is the number of elements in x. When x is a text, the result is the number of characters in x. When x is a binding, the result is the number of binding elements in x.

```
PROCEDURE NTH(x: Value; n: IntegerT): Value;
```

x may be a list or text (but not a binding since they are unordered). When n is in the range [0..LENGTH(x)), the result is a list element or a (single-character) text. When n is out of range it is an error and the result is the ERROR value.

```
PROCEDURE LT(l: Value; r: Value): BooleanT;
PROCEDURE LE(l: Value; r: Value): BooleanT;
PROCEDURE GT(l: Value; r: Value): BooleanT;
PROCEDURE GE(l: Value; r: Value): BooleanT;
```

l and r may be either both of type IntegerT or both of type TextT. If both are of type TextT, a lexicographic comparison of the character sequences l and r is done, using the ordering of the ASCII character set.

LT is TRUE if l is less than r; LE is TRUE if l is not greater than r; GT is TRUE if l is greater than r; GE is TRUE if l is not less than r.

```
PROCEDURE EQTYPE(l: Value; r: Value): BooleanT;
```

This function is defined for all types. EQTYPE returns TRUE iff l^.type = r^.type. Note that this implies a TRUE result if both arguments are closures, regardless of differences in their domains. Note also that if both arguments are of type OpaqueT, then EQTYPE(l,r) is TRUE if and only if l^.bridge = r^.bridge.

```
PROCEDURE NEQTYPE(l: Value; r: Value): BooleanT;
```

This function is defined for all types. **NEQTYPE** is defined as **NOT(EQTYPE(l,r))**.

```
PROCEDURE EQ(l: Value; r: Value): BooleanT;
```

This function is defined for all types. **EQ** requires **EQTYPE(l,r)**; otherwise, an error occurs and the result is the **ERROR** value. Two integers are **EQ** if they have the same integer value. Two texts are **EQ** if the contain the same characters. Two booleans are **EQ** if the have the same boolean value. Two bindings are **EQ** if they define the same set of names and, for each name in that set, the values bound to the name in the two bindings are **EQ**. Two lists are **EQ** if they are structurally identical and their leaves are **EQ**. Two opaque objects are **EQ** if they were manufactured by the same bridge and that bridge says they are equal. Two closures are **EQ** if (1) they have identical static environments (in the sense of Lisp EQ) and (2) either the closures result from the evaluation of the same lexical occurrence of a **LAMBDA** expression or both their bodies are opaque values and the opaque objects are **EQ**.

```
PROCEDURE NE(l: Value; r: Value): BooleanT;
```

This function is defined for all types. **NE** is defined as **NOT(EQ(l,r))**.

4.5.5 Functions on Lists

```
PROCEDURE CONS(x: Value; l: ListT): ListT;
```

The result is a list; the first element of the result list is **x** and the remaining elements are the elements of **l**.

```
PROCEDURE APPEND(l1, l2: ListT): ListT;
```

Appends two lists together, returning the new list.

```
PROCEDURE REVERSE(l: ListT): ListT;
```

Reverses a list.

```
PROCEDURE SORT(l: ListT; f: ClosureT): ListT;
```

Sorts a list in ascending order. The closure **f** is used to compare two elements in the list. Its signature is

```
f(v1, v2: Value): IntegerT.
```

If **v1** is less than **v2** then **f** returns a negative value; if **v1** equals **v2** then **f** returns zero; if **v1** is greater than **v2** then **f** returns a positive value.

The implementation is time- and cons-efficient but is not stable.

```
PROCEDURE MEMBER(l: ListT; x: Value): BooleanT;
```

Returns **TRUE** if for some **y** at the top level of list **l**, **EQ(y, x)**; returns **FALSE** otherwise.

```
PROCEDURE FIRST(l: ListT): Value;
```

The result is the first element of the list. If **l** is an empty list it is an error and the result is the **ERROR** value.

```
PROCEDURE FIRST_N(l: ListT; n: IntegerT): ListT;
```

A new list consisting of the first **n** elements of **l**. Undefined if **n** $<$ 0 or **n** $>$ **LENGTH(l)**.

```
PROCEDURE TAIL(l: ListT): ListT;
```

The result is a list with the first element removed. If **l** is an empty list it is an error and the result is the **ERROR** value.

```
PROCEDURE NTH_TAIL(l: ListT; n: IntegerT): ListT;
```

The result of applying **TAIL** to **l**, n times. Undefined if **n** $<$ 0 or **n** $>=$ **LENGTH(l)**.

```
PROCEDURE NO_DUPLICATES(l: ListT): ListT;
```

Returns a new list with all the duplicate elements removed.

```
PROCEDURE INTERSECTION(l1, l2: ListT): ListT;
```

Returns the intersection of `l1` and `l2`.

```
PROCEDURE DELETE(l: ListT; x: Value): ListT;
```

Returns a list containing all the elements **elem** of `l` that are not **EQ(elem, x)**.

```
PROCEDURE REDUCE(l: ListT; x: Value; f: ClosureT): Value;
```

The signature of **f** is

```
f(v1, v2: Value): Value.
```

If `l` has elements `l1`, `l2`, ..., `ln`, **REDUCE** evalutes to

```
f(...f(f(x, l1), l2)...,ln)
```

or **x** if `l` is empty.

```
PROCEDURE MAP(l: ListT; f: ClosureT): ListT;
```

The signature of **f** is

```
f(v: Value): Value.
```

MAP applies **f** to each element of `l`, returning the list of results. Equivalent to

```
REDUCE(l, (), LAMBDA v1, v2 IN APPEND(v1, (f(v2)))).
```

```
PROCEDURE FLATTEN(l: ListT): ListT;
```

If `l` is an empty list, the result is an empty list. Otherwise, `l` may be a general list structure. In this case, the result is a flat list, with the elements of the list produced by a depth-first, left-to-right traversal of the structure.

```
PROCEDURE ASSOC(l: ListT; x: Value): ListT;
```

The list `l` is an association list; a possibly-empty list whose top-level elements are varying-length non-empty lists of the form:

```
(key val1 val2 ... valn)          n >= 0.
```

Returns the first such list for which **EQ(key, x)**. Returns `()` if no such tuple is found.

```
PROCEDURE ASSOC_LIST_TO_BINDING(l: ListT): BindingT;
```

Each top-level element of an association list is a 2-element list; the first element is a text and the second may have any type.

If `l` is an empty list, the result is an empty binding. Otherwise, the list is converted to a binding. There may be more than one element in the association list with the same text value; only the last element of the association list with the same text value appears in the resulting binding value.

4.5.6 Functions on Bindings

```
PROCEDURE MERGE(b1: BindingT; b2: BindingT): BindingT;
```

The result is defined as **UNION(DIFF(b1, b2), b2)**. Thus, if the bindings `b1` and `b2` have a name `n` in common, `b2`'s value for `n` is associated with `n` in **MERGE(b1, b2)**.

```
PROCEDURE IS_IN(b1: BindingT; b2: BindingT): BooleanT;
```

The result is **TRUE** if and only if, for all **(n, v)**,

```
(n, v) IN b1 => ThereExists v' SuchThat (n, v') IN b2
```

```
PROCEDURE BINDING_TO_ASSOC_LIST(b: BindingT): ListT;
```

Returns an association list containing the same information as the binding. An association list is a list of two-element lists. The first element of each two-element list is the text name of an element of the binding; the second element is the corresponding value, which may have any type.

```
PROCEDURE BINDING_TO_NAME_LIST(b: BindingT): ListT;
```

Returns a list of the text names in the binding **l**. If **l** is an empty binding, the result is an empty list.

4.5.7 Functions on models

```
PROCEDURE DIRECTORY_OF(m: Value): BindingT;
```

The parameter **m** must be the value produced from evaluating a model. The result is a binding value with an element for each entry in the directory portion of the model.

```
PROCEDURE IMPORTS_OF(m: Value): BindingT;
```

The parameter **m** must be the value produced from evaluating a model. The result is a binding value with an element for each entry in the imports portion of the model.

4.5.8 Miscellaneous functions

```
PROCEDURE SELF(): TextT
```

The result is the UID of the enclosing model as a text value.

```
PROCEDURE ERROR(t: TextT): ErrorT
```

The result is the **ERROR** value, and the text **t** is written on the Vesta evaluator message reporting stream.

Vesta is a purely functional language, and there is traditionally some difficulty integrating the idea of I/O with a functional language. An expression in a functional language is evaluated to generate a value, and not for its side-effects (such as I/O). Furthermore, a functional value may be produced by evaluating expressions in some other order than they appear in the program, whereas I/O implies sequencing.

To deal with these concerns, the **ERROR** function is treated applicatively like all other functions in the language. The **ERROR** function may not be evaluated at all, or it may be evaluated more than once, and it may not be executed in the obvious sequential order.

```
PROCEDURE FORCE(x: Value): Value
```

The **FORCE** primitive completely evaluates any lazy (previously unevaluated) components of its argument, and returns the now fully-evaluated value **x**. **FORCE** is used, for instance, in the luke model to ensure that the Lukify program is compiled and linked when build is applied during a release, rather than when Luke$MakeSoft is invoked.

```
PROCEDURE PRAGMA(t: TextT): Value
```

The **PRAGMA** primitive is used to change Vesta's behavior in miscellaneous ways, as a function of the parameter **t**.

PRAGMA("umbrella") returns the empty binding and causes Vesta not to read caches of imported models while evaluating the model containing the **PRAGMA("umbrella")** call. This is useful in evaluating umbrella models such as the buildingenv and release models.

```
PROCEDURE BRIDGE(b: OpaqueT): BindingT;
```

The parameter **b** is a program that implements a bridge. The result of **BRIDGE** is a two-element binding; binding elements are named **exports** and **versionForClass**.

The value of **exports** is a binding which provides the bridge facilities that a client may use in models. Typically, a bridge will export functions, and it will also export default parameters which may be used (or overridden) when one if its functions is applied. The functions are represented as functions whose bodies are opaque values; the parameters are bindings. The documentation for each bridge describes the facilities provided by the bridge.

The value of **versionForClass** is a single-element binding which binds the *bridge class* to a specific version of the bridge. A bridge class is a group of bridges that all implement the same semantics and have the same value for **exports**; the **versionForClass** binding tells which version of the bridge from the bridge class should be used when evaluating the client model.

In general, the **b** parameter will be an expression involving a model that defines the construction of a bridge. This construction usually involves other bridges.

4.5.9 Wizard functions

```
PROCEDURE NEW_BINDING()
PROCEDURE NEW_OPAQUE()
PROCEDURE NEW_OPAQUE_CLOSURE()
```

These three functions are provided for the exclusive use of the vcapture program. See Chris Hanna if you are writing a program like vcapture.

4.6 LANGUAGE EXAMPLES

Need some.

4.7 The Vesta Evaluator

The Vesta evaluator is an interpreter; it executes the commands in the model as it evaluates the model. While a model is being interpreted the software system that the model describes is being built. Type-checking is

done during model execution, and error and informational messages are written to a message reporting stream.

A client does not need to know about the design of the evaluator to write and/or evaluate models. However, some information may help a client understand performance when building a software system. These issues are discussed below.

4.7.1 Lazy evaluation

With lazy evaluation, a value is computed only when it is needed (and then it is computed only once). In a general-purpose functional language, lazy evaluation adds to the expression power of the language, allowing, for example, the construction and manipulation of *infinite data structures* and *streams*.

Vesta's use of lazy evaluation is more restricted; the Vesta evaluator employs lazy evaluation only when evaluating the elements of a binding that have the syntactic form **N ~ E**. In this case, the right-hand side **E** of the binding element is evaluated at most one time, and is not evaluated at all until the name **N** on the left-hand side of the binding element is referenced for the first time.

There is one instance when the Vesta evaluator is not lazy: the value resulting from an evaluation is evaluated fully. For example, consider the model:

```
DIRECTORY
IN
  LET {
    a ~ PLUS(1, 2),
    b ~ PLUS(3, 4)
    }
  IN {
    result1 ~ PLUS(a, 6),
    result2 ~ PLUS(a, 8)
    }
```

During the evaluation of this model, the **PLUS** primitive will be called three times: once with arguments **1** and **2**, once with arguments **3** and **6**, and once with arguments **3** and **8**. Note that **PLUS** is not invoked with

arguments 3 and 4, since this value is not required to compute the final value. The result of evaluating this model is the binding value:

```
{result1: 9, result2: 11}
```

Lazy evaluation of bindings is quite useful when evaluating models. For example, consider the following model fragment:

```
DIRECTORY
...
IN {
  ...,
  build1 ~ ...,
  build2 ~ ...,
  build3 ~ ...,
}
```

With lazy evaluation, a user may evaluate one of the **build** components in the model without evaluating all of them.

4.7.2 Caching

With the Vesta language (as with all functional languages), the result of evaluating a function is completely dependent on the inputs to the function; if two applications use the same function and inputs they will produce the same result. The Vesta evaluator takes advantage of this fact to cache the results of function applications for later evaluations.

Caching function application results is a crucial performance enhancement during model evaluation. For instance, a typical application model imports a version of buildingenv and references a shared library, say **SL-ui**, from **buildingenv.v$Env-default()**. When producing this buildingenv it may have taken ten minutes to link this shared library and possibly hours to build the librariy from sources if a low-level interface changed. Vesta would not be practical to use for developing large systems without caches to bypass such lengthy computations after they have been performed once.

A Vesta cache is a file containing a set of cache entries for different function applications. Each evaluation creates a cache in memory. At the

end of the evaluation, Vesta writes the cache to disk and associates it with the top-level model in the evaluation. Cache entries for function applications in models referenced from the top-level model appear in the cache of the top-level model. If a cache entry from an imported model was used in the evaluation, the cache for the evaluation contains a reference to the cache of the imported model, not a copy of the imported cache entry.

When evaluating a model that does not have its own cache, such as a new model, the evaluator uses caches from *related* models. The evaluator maps the UID of the model to a package name (RNPN), then works backward from that RNPN by decrementing the final number (or, if the final number is zero, removing it and the preceding branch name). As it visits each model it looks to see if it has an associated cache. It stops as soon as it finds a cache, or when it has looked at a certain number of models.

This explains why the UseVesta `Grab` operation is not equivalent to file copying. `Grab` both copies the file contents and preserves the Vesta UID. So after a `Grab` the evaluator can find the cache associated with the model.

Sometimes, vrweed will delete a derived object from the public repository that is referenced by a cache in your private repository. When this happens evaluations that read this cache may fail, because the evaluator does not check for the existence of the derived object at the time it gets a cache hit. By the time the missing object is looked up, the information on how to create it has been forgotten. To work around this problem you should comment out the component of your model that actually performs the building (usually "test"), evaluate the model, remove the comment, and evaluate the model again. The first evaluation produces an empty cache. The second evaluation finds the empty cache and therefore performs the evaluation from scratch, creating the missing derived and a new cache.

Caching can take place at other levels of the Vesta system. The Vulcan bridge is the only example of this at present. Vulcan uses the Vesta repository as a cache. A derived object produced by a function application is named by the inputs to the application. Before Vulcan creates a new derived object it looks to see if the object already exists in the repository. If the object does exist, the bridge does not recreate the object but simply returns the existing object. The Vulcan bridge calls this a repository cache hit.

The evaluator's general strategy for caching the results of function applications works well in most cases, but is not perfect. For instance, if you check in a version of some package without evaluating it, that version

will never have a cache associated with it. If several people then perform similar evaluations (e.g. build with overrides) involving the cacheless version, the version will be evaluated over and over. Fortunately for Vulcan users, repository cache hits will speed up the repeated evaluations.

Occasionally, you will attempt to evaluate a model that imports an old buildingenv whose cache and deriveds have been weeded (and the caches and deriveds for all previous versions of buildingenv have also been weeded). Since one version of buildingenv imports the previous version of buildingenv, this leads to a "garden-of-eden" rebuild: First we build buildingenv.1, then buildingenv.2, and so forth. This is the theory, anyway, but it is foiled by practical considerations since the Vesta evaluator overflows its stack and dumps core while carrying out this process.

The evaluator prints a warning message if no cache is found for an imported root model in the buildingenv family. If you notice this message you should stop the evaluation and update your buildingenv import. If the Vesta server dumps core, check the .TYPESCRIPT file to see if Vesta issued this warning.

Earlier, we said that at the end of the evaluation, Vesta writes the cache to disk and associates it with the top-level model in the evaluation. This happens even if the evaluation encountered errors or was stopped before completion by human intervention. The evaluator writes out the cache after returning control to the user interface (UseVesta or vmake.) Even though the evaluator is still working (in background) to write the cache out, it can begin new evaluations if they don't require the previous cache.

Usually, you don't need to be aware of any of this. However, if you want to kill the Vesta server or plan to boot your machine, it's usually best to wait until the server is idle. Unfortunately, there isn't any visible indication at present. If you need help in determining whether the server is idle, contact a Vesta implementor.

4.7.3 Checkin evaluation

Vesta user tools, such as UseVesta and vmake, always perform an evaluation before doing a checkin. This evaluation is performed to ensure that the model has been evaluated, and because of caching should be quite fast if the model was previously evaluated.

(Historical note: In older versions of Vesta, checkin evaluation needed to examine every new derived produced during the checkout session to update any recipes contained in the derived. This evaluation could take almost as long as evaluating the entire package from scratch.)

5. Bridge and utility functions

This chapter documents the functions exported by various bridges, and utility functions that are part of buildingenv. Functions are described in the notation used for describing the Vesta language's built-in functions in section 4.5, page 65. The token ... at the end of a function signature means that the function can access the dynamic environment of the function's application. This mechanism is used, for instance, by the Modula-2+ **Compile** function to bind imported definitions modules, and by the C **Compile** function to bind include files needed by the C preprocessor.

5.1 The Vulcan bridge

The environment produced by buildingenv includes the name **M2**, which is a binding of functions implemented by the Vulcan Modula-2+ bridge. Default values for many of the parameters to these functions are also bound in the environment.

The Vulcan bridge is described in a separate document, which you can obtain by typing "readdoc vulcanbridge" to a shell. Unfortunately, the Vulcan bridge document reads more like a specification than like user documentation.

The Vulcan and MM bridges were designed with a high degree of compatibility. If you use only the functions **Compile**, **Bind**, **Prog**, **Flume**, **ToImageText**, and **Bundle**, and you use only the parameters whose names begin with **M2-**, your model is source-compatible with both bridges. Therefore you may find the MM bridge documentation, contained in the following section, useful in learning to use Vulcan. Keep in mind, however, that the bridges are *not* identical. Consult the Vulcan documentation for **M2-standAlone**, which is important in building some Vulcan programs.

5.2 The MM bridge

In some special buildingenvs, the name **M2** is a binding of functions implemented by the Modula-2+ bridge that uses *mm*. Default values for many of the parameters to these functions are also bound in the environment. These special buildingenvs are branches with names like buildingenv-vax.72.mm.0.

Several functions have parameters named **Env-instSet** and/or **Env-platform**, which are nearly always defaulted by the invoker. Default values for these parameters are defined in buildingenv, so users can usually ignore them. The present implementation of the MM bridge requires that **Env-instSet** have the value "VAX" and that **Env-platform** be either "Topaz" or "Ultrix".

5.2.1 Compile

The Compile function has the following signature

```
PROCEDURE Compile(
  source:              TextT;
  MM-enableProfiling: BooleanT;
  M2-checking:         BooleanT;
  MM-loupe:            BooleanT;
  MM-registers:        IntegerT;
  MM-assemblyCode:     BooleanT;
  MM-objectCode:       BooleanT;
  Env-instSet:         TextT;
  Env-platform:        TextT;
  ...
  ): BindingT;
```

source is the program to be compiled. The other arguments translate into **mm** switches as follows:

name	default	mapping to **mm** switch
MM-enableProfiling	**FALSE**	**TRUE** ⇒ -pg
M2-checking	**FALSE**	**TRUE** ⇒ -C
MM-loupe	**TRUE**	**TRUE** ⇒ -G
MM-registers	6	-R<MM-registers>
MM-assemblyCode	**FALSE**	**TRUE** ⇒ -k
MM-objectCode	**TRUE**	**FALSE** ⇒ -S
Env-platform	"Topaz"	"Ultrix" ⇒ -x

Compile reads the module header of its **source** argument and constructs its result binding accordingly. If the header says **DEFINITION MODULE Mumble**, Compile returns a binding of the form

```
{Mumble.d ~ InterfaceT}
```

where **InterfaceT** is a subtype of **OpaqueT** recognizable only to the MM bridge.

If the header says **MODULE Mumble** without a preceding **DEFINITION**, Compile returns a binding that includes

```
{Mumble.o ~ ObjectT}
```

if MM-objectCode is **TRUE**, and

```
{Mumble.s ~ TextT}
```

if MM-assemblyCode is **TRUE**. **ObjectT** is again a subtype of **OpaqueT** recognizable only to the MM bridge.

5.2.2 Foreign

The **Foreign** function is used to convert standard Unix binary files to a form that is acceptable to other functions in the MM bridge. Its signature is:

```
PROCEDURE Foreign(
  object:       TextT;
  Env-instSet:  TextT;
  Env-platform: TextT;
  ...
  ): OpaqueT; (* ObjectT or LibraryT *)
```

object should be a standard Unix ".o" or ".a" file. **Foreign** returns an **ObjectT** if **object** was a ".o" file, and a **LibraryT** (see section 5.2.3, page 86) if **object** was a ".a" file.

Foreign doesn't actually check to see if **object** is a sensible thing to include in a Modula-2+ program. The Vulcan version of **Foreign** is much stricter, and has an incompatible function signature. Clearly, to ensure compatibility Foreign should be used sparingly.

5.2.3 Bind

The **Bind** function constructs a library. Its signature is:

```
PROCEDURE Bind(
  pieces:       ListT;
  Env-instSet:  TextT;
  Env-platform: TextT;
  ...
  ): ListT; (* of LibraryT *)
```

The MM bridge implements a **LibraryT** as an opaque value whose representation includes a Unix archive file. **Bind** scans the **pieces** list, whose elements either may be individual **ObjectT** or **LibraryT** values produced by the MM bridge, or may be bindings or lists containing **ObjectT** and/or **LibraryT** values as their elements. Bind collects all the **ObjectT** values, constructs a single **LibraryT** from them, then returns a list in which this library is first and the remaining elements are the libraries in **pieces** taken in order in which appear. Thus, if **pieces** consists entirely of **ObjectT** values, **Bind** returns a single-element list.

5.2.4 Prog

The **Prog** function constructs an executable program. Its signature is:

```
PROCEDURE Prog(
  pieces:          ListT;
  M2-profiling:    BooleanT;
  MM-loupe:        BooleanT;
  MM-enableGC:     BooleanT;
  M2-metering:     TextT;
  MM-imc:          BooleanT;
  MM-loadAddress:  IntegerT;
  Env-instSet:     TextT;
  Env-platform:    TextT;
  ...
  ): TextT;
```

Elements of **pieces** either may be individual **ObjectT** or **LibraryT** values produced by the MM bridge, or may be bindings or lists containing **ObjectT** and/or **LibraryT** values as their elements. The list is "flattened" and passed to **mm** to produce an executable result. (During flattening, the order of the list and any embedded lists is preserved, but the resulting order of elements taken from a binding is undefined.)

The remaining parameters affect the switches passed to **mm**, as follows:

name	default	mapping to **mm** switch
M2-profiling	**FALSE**	**TRUE** ⇒ -pg
MM-loupe	**TRUE**	**TRUE** ⇒ -G
MM-enableGC	**FALSE**	**TRUE** ⇒ -r
M2-metering	"off"	"brief" ⇒ -pm
		"detailed" ⇒ -pM
MM-imc	**TRUE**	**TRUE** ⇒ -m
MM-loadAddress	0	-T MM-loadAddress
Env-platform	"Topaz"	"Ultrix" ⇒ -x

Bug to be corrected eventually: The loadAddress parameter should be a TextT.

5.2.5 Flume

The signature of the **Flume** function is:

```
PROCEDURE Flume(
  source:                    InterfaceT;
```

```
M2-clientBinding:           TextT;
M2-serverBinding:           TextT;
M2-clientAdjustsByteOrder:  TextT;
M2-serverAdjustsByteOrder:  TextT;
M2-marshalling:             ListT;
M2-rpcProtocol:             IntegerT;
Env-instSet:                TextT;
Env-platform:               TextT;
...
): BindingT;
```

source is the **InterfaceT** to be "flumed". Recall that an **InterfaceT** is not a definition module, but rather the result of calling Compile on a definition module. The distinction might seem insignificant, since the MM representation of InterfaceT contains the same token stream as the definition module that produced it. However, in the Vulcan bridge, this isn't the case – an **InterfaceT** is truly a compiled definition module. For source compatibility in models, MM's Flume function adopts the same approach.

The **Flume** function does its real work by calling the flumeMain(1) program. The four arguments following **source** translate into **flumeMain** switches as follows:

name	default	mapping to **flumeMain** switch
M2-clientBinding	"single"	"single" ⇒ nothing
		"multiple" ⇒ -xc
		"pass-through" ⇒ -xc -l
		"implicit" ⇒ "=ic"
M2-serverBinding	"single"	"single" ⇒ nothing
		"multiple" ⇒ -xs
M2-clientAdjustsByteOrder	"never"	"never" ⇒ nothing
		"asNeeded" ⇒ -bc
		"always" ⇒ -ac
M2-serverAdjustsByteOrder	"never"	"never" ⇒ nothing
		"asNeeded" ⇒ -bs
		"always" ⇒ -as

Consult the flume(1) manpage for documentation of these switch settings.

The **M2-marshalling** argument is a list of TextT values, which are passed as the explicit marshalling types and procedure names on the flumeMain

command line. This list should have an even number of elements. By default, `M2-marshalling` is empty.

The `M2-rpcProtocol` argument is an integer that specifies the desired version of flume behavior. The intent is that the value bound to this argument identifies the wire protocol version number to be used by the generated stubs. The specified value determines which version of flume to execute. At present, the values 0, 1, and 2 are defined, but, regrettably, we no longer have versions of flume that emit stubs for wire protocols 0 and 1. Consequently, the value 2 should be used for all new interfaces. [There is a backward compatibility kludge that recognizes 0 and 1 and produces an empty binding as the result of the Flume function.] Since the choice of wire protocol is a long-term compatibility issue, no default value is provided for this parameter.

The result binding returned by the invocation of Flume contains the (file) names described in the manpage. Here's a summary. If the definition file supplied as the `source` parameter has the name `Test` in its module header, then the binding will contain the following names:

name	present when
TestClient.def	always
TestClient.mod	always
TestRPC.mod	M2-clientBinding = "multiple" or "implicit"
TestServer.def	always
TestServer.mod	always
TestClientRef.def	Test contains opaque REFs
TestClientRef.mod	Test contains opaque REFs
TestImpl.def	M2-serverBinding = "multiple"

5.2.6 ToText

The `ToText` function allows special purpose programs that understand the structure of Unix binaries to extract them from the opaque values produced by the MM bridge. Its signature is:

```
PROCEDURE ToText(
  opaque: OpaqueT
  ): TextT;
```

`opaque` is an `InterfaceT`, an `ObjectT`, or a `LibraryT`. ToText returns a text value containing the underlying representation of these entities: for

`InterfaceT`, it's a .d file; for `ObjectT`, it's a .o file; for `LibraryT`, it's a .a file.

5.2.7 ToImageText

`ToImageText` resembles `ToText`, but works only on the result of `Prog`. Strictly speaking, it isn't necessary, since `Prog` returns a text. However, this isn't be true for the Vulcan bridge, so systematic use of `ToImageText` to extract bytes in a.out(5) format eases the conversion of models from MM to Vulcan. The signature of `ToImageText` is:

```
PROCEDURE ToImageText(
  image: TextT
  ): TextT;
```

`image` is the output of the `Prog` function. The result of the function is precisely `image`; that is, `ToImageText` is the identity function on values of type text.

5.2.8 Bundle

`Bundle` encapsulates a restricted class of Vesta bindings so that they may be referenced from a running program. This functionality resembles that of the existing Bag utility, but, because Vesta values are immutable, Bundle can encapsulate "pointers" (really, UIDs) in the program, while Bag must copy entire values. Thus, Bundle produces a smaller program image, but increases the access time to the encapsulated data during execution.

The signature of Bundle is:

```
PROCEDURE Bundle(
  rootName:           TextT;
  bundleElems:        BindingT;
  M2-bundleThreshold: IntegerT;
  ): BindingT;
```

`bundleElems` must be a binding, each of whose components is a text value. Bundle returns a two-element binding, with components whose

names are <rootName>.def and <rootName>.mod. As their names suggest, these components are text values that define and implement a Modula-2+ interface named **rootName**.

Here's a fragment of a system model that uses Bundle:

```
DIRECTORY
  bitmap ~ <uid1>,
  textfile ~ <uid2>,
  ThingsUser.mod ~ <uid3>,
IN {
  ...
  LET {
    M2;
    things ~ Bundle("Things", {bm ~ bitmap, text ~ textfile});
    Compile(things$Things.def)
    }
  IN {
    ...
    Compile(things$Things.mod),
    ...
    Compile(ThingsUser.mod)
    ...
    }
 }
```

ThingsUser.mod contains source code that imports both the Things interface and the BundleAccess interface. The latter is part of the standard buildingenv; see the /vesta/proj/bundle package family. The former is produced by the invocation of Bundle and the compilation of the definition module it produces. A code fragment in ThingsUser.mod might be

```
rd := BundleAccess.Open(Things.b, ''text'');
```

which returns a reader (Rd.T) on the bytes bound to uid2 in the model above.

Notice that, because of the rootName parameter to Bundle, a program can have as many bundles as it likes, as long as they have distinct

rootNames. So, in this example, we could have put the bitmap `bm` and the text file `text` in separate bundles if we wished.

The M2-bundleThreshold parameter is a non-negative integer, default 0. If the value being bundled occupies fewer than M2-bundleThreshold bytes, it will be copied into the bundle. Since the default is 0, no values receive the copying treatment unless you explicitly set M2-bundleThreshold.

An application that does not use copying of bundled resources can access its resources even if no Vesta server is running on the machine. This would change if we started using source compression in our repository, and the application's resources were compressed.

5.3 The C bridge

The environment produced by buildingenv includes the name `C`, which is a binding containing the three functions implemented by the C bridge: **`Preprocess`**, **`Compile`**, and **`Prog`**.

- **`Preprocess`** does essentially what *cc -E* does: It runs the C preprocessor.
- **`Compile`** does roughly what *cc -c* does: It compiles a single source file after running the C preprocessor over it. **`Compile`** has a Boolean parameter **`C-preprocess`** that, if **`FALSE`**, omits the preprocessing step.
- **`Prog`** does roughly what *cc* does when given a list of object files: It links an executable.

`Compile` and **`Prog`** have an **`Env-instSet`** parameter, which may be either **`"VAX"`** or **`"R3000"`**. When **`Env-instSet = "R3000"`** the compilation or linking is performed on a 3MAX even though Vesta is running on a Firefly.

The C bridge provides a binding, **`C-defaults`**, containing the default values of the defaultable parameters to **`Preprocess`**, **`Compile`**, and **`Prog`**. You can assume that this binding has been opened in any standard environment that contains the C bridge. For instance, if **`C`** is defined, then **`C-define`** (the first defaultable parameter of **`C$Preprocess`**) is also defined.

5.3.1 Preprocess

```
PROCEDURE Preprocess(
  source:                 TextT;
  C-define:               BindingT;
  C-undefine:             ListT or BindingT;
  C-keepComments:         TextT;
  C-keepLineNumbers:      BooleanT;
  C-allowRecursiveMacros: BooleanT;
  Env-instSet:            TextT;
  Env-platform:           TextT;
  ...
  ): TextT;
```

- **C-define**: The bridge converts each <name, value> pair in this binding to -Dname=value on the *cc* command line. Each value must have type TextT. Default value: {}
- **C-undefine**: If this is a binding, the bridge converts each <name, value> pair to -Uname on the *cc* command line (ignoring the value component of each pair). If this is a list of TextT elements, the bridge converts each list element <elem> to -U<elem> on the *cc* command line. Default value: {}
- **C-keepComments**: This parameter controls the treatment of comments encountered in **source**. If it has the value **"all"**, then both C and C++ style comments are passed through to the output (i.e., the bridge passes -C to cpp). If it has the value **"C++"**, then C-style comments are discarded, but C++-style comments are retained. If it has the value **"none"**, then both C and C++ comments are discarded (i.e., the bridge passes -B to cpp). Default value: **"C++"**.
- **C-keepLineNumbers**: If **C-keepLineNumbers = FALSE** the bridge passes the -P switch on the cpp command line, eliminating line numbers in the output. Default value: **TRUE**.
- **C-allowRecursiveMacros**: If **C-allowRecursiveMacros = TRUE** the bridge passes the -R switch on the cpp command line, causing cpp to tolerate recursive macros. When **C-allowRecursiveMacros = FALSE** cpp treats recursive macros as errors. Default value: **FALSE**.
- **Env-instSet**: **"VAX"** or **"R3000"**.

- **Env-platform**: "Topaz" or "Ultrix".

Preprocess establishes the following default symbols, based on the values of **Env-instSet** and **Env-platform**:

symbol	defined iff
vax	**Env-instSet** = "VAX"
mips	**Env-instSet** = "R3000"
MIPSEL	**Env-instSet** = "R3000"
unix	always defined (!)
ultrix	**Env-platform** = "Ultrix"

If defined, these symbols are bound to the value "1".

When the preprocessor is directed to access an include file, e.g. because the file it is preprocessing contains the line

```
# include <stdio.h>
```

the preprocessor gets a value for the file from the dynamic environment of the **Preprocess** application (recall the ... in the function signature.) The building environment defines **C-env**, a binding containing a set of standard C include files from /usr/include. So to preprocess a C program that includes **stdio.h** your model would say

```
LET { buildingenv.v$Env-default(); C-env } IN
  C$Preprocess(hello.c)
```

By opening the **C-env** binding in the expression above, you provide a value for **stdio.h** when **hello.c** is preprocessed.

5.3.2 Compile

```
PROCEDURE Compile(
  source:                TextT;
  C-optimization:        IntegerT;
  C-standard:            BooleanT;
  C-define:              BindingT;
  C-undefine:            ListT or BindingT;
  C-keepComments:        TextT;
```

```
C-keepLineNumbers:      BooleanT;
C-allowRecursiveMacros: BooleanT;
C-shortFloat:           BooleanT;
C-unsignedChar:         BooleanT;
C-volatileVars:         BooleanT;
C-varargs:              BooleanT;
C-noWarnings:           BooleanT;
C-preprocess:           BooleanT;
C-maxSizeForGP:         IntegerT;
Env-instSet:            TextT;
Env-platform:           TextT;
...
): TextT;
```

- **source**: The source text to be compiled by *cc*.
- **C-optimization**: Controls the level of optimization performed by the compiler. If **Env-instSet = "VAX"** and **C-optimization > 1**, the bridge passes the -O switch on the *cc* command line, If **Env-instSet = "R3000"** and **C-optimization** has value <val> in the range [0..2], the bridge passes the -O<val> switch on the *cc* command line. Default value: **1**.
- **C-standard**: Ignored if **Env-instSet = "VAX"**. If **Env-instSet = "R3000"** and **C-standard = TRUE**, the bridge passes the -std switch on the *cc* command line, Default value: **FALSE**.
- **C-define, C-undefine, C-keepComments, C-keepLineNumbers, C-allowRecursiveMacros**: Ignored unless **C-preprocess = TRUE**. See the description of **C$Preprocess** in section 5.3.1, page 93.
- **C-shortFloat**: If **C-shortFloat = TRUE**, the bridge passes the -f switch on the *cc* command line. Default value: **FALSE**.
- **C-unsignedChar**: Ignored if **Env-instSet = "VAX"**. If **Env-instSet = "R3000"** and **C-unsignedChar = TRUE**, the bridge passes the -unsigned switch on the *cc* command line. Default value: **FALSE**.
- **C-volatileVars**: Ignored if **Env-instSet = "VAX"**. If **Env-instSet = "R3000"** and **C-volatileVars = TRUE**, the bridge passes the -volatile switch on the *cc* command line. Default value: **FALSE**.
- **C-varargs**: Ignored if **Env-instSet = "VAX"**. If **Env-instSet = "R3000"** and **C-varargs = TRUE**, the bridge passes the -varargs switch on the *cc* command line. Default value: **FALSE**.

- **C-noWarnings**: If **C-noWarnings = TRUE** the bridge passes the -w switch on the *cc* command line. Default value: **FALSE**.
- **C-preprocess**: If **C-preprocess = FALSE** the bridge does not run the C preprocessor as part of the application of **Compile**. Default value: **TRUE**.
- **C-maxSizeForGP**: Ignored if **Env-instSet = "VAX"**. If **Env-instSet = "R3000"**, the bridge passes the -G<val> switch on the *cc* command line, where <val> is the value of **C-maxSizeForGP**. This controls which variables are placed in the area accessed by the global pointer register. Default value: **0**; note that this differs from the *cc* default.
- **Env-instSet**: **"VAX"** or **"R3000"**.
- **Env-platform**: **"Topaz"** or **"Ultrix"**.

Section 5.3.1, page 93 describes how **Preprocess** gets values for include files; the same description applies to **Compile**, because it uses the same preprocessor. Thus, to compile a C program containing the directive

```
# include <stdio.h>
```

your model could say

```
LET { buildingenv.v$Env-default(); C-env } IN
  C$Compile(hello.c)
```

By opening the **C-env** binding in the expression above, you provide a value for **stdio.h** when **hello.c** is compiled.

5.3.3 Prog

```
PROCEDURE Prog(
  pieces:           ListT;
  C-retainSymbols:  TextT;
  C-loadFormat:     TextT;
  C-entryPoint:     TextT;
  C-linkResult:     TextT;
  C-forceInclusion: ListT;
```

```
C-maxSizeForGP:    IntegerT;
Env-instSet:       TextT;
Env-platform:      TextT;
...
): TextT;
```

- **pieces**: A list of TextT values produced by Compile, or libraries in archive format produced using facilities outside this bridge.

- **C-retainSymbols**: This parameter controls retention of symbol table information in the output file. If **C-retainSymbols = "all"** *ld* retains all symbols. If **C-retainSymbols = "globals"** the bridge passes the -x switch on the *ld* command line. If **C-retainSymbols = "none"** the bridge passes the -s switch on the *ld* command line. Default value: **"all"**.

- **C-loadFormat**: This parameter controls the "magic number" of the output file. If **C-loadFormat = "demand"** the bridge passes no extra switch on the *ld* command line, producing ZMAGIC (413) format. If **C-loadFormat = "sharedText"** the bridge passes the -n switch on the *ld* command line, producing NMAGIC (410) format. If **C-loadFormat = "unshared"** the bridge passes the -N switch on the *ld* command line, producing OMAGIC (407) format. Default value: **"demand"**.

- **C-entryPoint**: If **C-entryPoint** is bound to a non-empty text value <val>, the bridge passes the -e <val> switch on the *ld* command line. Default value: "".

- **C-linkResult**: This parameter controls the relocatability of the output file. If **C-linkResult = "absolute"** the output file is an ordinary executable. If **C-linkResult = "relocatable"** the bridge passes the -r switch on the *ld* command line. (Note that this functionality is also available with **AS$Combine**.) If **C-linkResult = "relocatableWithCommon"** the bridge passes the -r and -d switches on the *ld* command line. Default value: **"absolute"**.

- **C-forceInclusion**: The value of **C-forceInclusion** must be a ListT of TextT. For each element <elem> of the list, the bridge passes the -u <elem> switch on the *ld* command line. Default value: ().

- **C-maxSizeForGP**: See description under **Compile** above.

- **Env-instSet**: **"VAX"** or **"R3000"**.
- **Env-platform**: **"Topaz"** or **"Ultrix"**.

Prog automatically includes references to standard libraries on the *ld* command line: crt0.o is loaded first, and libc.a is searched last.

When *ld* accesses a library that is not in **pieces**, the bridge gets a value for the library from the dynamic environment of the **Prog** application (recall the ... in the function signature.)

The **C-env** binding contains the standard libraries normally found in /lib, including libc.a. Thus a more complete model for a C "hello, world" program is

```
LET { buildingenv.v$Env-default(); C; C-env } IN {
  hello ~ Prog((Compile(hello.c)))
  }
```

5.3.4 Limitations of the C bridge

There isn't a **C-loadAddress** parameter to **Prog**, which would correspond to the -T option to *ld*. This is because bugs in VAX *cc* prevent -T from being passed through *cc* to *ld*. If you need to use -T (typically, for building the Nub), you must use **AS$Prog** (section 5.4.3, page 105.) Note that **AS$Prog** does not add any libraries to your **pieces** list; you must list all libraries explicitly.

Many other options available on the *cc* and *ld* command lines are not available through the C bridge at present. When compiling for **Env-instSet = "VAX"**, the following options are not yet supported:

```
cc:
  -b: Don't pass -lc automatically to ld.
  -C: Preprocessor includes comments in output
  -g: Include debugging symbol table info
  -p, -pg:
      Profiling control
  -R: Initialized shared variables
  -M: Floating point type
  -S: Produce assembly code file
```

```
ld:
  -H, -D:
      Padding for text and/or data segment.  Available
        through AS$Prog.
  -X, -S:
      Additional symbol retention options beyond C-retainSymbols
  -t, -V, -y, -M:
      Output to aid debugging.
```

When compiling for `Env-instSet = "R3000"`, the following options are not yet supported:

```
cc:
  -g0..3:
      Include debugging symbol table info.  -g2 interacts
           with -O3
  -p0..1:
      Profiling control.
  -cord, -feedback:
      Link-time rearrangement of a program for (marginally?)
        better performance, based on outputs of profiling runs.
  -O3, -j, -ko, -k:
      Optimizations that involve ucode-related manipulations.
        -O3 is likely to be awkward to implement, and of
        marginal payback.
  -Olimit:
      Limiting size of routine for optimization.  Importance
        unknown.  Probably easy to add, if necessary.
  -S: Produce assembly code file.  Somewhat awkward to
        accommodate in the present structure of models.
ld:
  -p: This seems to be related to special ucode processing.
        Importance unknown.
  -D, -B:
      Set data and bss origin addresses.  Available through AS$Prog.
  -f: Fill pattern for holes.  Available through AS$Prog.
  -jmpopt, -nojmpopt, -g:
      Controls certain kinds of jump delay slot filling. Importance
        unknown.
  -S: Suppress error messages.  Probably easy to add, if necessary.
  -VS:
```

```
        Control decimal version stamp.  Probably easy to add,
          if necessary.
    -v, -V, -y, -m, -M:
        Output to aid debugging.  Probably easy to add, if necessary.
    -b: This sketchily-documented feature seems to involve some
          symbol table manipulation when files to be linked have
          been compiled with slightly different include files.
          Vesta probably doesn't need to support this.
    -i: This sketchily-documented switch performs some special
          handling of text segment.  It could easily be added,
          if anyone can figure out what it's for.
```

The following options will probably never be supported by the C bridge:

```
cc:
  -H, -K -#, -W, -t, -h, -B:
        These are used for compiler development.  They would be
          irrelevant if the compiler were ever to be developed in
          Vesta.
  -c:   This switch is unnecessary, given the separation of
          Compile and Prog.  It is used internally by the Compile
          function's implementation.
  -I:   This switch controls the search path for #include files.
          Vesta's scope rules make this unnecessary.
  -o:   This switch controls the output file name, which is
          handled in other ways by the Vesta language.
  -E, -cpp, -nocpp:
        These switches are unnecessary, given the separation of
          Compile and Preprocess.
  -V, -v:
        These are used for compiler debugging.  It would be easy
          to add them if the compiler were ever to be developed
          in Vesta.
  -Y    This functionality can be achieved by using C-define.
  -l    This is an abbreviation mechanism for library names.
          Under Vesta, we use explicit names relative to the
          current scope.
ld:
  -EB, -EL:
        We have no plans to support big-endian machines.
  -kl, -L, -K -B:
```

```
        These switches control search paths and names emitted by
          the linker.  In Vesta, we control these with scopes.
  -bestGnum, -count, -nocount, -countall:
        These sketchily-documented switches apparently control
          some sort of automatic recompilation facility.  It
          seems to be a fairly gross hack and intended to milk
          the last ounce of performance out of certain
          applications (benchmarks?)  The approach seems
          inconsistent with the Vesta philosophy.  At least,
          considerable restructuring of the bridge would be
          necessary to incorporate these facilities, with little
          apparent benefit.
  -Y    This functionality can be achieved by conditionally
          placing suitable libraries in the Vesta scope.
  -A    This facility invokes incremental loading.  Its
          definition appears to be a bit awkward for Vesta, and
          we don't see any particular need for it.  However, it
          might be reasonable to add if there was a compelling
          need.
```

5.4 The AS bridge

The environment produced by buildingenv includes the name **AS**, which is a binding containing the four functions implemented by the AS bridge: **Assemble**, **AssembleOnly**, **Combine**, and **Prog**.

- **Assemble** assembles a source file after running the C preprocessor over it.
- **AssembleOnly** assembles a source file without using the C preprocessor.
- **Combine** links a set of object files into a combined object file.
- **Prog** links a set of object files into an executable.

All AS bridge functions have an **Env-instSet** parameter, which may be either **"VAX"** or **"R3000"**. When **Env-instSet = "R3000"** the assembly or linking is performed on a 3MAX even though Vesta is running on a Firefly.

The AS bridge provides a binding, **AS-defaults**, containing the default values of the defaultable parameters to **AssembleOnly**, **Assemble**, and **Prog**. You can assume that this binding has been opened in any standard environment that contains the AS bridge. For instance, if **AS** is defined, then **AS-define** (the first defaultable parameter of **AS$Assemble**) is also defined.

5.4.1 Assemble and AssembleOnly

```
PROCEDURE Assemble(
  source:                  TextT;
  AS-define:               BindingT;
  AS-undefine:             BindingT or ListT;
  AS-keepComments:         TextT;
  AS-keepLineNumbers:      BooleanT;
  AS-allowRecursiveMacros: BooleanT;
  AS-keepAllLabels:        BooleanT;
  AS-useLongJumps:         BooleanT;
  AS-offsetBytes:          IntegerT;
  AS-readonlyDataSegment:  BooleanT;
  AS-symTabLevel:          IntegerT;
  AS-maxSizeForGP:         IntegerT;
  AS-noWarnings:           BooleanT;
  Env-instSet:             TextT;
  Env-platform:            TextT;
  ...
  ): TextT;
```

- **source**: The source text to be assembled by as.
- **AS-define, AS-undefine, AS-keepComments, AS-keepLineNumbers, AS-allowRecursiveMacros**: These five parameters are identical in function to the similarly named parameters to **C$Preprocess**. The AS preprocessor establishes the same default symbols (based on the values of **Env-instSet** and **Env-platform**) as **C$Preprocess**. See section 5.3.1, page 93 for documentation of **C$Preprocess**.
- **AS-keepAllLabels**: Ignored if **Env-instSet = "R3000"**. If **Env-instSet = "VAX"** and **AS-keepAllLabels = TRUE**, the bridge passes the -L switch on the *as* command line. This includes labels

beginning with "L" in the symbol table produced by the assembly. Default value: **FALSE**.

- **AS-useLongJumps**: Ignored if **Env-instSet = "R3000"**. If **Env-instSet = "VAX"** and **AS-useLongJumps = TRUE**, the bridge passes the -J switch on the *as* command line. This causes the assembler to use long jumps when byte-displacement branches are insufficient. Default value: **FALSE**.

- **AS-offsetBytes**: Ignored if **Env-instSet = "R3000"**. If **Env-instSet = "VAX"**, the bridge passes the -d<val> switch on the *as* command line, where <val> is the value of **AS-offsetBytes**. This specifies the number of bytes to be assembled for offsets whose sizes aren't known a priori by the assembler. Default value: **4**.

- **AS-readonlyDataSegment**: Ignored if **Env-instSet = "R3000"**. If **Env-instSet = "VAX"** and **AS-readonlyDataSegment = TRUE**, the bridge passes the -R switch on the *as* command line. This makes initialized data segments read-only. Default value: **FALSE**.

- **AS-symTabLevel**: Ignored if **Env-instSet = "VAX"**. If **Env-instSet = "R3000"** and **AS-symTabLevel** has a value <val> in [0..3], the bridge passes the -g<val> switch on the *as* command line. This controls how much information is placed into the symbol table. Default value: **0**.

- **AS-maxSizeForGP**: Ignored if **Env-instSet = "VAX"**. If **Env-instSet = "R3000"**, the bridge passes the -G <val> switch on the *as* command line, where <val> is the value of **AS-maxSizeForGP**. This controls which variables are placed in the area accessed by the global pointer register. Default value: **8**.

- **AS-noWarnings**: If **-noWarnings = TRUE**, the bridge will suppress assembler warning messages. Default value: **FALSE**.

- **Env-instSet**: **"VAX"** or **"R3000"**.

- **Env-platform**: **"Topaz"** or **"Ultrix"**.

Assemble looks up included files in its dynamic environment, like **C$Preprocess**. See section 5.3.1, page 93 for documentation of **C$Preprocess**..

```
PROCEDURE AssembleOnly(
  source:                 TextT;
  AS-keepAllLabels:       BooleanT;
  AS-useLongJumps:        BooleanT;
  AS-offsetBytes:         IntegerT;
  AS-readonlyDataSegment: BooleanT;
  AS-symTabLevel:         IntegerT;
  AS-maxSizeForGP:        IntegerT;
  AS-noWarnings:          BooleanT;
  Env-instSet:            TextT;
  Env-platform:           TextT;
  ...
  ): TextT;
```

AssembleOnly assembles a source file without using the C preprocessor. Its parameters are all as described for **Assemble**.

The text value returned by **Assemble** and **AssembleOnly** is in a.out(5) format for the instruction set and platform defined by **Env-instSet** and **Env-platform**.

5.4.2 Combine

```
PROCEDURE Combine(
  pieces:       ListT;
  Env-instSet:  TextT;
  Env-platform: TextT;
  ...
  ): TextT;
```

- **pieces**: A list of TextT values in a.out(5) or ar(5) format.
- **Env-instSet**: **"VAX"** or **"R3000"**.
- **Env-platform**: **"Topaz"** or **"Ultrix"**.

Combine passes the elements of **pieces** to *ld -r* in order; the resulting file (in a.out format) is the result of **Combine**.

5.4.3 Prog

```
PROCEDURE Prog(
  pieces:             ListT;
  AS-retainSymbols:   TextT;
  AS-loadFormat:      TextT;
  AS-entryPoint:      TextT;
  AS-maxSizeForGP:    IntegerT;
  AS-loadAddress:     IntegerT;
  AS-dataAddress:     IntegerT;
  AS-bssAddress:      IntegerT;
  AS-fillPattern:     IntegerT;
  AS-textPad:         IntegerT;
  AS-dataPad:         IntegerT;
  AS-linkResult:      TextT;
  AS-forceInclusion:  ListT;
  Env-instSet:        TextT;
  Env-platform:       TextT;
  ...
  ): TextT;
```

- **pieces**: A list of TextT values produced by Compile, or libraries in archive format produced using facilities outside this bridge.
- **AS-retainSymbols, AS-loadFormat, AS-entryPoint, AS-maxSizeForGP**: These four parameters are identical in function to the similarly named parameters to **C$Prog**.
- **AS-loadAddress**: If **AS-loadAddress** has an integer value <val>, the bridge passes the -T <val> switch on the *ld* command line. This sets the text segment origin. Default value: "".
- **AS-dataAddress**: If **Env-instSet = "R3000"** and **AS-dataAddress** has an integer value <val>, the bridge passes the -D <val> switch on the *ld* command line. Default value: "".
- **AS-bssAddress**: If **Env-instSet = "R3000"** and **AS-bssAddress** has an integer value <val>, the bridge passes the -B <val> switch on the *ld* command line. Default value: "".
- **AS-fillPattern**: If **Env-instSet = "R3000"** and **AS-bssAddress** has an integer value <val>, the bridge passes the -f <val> switch on the *ld* command line. Default value: "".

- **AS-textPad**: If **Env-instSet = "VAX"** and **AS-textPad** has an integer value <val>, the bridge passes the -H <val> switch on the *ld* command line. This increases the size of the text segment by <val> bytes. Default value: "".
- **AS-dataPad**: If **Env-instSet = "VAX"** and **AS-dataPad** has an integer value <val>, the bridge passes the -D <val> switch on the *ld* command line. This increases the size of the data segment by <val> bytes. Default value: "".
- **AS-linkResult, AS-forceInclusion**: These two parameters are identical in function to the similarly named parameters to **C$Prog**.
- **Env-instSet**: **"VAX"** or **"R3000"**.
- **Env-platform**: **"Topaz"** or **"Ultrix"**.

Prog does not add any libraries to your **pieces** list; you must list all libraries explicitly.

5.5 The Shell bridge

The environment produced by buildingenv includes the name **Shell**, which is a binding containing the three functions implemented by the Shell bridge: **Sh**, **UnsafeSh**, and **UnsafeCsh**:

- **Sh** runs *sh*, the Bourne shell, in a restricted way that guarantees that *sh* computes a function (while allowing side-effects to /tmp.)
- **UnsafeSh** runs *sh* in a less restricted way that allows *sh* (and programs *sh* invokes) to do arbitrary things with the file system. This is useful, for instance, in order to copy results from Vesta to release directories in the file system.
- **UnsafeCsh** is like **UnsafeSh** but runs *csh*, the C shell, instead of *sh*.

The Shell bridge provides a binding, **Shell-defaults**, containing the default values of the defaultable parameters to Shell bridge functions. You can assume that this binding has been opened in any standard environment that contains the Shell bridge. For instance, if **Shell** is

defined, then **Shell-argv** (the first defaultable parameter of **Shell$Sh** is also defined.

Note: The bridge actually exports one more function, **Csh**, which is like **Sh** but runs *csh*. But this function doesn't work properly; don't use it.

5.5.1 Sh

```
PROCEDURE Sh(
  script:                 TextT;
  Shell-argv:             ListT; (* of TextT*)
  Shell-stdoutTreatment: TextT;
  Shell-stderrTreatment: TextT;
  Shell-exitOnError:      BooleanT;
  Shell-verbose:          BooleanT;
  Shell-echo:             BooleanT;
  Shell-unsetVarIsError: BooleanT;
  ...
  ): BindingT;
```

- **script**: The commands to be executed by *sh*.
- **Shell-argv**: A list of text values that are made available to the *sh* script as its numbered parameters (e.g., $1, $2.)
- **Shell-stdoutTreatment, Shell-stderrTreatment**: These parameters control how the bridge responds to output on stdout and stderr.

value	meaning
"ignore"	Output is discarded.
"feedback"	Output is displayed in the user interface.
"feedbackError"	Output is displayed in the user interface and is treated as an error; that is, the evaluation of the function fails.
"value"	Output is returned as part of the result of the function, bound to the name **stdout** or **stderr** as appropriate.

The default value for **Shell-stdoutTreatment** is **"value"**, so, by default, the result binding contains a component named **stdout**. The default value for **Shell-stderrTreatment** is **"feedbackError"**,

so, by default, any output to stderr results in a Vesta evaluation error.

- **Shell-exitOnError, Shell-verbose, Shell-echo, Shell-unsetVarIsError**: These parameters translate into *sh* command-line switches as follows:

name	default	mapping to *sh* switch
Shell-exitOnError	**FALSE**	-e
Shell-verbose	**FALSE**	-v
Shell-echo	**FALSE**	-x
Shell-unsetVarIsError	**FALSE**	-u

The environment variables of the *sh* process are initialized from the environment variables of the Vesta server. This is a bug, because it represents a way of passing function parameters that is not controlled by the Vesta language.

The bridge executes **script** in a virtual working directory that contains every text- and binding-valued name in the Vesta environment in which **Shell$Sh** was applied. (This access to the application environment is allowed because of the ... in the signature of **Sh**.) **script** sees a text-valued name in this environment as a Unix file. **script** sees a binding-valued name in this environment as a subdirectory of its working directory; this subdirectory in turn contains subdirectories if the binding contains bindings. If **script** tries to read a file in its working directory that corresponds to another type of Vesta value (e.g. a list) it receives a "file not found" error.

script is allowed to write files in its working directory and in /tmp (and in subdirectories of these directories), but nowhere else. By writing in its working directory, **script** does not alter the Vesta environment in which **Shell$Sh** was applied. **script** is allowed to read any file it has written in a given application of **Shell$Sh**, obtaining the results of the earlier write. In performing writes to /tmp, normal file system protections apply; **script** is executing in the role of the user who started the Vesta server.

If **script** attempts to read a file whose absolute path name does not begin with the working directory or /tmp, the bridge performs the read by extracting the final component of the path and looking it up in **script**'s Vesta environment. This is a kludge, to deal with existing scripts containing absolute path names like /bin/echo. The read fails if the name is not bound, or is bound to a non-text value.

If **script** exits with non-zero status, **Shell$Sh** causes a Vesta evaluation error. Otherwise, **Shell$Sh** returns a binding containing a name for each file written by the **script** into its working directory, plus the name **stdout** if **Shell-stdoutTreatment = TRUE** and the name **stderr** if **Shell-stderrTreatment = TRUE**. For example, using default values of **Shell-stdoutTreatment** and **Shell-stderrTreatment**,

```
Shell$Sh("echo abc > x")
```

returns a binding equivalent to

```
{x ~ "abc", stdout ~ ""}
```

Note that **script** may construct subdirectories of its working directory, which **Shell$Sh** returns as bindings. For example,

```
Shell$Sh("mkdir a; mkdir b; echo abc > a/x; echo hello")
```

returns a binding equivalent to

```
{a ~ {x ~ "abc"}, b ~ {}, stdout ~ "hello"}
```

The preceding examples are not complete, because they do not show how the names **echo** and **mkdir** are bound in the environment of the **Shell$Sh** application. The building environment defines **Shell-Utils**, a binding containing a set of standard utility programs that can be invoked from shell scripts. So to create a file /tmp/hello containing the string "Hello, world" your model would say

```
LET {
  buildingenv.v$Env-default();
  Shell-Utils;
  contents ~ "Hello, world" }
IN
  Shell$Sh("copy contents /tmp/hello")
```

By opening the **Shell-Utils** binding in the expression above, you provide a value for the **copy** program in the script executed by **Shell$Sh**.

It is prudent to open the **Shell-Utils** binding in the smallest possible scope, because it introduces a host of popular names into the environment (including **test** and **cat**.) Here are two alternative ways of writing the example that avoids opening **Shell-Utils**:

```
LET {
  buildingenv.v$Env-default();
  copy ~ Shell-Utils$copy;
  contents ~ "Hello, world" }
IN
  Shell$Sh("copy contents /tmp/hello")

LET {
  buildingenv.v$Env-default();
  contents ~ "Hello, world" }
IN
  Shell$Sh("Shell-Utils/copy contents /tmp/hello")
```

5.5.2 UnsafeSh and UnsafeCsh

```
PROCEDURE UnsafeSh(
  script:                 TextT;
  Shell-argv:             ListT; (* of TextT*)
  Shell-stdoutTreatment: TextT;
  Shell-stderrTreatment: TextT;
  Shell-exitOnError:      BooleanT;
  Shell-verbose:          BooleanT;
  Shell-echo:             BooleanT;
  Shell-unsetVarIsError: BooleanT;
  ...
  ): BindingT;
```

UnsafeSh differs from **Sh** (section 5.5.1, page 107) in permitting **script** to perform reads and writes outside of its working directory and /tmp. In performing these reads and writes, normal file system protections apply; **script** is executing in the role of the user who started the Vesta server.

Thus, **Shell$UnsafeSh** allows the user to "step outside" the functional programming world of Vesta. You should use **Shell$Sh** in preference to **Shell$UnsafeSh** when you are actually doing functional programming.

```
PROCEDURE UnsafeCsh(
  script:                 TextT;
  Shell-argv:             ListT; (* of TextT*)
  Shell-stdoutTreatment: TextT;
  Shell-stderrTreatment: TextT;
  Shell-exitOnError:      BooleanT;
  Shell-verbose:          BooleanT;
  Shell-echo:             BooleanT;
  ...
  ): BindingT;
```

UnsafeCsh is like **UnsafeSh**, but uses *csh* instead of *sh* to process **script**.

5.6 Utility functions

All of the utility functions that create files will also create intermediate directories as necessary. All of the utility functions return a binding with uninteresting contents.

```
PROCEDURE CopyToFile(
  textvalue: TextT;
  name:      TextT;
  ...
  ): BindingT;
```

CopyToFile copies **textvalue** to the file with pathname **name**.

```
PROCEDURE CopyM2ProgToFile(
  program: Value;
  name:    TextT;
  ...
  ): BindingT;
```

CopyM2ProgToFile copies the image text of **program** to the file with pathname **name**.

```
PROCEDURE ShipM2(
```

```
  dir:  TextT;
  name: TextT;
  file: Value;
  ...
  ): BindingT;
```

ShipM2 ships an executable program to a release or test directory. It copies the image text of the program **file** to the file with name **name** in the appropriate subdirectory of **dir**.

```
PROCEDURE ShipForkedM2(
  dir:  TextT;
  name: TextT;
  file: Value;
  ...
  ): BindingT;
```

ShipForkedM2 is like **ShipForkedM2**, but for programs that need to be forked (e.g. most window-based applications.) It copies a generated shell script to the file with name **name** in the appropriate subdirectory of **dir**, and copies the image text of the program **file** into a different subdirectory (to be invoked by the shell script.)

```
PROCEDURE ShipManpage(
  rootDir: TextT;
  name:    TextT;
  fname:   TextT;
  ...
  ): BindingT;
```

ShipManpage copies the file named **fname** to the subdirectory of **rootDir** that holds manpages. **name** is the name of the manpage, e.g. **"cat.1"**.

```
PROCEDURE ShipPrintDoc(
  dir:      TextT;
  fileName: TextT;
  file:     TextT;
  ...
  ): BindingT;
```

ShipPrintDoc copies the file named **file** to the subdirectory of **rootDir** that holds printdoc and readdoc documents. **fileName** is the name of the printdoc or readdoc document, e.g. **"vesta.ps"** or **"vesta.doc"**.

The four **Ship** functions insulate your model from details of where the files are shipped. It is more difficult to insulate *you* from such details. For instance, when you test-ship files to /tmp, you need to know where the files landed, so you can find them. Here's a key to the files created by the four **Ship** functions:

Executing the ship function:	Creates the file(s) named:
`ShipM2("/tmp", "a", file)`	/tmp/bin/a
`ShipForkedM2("/tmp", "a", file)`	/tmp/bin/a and /tmp/lib/a.run
`ShipManpage("/tmp", "a.N", file)`	/tmp/doc/man/catN/a.N
`ShipPrintDoc("/tmp", "a", file)`	/tmp/doc/printdoc/a

```
PROCEDURE SendMail(
  user:    TextT;
  text:    TextT;
  subject: TextT;
  ...
  ): BindingT;
```

SendMail sends mail to **user** with subject **subject** and body **text**. Ignore the result binding.

6. Repository

6.1 Abbreviating path names in the clerk

The clerk is the part of Vesta that gives access to the Vesta repository via the Unix file system interface.

The clerk provides some abbreviation capabilities to help you deal with versions.

You can default version numbers of package names. For example, if you type

```
ivy /vesta/proj/buildingenv/MODEL
```

to a shell, you will get the model that is bound to the latest buildingenv (buildingenv.73 the last time I checked). If you type

```
ivy /vesta/proj/buildingenv.19.vulcan/MODEL
```

you will get the model that is bound to the latest version along the vulcan branch off buildingenv.19.

If you type

```
ivy /vesta/proj/buildingenv.vulcan/MODEL
```

you will get the model of the latest bound /vesta/proj/buildingenv.n.vulcan.m where "latest" is defined by the lexicographic ordering on the print names of packages, except mumble.m is greater than mumble.n in our ordering if m is a larger version number than n. For example, in our ordering (let's call it the FLO, for "funny lexicographic ordering")

```
buildingenv.9 is less than buildingenv.10
buildingenv.1 is less than buildingenv.1.vulcan.0
```

In general you could default a version number by simply leaving it (and its preceding dot) out. And a defaulted package name (DFN) will expand to the latest (in FLO) bound package whose name matches the DFN. An undefaulted package name is said to match a DFN if their print names, with versions stripped out, are Text.Equal

In addition to defaulting by leaving version numbers out, you can also default by "prefixing". For example, if you type

```
ivy $vp/buildingenv@/MODEL
```

you will get the latest (in FLO) bound package that has "buildingenv" as a prefix (today, $vp/buildingenv.35.vulcan.0/MODEL.) This is especially useful in private repositories, where names tend to be long and the latest version is almost always what you want:

```
ivy $ep/m2ast@/M2AST.def
```

might get you $ep/m2ast.4.disconnected.25/M2AST.def.

You can combine version defaulting and prefixing if you like. For example,

```
ivy /vesta/proj/vestarep.hanna@
```

will give you the latest bound package that has a prefix that matches "vestarep.hanna".

The prefix operator @ can only be used immediately after a family name or branch or version. For example,

```
/vesta/proj/vestarep.14@
```

will expand to /vesta/proj/vestarep.14.hanna.0 today, but

```
/vesta/proj/vestarep.1@
```

will expand to /vesta/proj/vestarep.1 (not /vesta/proj/vestarep.14.hanna.0), and

```
/vesta/proj/vestarep.@
```

will return an error. So will

```
/vesta/proj/vestar@
```

because there is no "vestar" family in /vesta/proj.

6.2 Managing your private repository

You are responsible for controlling the amount of space consumed by your private repository. You'll discover that space is chewed up rapidly, since everything is versioned. Object files and program images pile up quickly when you are developing a program.

The *vrweed* utility allows you to delete derived objects from your private repository. You specify a set of models to act as the roots of a garbage collection, and vrweed figures out the derived objects that are unreachable from the roots. vrweed provides a variety of ways of specifying the roots of the garbage collection. We would like a storage management system that requires less manual intervention; that's a topic for future work.

You can also delete source objects from your private repository with vrweed. Weeding sources objects violates Vesta's model of immortal sources, but when you've checked a package into the public repository you are unlikely to need all of the intermediate versions from your private repository. Weeding sources is seldom necessary, because sources are small, but you might need to weed sources every six months or so if you generate a lot of versions. Naturally, you should be especially careful when weeding sources from your private repository.

The *vrcompress* utility reduces the amount of space consumed by source objects by compressing their representation. Once compressed, the objects remain available and can be accessed by the UID without any special action on your part. Reading them takes longer, however, since the bytes must be reconstructed from the compressed representation instead of simply being read from a file.

As of this writing, vrcompress has not been applied to the public repository. If you are a conservative user, you should wait for this event before applying vrcompress to your own repository.

See the vrweed(1) and vrcompress(1) manpages for complete information.

6.3 Miscellaneous issues

6.3.1 The comment associated with a package

The respository stores this information with each checked-in package version: when created, creator, when checked in, checked in from where, root UID, comment.

The "comment," generally supplied at checkin time, should be one or two key phrases that identify the package version and its purpose. A future release of VestaRep will restrict the size of the field to some small value, perhaps two hundred bytes. Restricting comments in this way will allow applications such as ManageReps to list a set of package versions and provide the comments as brief extensions of the package names.

Additional commentaries, as desired by the package owner, should be stored as source files at the package's root-level. For instance, versions of the taos package contain a file release.msg that describe the new content of a taos version in detail.

6.3.2 Lack of access control

This release of Vesta doesn't emphasize privacy. Anyone can create things in a public repository (just like /proj/packages), and only you can create things in your private repositories (just like conventional working directories). Vesta keeps access control lists that ensure these properties. There are also some other access control lists that protect our precious public repositories against blunders.

However, Vesta has no access controls that limit reading. Anyone can look at anything in any repository. Obviously, we couldn't sell Vesta to real customers with this lax attitude, but it matches what SRC has done for years with /proj/packages and (with rare exceptions) working directories.

Just don't keep personal (or personnel) material in Vesta repositories.

6.3.3 Replicas

Vesta has the groundwork in place for replicating public repositories. Eventually, /vesta/proj will have a replica in Paris as well as in Palo Alto,

and a siphon will keep things consistent at the two sites. But all of this is in future releases.

For now, you need only be aware that public repositories will eventually be replicated and that you will occasionally come across replica names (e.g., when creating repositories with ManageReps).

7. Conventions for sharing software at SRC

Let $vp be shorthand for /vesta/proj in what follows.

7.1 Example models

By far the most effective way to write a Vesta model is to evolve some existing model. In many cases, a model exists that is quite similar to the one you want; your problem is to find the similar model, then modify it to suit your needs.

Here is a table listing different categories of packages, with an example of each:

- Application with command-line user interface: settrestleparams
- Application with FormsVBT user interface (uses Bundle for resources): calculator
- Application that (1) requires libraries from Env-pkgs(), (2) exports most of its functionality as a library, and (3) implements one public interface using several modules, with private interfaces hidden by the model: tem
- Documentation written in Luke: companion
- Simplest library (no private interfaces): random
- Library with Tinylisp interface: runfilter
- RPC server: mac_thes
- Substitutes the current version of the library exported by the package for the released version, in order to build a test program (uses Env-buildWithOverrides): runfilter/tests

- Replaces a portion of a basic shared library on every build, not just on experimental or test builds (uses Env-buildWithOverrides): efb
- Sends mail to the owner on each checkin: spell
- Ships compiled Tinylisp code (so far, only for the Vax): m2pp
- Supplies "shell utility" used in buildingenv: trans
- Runs regression test as part of preCheckin: trans
- Uses Env-forBridges to build a program transforming tool: bootfile or llgen/generator

The Vesta language is quite general and there's no way to tell, just from reading a example model, what alterations to the model are OK and what alterations would violate SRC's conventions for models. The goal of the remainder of this chapter is to explain and motivate SRC's conventions for models, enabling you to make effective use of Vesta at SRC.

7.2 The release model

The release model ($vp/release/MODEL) describes all of SRC's shared software. It builds and ships the executables, man pages, documents, and other derived objects that SRC users need to access through the file system.

The release model imports a version of buildingenv and all the individual models that build executables and documents. The release model calls those models to build their derived objects using the release model's buildingenv (rather than the building environments imported by the individual models) and then calls those models to ship their deriveds into the file system.

Here is an overview of the structure of /vesta, including the directories created by the release process:

```
/vesta
    /proj -- the public repository
    /out -> releases/release.15 -- link to cur release
    /releases
```

```
/release.15
    /rls -> $INST_SET/rls
    /exp -> $INST_SET/exp
    /VAX
        /rls -- release directory
            /bin -- executables
            /etc -- system admin executables
            /lib -- data files, etc.
            /doc
                /man
                    /cat[1-n] -- manpage dirs
                    whatis    -- apropos index
                /printdoc     -- printdoc/readdoc
        /exp -- experimental release directory
            ...same structure as /rls...
    /R3000
        ...same structure as /VAX...
    /Alpha
        ...same structure as /VAX...
/release.14
    ...same structure as /release.15...
...
```

The release.NN directories are numbered in correspondence to the version of the release model that created them. For instance, $vp/release.14 created /vesta/releases/release.14.

The path /vesta/out/rls takes you, via two levels of symbolic links, to the release directory for the current release and the instruction set you are executing. For example, if the current release is release.15 and you are running on a VAX, /vesta/out/rls expands to /vesta/releases/release.15/VAX/rls.

The release process populates the /rls directory via the ship actions called from the release model, and creates a nearly empty /exp directory. The initial /exp contains files shipped by applications that could not be built with the release's buildingenv for one reason or another. Between releases, application developers can ship new versions of their applications to /exp, using the preCheckin or postCheckin actions of their models. (Application developers can also disable these actions in order to wait until the next release.) The standard SRC search path includes /vesta/out/exp followed by /vesta/out/rls, so bug fix executables are loaded in preference to the

released executables.

7.3 The buildingenv model

The buildingenv model defines a consistent environment in which you can compile and link modules. This environment defines both the libraries that your modules can call and the programming tools that you use to transform your modules into executable code.

At present, the instruction set and operating system are encoded in the name of each building environment model. Thus, buildingenv-vax is a (Taos) building environment for the VAX, buildingenv-r3000 is a (Taos) building environment for the 3Max, and buildingenv-osf1-r3000 is an OSF/1 building environment for the 3Max. There are plans to combine these building environments under a single umbrella.

A few programs for the VAX are still built using the old **mm** compiler. mm-based building environments are branches of buildingenv-vax, e.g. buildingenv-vax.68.mm.0.

A buildingenv model returns a binding with several components, all of which are functions. You'll typically call the Env-default function. A detailed description of Env-default will be given in a later revision of this document; for now, a few basic facts about Env-default will suffice.

The binding returned by Env-default contains compiled interfaces exported by the most commonly used library packages. The names are of the form Mumble.d where Mumble is the interface name. The binding also contains implementations of these interfaces. The implementations are grouped into Vulcan shared libraries, with names like SL-basics and SL-ui. These shared library names appear in the binding, not the names of individual implementations. (The same grouping of implementations is used in the mm world, but the libraries are not shared.)

In writing a model that builds a program, the first question you may need to answer is, "Does Env-default contain all of the interfaces my program needs?" If you don't know the answer, you should assume "yes" and then ask a more experienced user to help you patch things up if a compiler complains about not finding a library interface that you need.

The second question you may need to answer is, "What shared library should I link into my program?" With the current library structure, the

correct answer is almost always SL-ui for window applications, and SL-basics for command-line applications.

It is possible to answer these questions for Vulcan applications using the vxref(1) tool. But at present the process is too laborious to do for every interface your program imports.

The binding returned by Env-default also contains the Env-pkgs function. Env-pkgs returns a binding containing the names of all packages known to buildingenv, each name paired with the current version of its model. You use Env-pkgs to access components of library packages that are not directly included in Env-default. Some examples below demonstrate the use of using Env-pkgs.

7.4 Model conventions required by release model

In order for the release model to evaluate successfully, models for individual applications and libraries must follow certain conventions.

7.4.1 Application models: build, doc, and ship functions

If your model builds an application to be included in the release model, your model must include **build**, **doc**, and **ship** functions that conform to the following specifications:

build

The **build** function assumes that the caller's environment has opened buildingenv.v$Env-default() and buildingenv.v$Env-pkgs() and has established suitable values for linker options such as M2$M2-standAlone. **build** is declared as **LAMBDA ... IN** if it requires access to the caller's environment, **LAMBDA IN** otherwise (this would be very unusual.)

build returns a binding containing the programs exported by the model. The name part of each binding element is the program's name, and the

value part is the program (typically the result of M2$Prog.) Names must be chosen to avoid naming conflicts with other applications.

All the compiled .defs and .mods from Env-default are available within **build**. To access compiled .defs and .mods that are not included in Env-default, you navigate via the package name, available within **build** because Env-pkgs is open. (Do not call Env-pkgs again from within the **build** function.) For instance, the **build** function of an application that imports the DupWr interface might look like

```
build ~ LAMBDA ... IN
    LET {
        M2;
        dupwr$intfs();
        }
    IN {
        hello ~ Prog ((
            Compile(Hello.mod),
            dupwr$impls(),
            SL-ui));
        };
```

Sometimes there is more than one level of indirection. For instance, the taos package, included in Env-pkgs, is itself an umbrella model containing a pkgs function. To compile an application that imports Taos's Terminal interface, you call

```
taos$pkgs()$consoleterminal$intfs();
```

(Example: $vp/tem/MODEL)

doc

The **doc** function assumes that the caller's environment has opened buildingenv.v$Env-default(). Declared as **LAMBDA ... IN** if it requires access to the caller's environment, **LAMBDA IN** otherwise.

doc returns a binding containing the documentation exported by the model. The name part of each binding element is the document's name, and the value part is the document (plain-text or PostScript.) Names

must be chosen to avoid naming conflicts with other documents and with applications. The usual convention for files to be shipped as manpages is <program-name>.<man-section>. (Example: $vp/settrestleparams/set_trestle_params.1.)

ship

The **ship** function assumes that the caller's environment has opened **buildingenv.v$Env-default()**, **build()**, and **doc()**. Declared as **LAMBDA dir, ... IN**.

ship copies applications and documents to conventionally-named subdirectories of dir. **ship** is executed for its side-effects; the result is ignored.

The utility functions ShipM2, ShipForkedM2, ShipManpage, and ShipPrintDoc are often useful in writing a **ship** function; see section 5.6, page 111.

7.4.2 All models: IMPORT conventions

Often, your model performs an import to get access to components that are needed in order to build test programs (in the case of library packages) or test/experimental versions of applications. (Recall that in an application model, the version of an application that you build in the pre/postCheckin function is an "experimental" version; the "release" version is built by the release model.) Call such an import a *test import*.

Sometimes, your model performs an import to access components that are logically an integral part of your package, but are split out into separate packages to facilitate concurrent development. For instance, the vesta model ($vp/vesta/MODEL) imports vestabase, vestaeval, and vestarep. Values from these three packages all contribute to the bindings returned by vesta$intfs() and vesta$impls(). The vesta model is an "umbrella" over these three packages. Call such an import an *include*.

The buildingenv model is an include of the release model. If a model exports both a library and one or more applications, it is included in buildingenv, not release; in this case, the release model accesses the components it needs via Env-pkgs.

Your model identifies its includes by placing them after a comment of the form

```
(***INCLUDE***)
```

that occurs either at the start of the IMPORT list or between two entries of the list (after the comma of one and before the name of the next.). Here's an example from the vesta model:

```
IMPORT
  buildingenv.v (*/vesta/proj/buildingenv.39*) ~ <uid>,
  (***INCLUDE***)
  vestabase.v (*/vesta/proj/vestabase.6*) ~ <uid>,
  vestaeval.v (*/vesta/proj/vestaeval.26.hanna.0*) ~ <uid>,
  vestarep.v (*/vesta/proj/vestarep.19.chiu.0*) ~ <uid>,
IN
```

If all includes are correctly identified, the include relation defines a tree rooted at the release model. Every package that contributes to the release is reachable from the release model via includes, and only one package version from a package family is reachable.

The release model requires that your package model be disciplined in referencing test imports: They should be referenced only in a test or pre/postCheckin component. Never reference a test import in a **build**, **doc**, intfs, or impls function. Your model may reference an include anywhere. By definition of "include", an application model will reference an include at least once from a **build** or **doc** component, and a library model will reference an include at least once from an intfs or impls component.

The typical model has no includes and a single test import: buildingenv. A typical use of buildingenv in an application model is:

```
IMPORT
  buildingenv.v (*/vesta/proj/buildingenv.70*) ~ <uid>,
IN {

  ...
```

```
LET {
  testf ~ LAMBDA dir IN
    LET {
      buildingenv.v$Env-default();
      Env-pkgs();
      }
    IN {
      build(), doc();
      ship(dir);
      };
  }
IN {
  postCheckin ~ testf(CAT(
    ("/vesta/out/",
     buildingenv.v$Env-default()$Env-instSet,
     "/exp")));
  test ~ testf("/tmp");
  }
}
```

It is important to call **build** within a **LAMBDA** expression, like **testf**, because this best simulates the environment that will be seen by outside callers of **build**, like the release model. If test calls **build** directly, without a layer of **LAMBDA**, the the dynamic environment of **build** will include the entire **DIRECTORY** of the application model.

The testf function's call on Env-pkgs is unnecessary if Env-default supplies everything needed by the application's **build** function.

Another typical application model has no includes and two test imports: buildingenv and some library package. Here's a scenario that results in this structure. You own an application, and somebody has shipped a new version of the table package with a bug fix that's important to your application. You would like to ship a repaired application, but the table package fix has not yet been included in the weekly buildingenv. So you check out your application, add a test import of the table package, and change the test/postCheckin component to perform a buildWithOverrides incorporating the imported table package:

```
IMPORT
  buildingenv.v (*/vesta/proj/buildingenv.40.vulcan.0*) ~ <uid>,
  table.v (*/vesta/proj/table.5*) ~ <uid>,
```

```
IN {

  ...

  LET {
    testf ~ LAMBDA dir IN
      LET {
        buildingenv.v$Env-buildWithOverrides(
          {table ~  table.v}, {});
        Env-pkgs();
        }
      IN {
        build(), doc();
        ship(dir);
        };
    }
  IN {
    postCheckin ~ testf(CAT(
      ("/vesta/out/",
       buildingenv.v$Env-default()$Env-instSet,
       "/exp")));
    test ~ testf("/tmp");
    }
  }
```

Note that you do *not* modify the **build** function to mention the new table package. That way, a later call to **build** from the release model will use the table package that's part of the release, not the table package that your model imports. Yet you can ship a fixed version of the application right away, without waiting for a release.

7.5 Model conventions required by buildingenv

If your model builds a library to be included in buildingenv, your model must include intfs and impls functions that conform to the following specifications:

7.5.1 intfs

The **intfs** function assumes that the caller's environment includes all of the bridges and compiled interfaces required by the public interfaces of this package. (The buildingenv model is responsible for calling the various **intfs** functions in the correct order.) Declared as **LAMBDA ... IN**.

intfs returns a binding containing the compiled interfaces exported by the model. Interface names must be chosen to avoid naming conflicts with other applications.

7.5.2 impls

The **impls** function assumes that the caller's environment includes all of the bridges and compiled interfaces required by the implementation modules of this package. In particular, the caller's environment includes the result of a call to **intfs**. Declared as **LAMBDA ... IN**.

impls returns a binding containing the compiled implementations exported by the model.

In case of a server package, the correct implementation to include in the **impls** function is the client stub (and any related glue modules.)

7.6 Naming conventions

Each model binds the identifier "owner" to the login name of the person who is responsible for maintaining the model.

All model imports have names derived by taking the package family name of the import and appending ".v". In the unusual circumstance of importing two models from the same package family, choose a name containing no upper case letters and ending with ".v". Immediately following the name of the import, and before the tilde, include a comment giving the Vesta pathname of the imported model. For instance,

```
IMPORT
  table.v (*/vesta/proj/table.5*) ~ /v/0haM08002P.Zay/Macd.7d1.table,
```

This comment is important both to make the import comprehensible to others and to aid tools that revise the imports of models.

To simplify the overall structure of SRC's models, your model only imports root models from other packages – never submodels. This implies that the root model of a package is designed to provide access to all components of the package that are meant to be used from other packages.

7.7 Updating buildingenv and release models

For now, maintaining these umbrella models is a job for experts. Known experts are Sheng-Yang Chiu, John Ellis, Roy Levin, and Paul McJones. If you feel a need to change either of these umbrella models, please consult an expert first. They know where the bodies are buried.

If you make a change that affects either of these umbrella models, please post a message to src.release with either (buildingenv component) or (release component) in the title, as appropriate. This is not necessary if you are just shipping a new main-line version of a package that is included in one of the umbreallas, unless your changes have structural implications for an umbrella model.

We would like to write down the rules you must follow to change these models, but we don't yet know what the rules should be.

7.8 Philosophy of package model structure

Four major considerations motivated the structure of package models described in this chapter:

1. Models for packages manipulated by "ordinary programmers", i.e., non-wizards, should be simple. (Of course, everyone at SRC is potentially a wizard, but people often prefer to pretend they aren't, or can't be.) In particular, the models that describe application packages and library packages should contain mininal boilerplate and no delicate wizardry.

2. The environment seen by people constructing and testing new package versions should be easy to manipulate. At the same time, those manipulations should not be visible to other people unless they take explicit actions. So, for example, if Fred has to rebuild a new version of Taos to try out some library package, Carol shouldn't even be aware of it unless she explicitly asks for that new version. This principle applies even if Fred has to modify a widely-used public interface (e.g., OS.def) to accomplish his goal.

3. Reconstruction of major collections of packages ("umbrellas") or large groups of applications should be smoothly automated. Thus, it should be a small intellectual task for anyone to "rebuild /proj/packages" following a source-compatible change to a low-level interface.

4. Construction of our software base for machines that execute different instruction sets should be straightforward. Of course, the challenging part of this task is making the vast majority of our software portable in the first place. Fortunately, that's the Portable Taos project's problem :-> But, given that this has been done, our models should be parameterized in a way that makes it easy to say "Do all of this for a VAX, a 3MAX, and an ALPHA". For the bulk of our software, the models should make it trivial to build any package for any machine on any machine, provided, of course, that the necessary cross-compilers, etc., exist.

If you've gotten far enough with Vesta to be reading this, you can probably begin to form your own opinion about how successfully the actual structure addresses these concerns.

7.9 Summary of Conventions for SRC Models

To make it feasible to build the **buildingenv** and **release** models, some conventions must be followed in the package models they import. Here's a summary of the conventions.

7.9.1 All Models

DIRECTORY clause:

- Any (sub)models named here lack an extension.

IMPORT clause:

- Only root models can be imported.
- Models have the extension ".v", and have no upper-case letters in their names.

Components of model body:

- Component names avoid upper case letters, with specific exceptions for the **buildingenv** model.

7.9.2 Application Models

IMPORT clause:

- References buildingenv.v exclusively, except in rare cases.

Required components of model body:

- build = function to build executable(s)
- doc = function to build documentation
- ship = function to copy exported results to Unix directories
- owner = text value identifying package version owner

Recommended components of model body:

- test = invocation of **build** and **doc** in suitable testing environment

Library Models

`IMPORT` clause:

- References buildingenv.v exclusively, except in rare cases.

Required components of model body:

- intfs = function to build client interfaces
- impls = function to build implementations of client interfaces
- owner = text value identifying package version owner

Additional required components of client/server library model body:

- serverIntfs = function to build server-only interfaces
- serverImpls = function to build implementations of server interfaces

Recommended components of model body:

- doc = function to build documentation

Optional components of model body:

- build = function to build closely-related executable(s) and documentation
- ship = function to copy exported results to Unix directories
- test = invocation of **`build`** and **`doc`** in suitable testing environment, and/or some amount of construction of the interfaces and implementations (possibly just compilation, or possibly a full-scale test program)
- additional functions for specialized clients

Server Models

`IMPORT` clause:

- buildingenv.v
- Library packages that are private to this server (not in buildingenv)

Required components of model body:

- intfs = function to build client interfaces
- impls = function to build implementations of client interfaces
- owner = text value identifying package version owner
- build = function to build closely-related executable(s)
- doc = function to build documentation

Recommended components of model body:

- test = invocation of **`build`** and **`doc`** in suitable testing environment

Optional components of model body:

- ship = function to copy exported results to Unix directories
- additional functions for specialized clients

7.9.3 Conventions and guidelines for package names

Any user of a public repository can create a new package family or a package version within an existing family. Once created, these names stay around indefinitely, so it's a good idea to choose them carefully.

1. Package names should be all lower-case. This convention is familiar to users of /proj/packages, although some people resist it. This time, let's not.

2. Pick a name that is suggestive of the package functionality. Two- and three-character Unix names are discouraged. But also remember that people type these names fairly frequently, so 20-character names, while informative, aren't going to endear you to your peers.

3. The more specific the package functionality, the more specific the name should be. Try not to preempt a name that you might reasonably expect would be used for a more widely-used package. For example, if you are implementing a language-runtime package that manages an Algol-style variable display, don't call it "display"; it's reasonable to expect that "display" should be used for something having to do with screens on desks. Another example: if you plan to call a package "math", you're probably making a mistake. Sweeping names, like "communications", or "ui", should be reserved for umbrellas, not consumed by particular packages.

4. When you create a branch in a family tree, try to pick a name that suggests the reason for the branch. Examples: text.3.bugfix.0, random.5.moreuniform.1. Sometimes there isn't any obvious name, in which case you can fall back on your user name. But even the name sort.4.faster.0 is better than sort.4.levin.0.

5. Avoid machine- or platform-specific syllables in package names. We seek to make our packages machine-independent, and, where necessary, we parameterize their machine-dependencies internally. Thus, we don't have a package called **vaxinstr**, as we did in /proj/packages. Instead we have a package called **instr** that contains machine-specific definitions for the VAX and for other machines internally.

8. UseVesta

8.1 Introduction

UseVesta is a window-based and FormsVBT-based tool that enables the user to command and control the ordinary operations associated with the use of Vesta: checkout, evaluation, checkin, etc.

It relies on the lower-level VestaUserLevel interface to actually perform much of the work. (The VestaUserLevel interface was designed to enable multiple user interface realizations to share the same lower-level implementations.) An understanding of the VestaUserLevel interface is not necessary for the reader of this document.

Within this chapter, the first sections give an overall description of UseVesta. The middle sections describe, in detail, individual commands. The final sections cover error messages, recovery from machine failures, and the like.

All commands mentioned are indexed under the main index item "UseVesta Command". Commands visible at the top level are indexed under their name. Commands within pop-up menus are indexed under their name, with the name of the pop-up following, e.g. "Abandon (in Cmds~). Commands that appear within dialogs (forms that are created by a command, a la Ivy's "Find.." form) are indexed under the command that produces the dialog and their name, e.g. as "Checkin.. -> Checkin".

8.2 The Session Directory

UseVesta (like VestaUserLevel) is designed around a particular usage paradigm, the session directory.

Vesta (with UseVesta) minimizes its dislocation of the user's development environment by creating, for the user, a "session directory" that mimics the structure of sources that he would have dealt with, via the package tools, in his previous "make"-centered development environment.

This greatly simplifies the construction of tools for the new environment, since most of the old tools work without change. File-based source editing, via ivy, emacs, or whatever, is not affected. Source manipulation via sed, cp, or whatever, is not affected. Source examination (in the session directory), via grep, find, or whatever, is not affected.

[This paradigm for source manipulation and control was chosen after discussion and debate within the Vesta project. Several possible arrangements, including the project's original paradigm (editors directly augment the repository), are described in messages to src.vesta of 3 August and 5 August 1989.]

8.2.1 The Paradigm

The basic use of the session directory is parallel to the getpackage-shippackage cycle:

- "Checkout" a package instance, say "text.123": This reserves the new package-name (text.123) in a public repository, creates an instance of the new package in the user's private repository, and copies a "starter" suite of source files from an existing package (text.122, the "parent" of the package being created) to the user's session directory.
- Go through edit-compile-test cycles: Each editing phase (one or several source files modified) is ended with an "Advance" operation, which adds (logically appends) a package-instance to the family in the user's private repository.
- "Checkin" the package: Takes a package instance in the user's private repository and copies it back into the public repository under the reserved name.

The basic arrangement is augmented with a number of ancillary facilities that, e.g., enable the state of repositories and the session directory to be examined, create package families, and enable packages to be "forked."

8.2.2 The Directory Content

For every checked out package, there is an associated "session directory" of the user's. The session directory is supplied (perhaps implicitly, from the user's viewpoint) at checkout.

The content of the session directory is a tree of directories and sources that exactly reflects the tree explicit in the root model of the checked out package. The root model of the package is in the session directory as the file MODEL. For each name in the (non-import part of the) directory of the root model:

- If the item is a non-model source, then the session directory contains a file which contains the source. The name of the file in the session directory is the same as the identifier in the DIRECTORY of the model.
- If the item is a (sub)model, then the session directory contains a (sub)directory with that name. The sub-model then appears in that subdirectory just as the model appears in the session directory. (The file MODEL in that subdirectory contains the source of the submodel, etc.)

Whenever UseVesta scans the session directory to find the user's current set of source files and submodels, it ignores any files whose names begin with a "." or begin with a "," or end with a "~" or are exactly "core".

The session directory also contains files that are used by VestaUserLevel to control the checkout/checkin process. All of these VestaUserLevel-specific files have names that begin with ".V." Their usage is described in the VestaUserLevel documentation.

8.3 Look and Feel

The UseVesta tool is built on the FormsVBT interface, and so its basic user-interface paradigm is that of FormsVBT. UseVesta, additionally, incorporates several "standard" displays and gestures. These "standards" apply to the ManageReps tool as well.

8.3.1 Over-all Structure

The tool's window (exclusive of Trestle top-bar) is broadly divided into three sections:

- An upper section containing a title and informational status, commands, and a feedback line.
- A typescript sub-window in the middle that displays the stdout-like output of the tool.
- A region below the typescript where transient sub-windows (dialogs) are displayed, a la Ivy.

8.3.2 Pop-Up Menus

Some command buttons in the commands region are really place-holders for pop-up menus. These pop-ups have names that end with a "~".

8.3.3 Sub-windows

Some of the tool's command buttons will cause a dialog (sub-window) to appear. The dialog enables some complicated action, akin to Ivy's "Find" sub-window.

Commands that bring up a dialog have names that end with "..".

Other command buttons optionally cause a dialog to appear. These are commands that initiate an action where a form may or may not be necessary. These commands display a dialog if they are clicked with the option key down; otherwise they perform their standard action.

Commands that bring up a dialog at the user's option have names that end with ",,".

Where a sub-window or dialog appears depends on the mouse-button that is used to bug the command:

- The left button causes a dialog to be inserted into the tool's window, just below the typescript area.

- The middle button causes a FormsVBT "popup" window to be displayed "over" the tool's window.
- The right button produces a new Trestle window. The window will appear "near" the tool's window, and will be in the same tiling/overlapping domain as the tool's window.

All of these sub-windows contain a "Dismiss" button that will cause them to go away.

These sub-windows are separate from one another and several (even several instances of the same generic subwindow) may appear at the same time. [This is different from how naive tools utilize FormsVBT popups.]

8.3.4 Typescript

The typescript subwindow contains a listing of the output of the tool. Most user interactions, especially those that modify system state, record their actions in the typescript.

Lines appended to the typescript are also appended to the file ".TYPESCRIPT" in the session directory.

The command "Empty Typescript" in the "Misc~" popup will empty (clear) the typescript displayed by the tool, without altering the ".TYPESCRIPT" file.

To actually delete the content of the typescript file, perform a "rm <sessiondirectory>.TYPESCRIPT" when an evaluation that appends to it is not active.

8.4 Ordinary Use

8.4.1 Starting

You create an instance of the UseVesta tool by typing

```
UseVesta -Ssessiondirectory
```

to a shell, where the "-Ssessiondirectory" optionally names a session directory to be associated with the instance.

If no session directory is specified, the session directory "." is used.

I find it convenient to associate a separate session directory with each separate package family that I deal with (a la the package tools) and to always start up UseVesta with the appropriate -S flag, running with multiple tool instances if I have multiple packages checked out. Others find it convenient to use only one UseVesta instance, and switch among their packages by dynamically changing the tool instance - session directory association (see 8.11.11, page 167)

8.4.2 The Development Cycle

Suppose that the user wants to make a change to the current package in the family named "application". Here is a basic scenario he could follow:

- Create a session directory for this activity, say the directory "wds/application"
- Obtain (unless he had one), a UseVesta instance associated with that session directory: execute "UseVesta -Swds/application" in a shell.
- Check the package out: Open up the UseVesta instance window, click "Checkout..", fill in "application" in the "Package To Checkout" field (when an explicit package number is not supplied, the tool automatically supplies the latest package in the trunk of the indicated family), and click "Checkout" in the dialog.
- Use his favorite editing paradigm to change one or more source files in the "wds/application" directory. (And in its sub-directories, if the package has sub-models.)
- Click "Advance & Eval" in UseVesta to copy the changed files to the private repository and evaluate the root model.
- Cycle, as necessary, correcting source errors discovered during the evaluation.
- Test the results of evaluation, perhaps via the "Run,," command in UseVesta.

- Check in the package: Click "Checkin..", supply a comment in the "Comment" field in the dialog, and click "Checkin" in the dialog.

Much has been elided in this brief scenario. Details of how the various commands work, how a tool instance is associated with particular public and private repositories, how testing is facilitated, other useful commands, and error reporting and recovery, are covered in the remainder of this document and in other Vesta documentation.

8.5 Session State

At any point in time a UseVesta tool-instance is in one of four states:

- No Session Directory: There is no session directory associated with the tool instance.
- Inactive: Nothing is going on.
- Checked Out: The tool is associated with a session directory that holds a checked-out package.
- Disconnected Development: The tool is associated with a session directory that holds a package undergoing disconnected development. The package and package content exist only in the user's private repository, not in any public repository.

In this document, the last two states are sometimes identified as "active." "If the tool is active ..." is shorthand for "If the state of the tool is Checked Out or Disconnected Development ..." and "An active session ..." is shorthand for "a session with the state Checked Out or Disconnected Development".

These states are connected in a simple state-transition diagram, with specification of a valid session directory moving from the first to one of the others (depending on the content of the session directory), checkout moving from the second to the third, begin-disconnected moving from the second to the fourth, merge-disconnected moving from the fourth to the third, abandon moving from the third to the second, and checkin moving from the third to the second.

The state of a tool-instance, together with information about the package(s) involved, is reflected in the top three lines of the tool's window.

It is important to realize that the state really resides within the session directory. In general, all state information is in the session directory, and the information in the tool is only a copy.

8.6 The Basic Commands

8.6.1 Checkout

To reserve a new package in a public repository, the user commands "Checkout.." and enters, in the form, the name of the package to be reserved and then commands "Checkout" in the form. He can supply a comment to be associated with the new package at this time via a field in the form; or he can wait and supply a comment later.

The new package name must extend the current family without leaving any "holes" in the naming of the package tree. In other words, it must be either of the form <something>.nn where <something>.nn-1 already exists, or must be of the form <somethingnumbered>.<forkname>.0 where <somethingnumbered> already exists.

To reserve a new package in the "main-line" of an existing family, the user can enter just the name of the family. The Checkout command will examine the family's packages in the public repository and supply the correct number for the next package instance.

An outline of the action of Checkout:

- See if the current state is "inactive" and complain and exit if not.
- Look at the current session directory. If it appears to contain valuable files, then complain and exit.
- Look at the supplied package name. If it lacks a trailing ".nnn", then examine the indicated family in the public repository, find the most recent package in the main branch, and set the package name to its successor.
- Using the package name in the "Package to Checkout" field and the user's public and private repositories, perform a checkout. Catch

any lower-level errors and appropriately display them. Checkout includes copying the sources from the parent package's value to the session directory.

- Update the tool's appearance to reflect its new state.

8.6.2 Creating a New Family via Checkout

To create a new family in a public repository (the equivalent of an old createpackage), the user commands "Checkout.." in an inactive UseVesta instance and enters, in the form, the name of the initial package of the desired family (this package name will end ".1") and then commands "Checkout" in the form. He can supply a comment to be associated with the new package at this time via a field in the form; or he can wait and supply a comment later.

This is family creation. To "fork" an existing family, one does a checkout of <existingpackage>.fork.0, as described in 8.6.1, page 146.

8.6.3 Advance

After making a set of edits to a checked out (or disconnected) package, by editing source files in the session directory, the user needs to create a package instance (in his private repository) that captures his new sources in a root model, its submodels, and their leaf-sources.

The user does this by commanding "Advance".

The package created in the private repository is the successor to the one previously created (or the one created at checkout). Thus if the previous package was "text.123.checkout.2" then "text.123.checkout.3" will be created.

An outline of the action of Advance:

- See if the current state is "active" and complain and exit if not.
- Enumerate the session directory tree. For all files encountered, see if they are new or altered by comparing Ultrix timestamps against information remembered in the .V.MTIMES files in the session directory and sub-directories. For new or altered files, write the files

to the repository, and fix up the appropriate DIRECTORY statement in a model with the new UID. Remove lines from DIRECTORY statements that refer to files that are no longer present.

- In all model's IMPORTS clauses, compare the current name, rnpn-surrogate (if any), and the UID against information stored in the .V.IMPORTS files. For a given name: If the rnpn-surrogate changed, update the UID to match, complaining if the rnpn-surrogate doesn't name a UID; if the UID changed, update the rnpn-surrogate similarly; if both changed, compare to see that they agree, and complain otherwise. If there is no rnpn-surrogate, supply one.
- Bind the new root model to the next package instance in the private repository. If nothing was changed, re-write the root model into the repository anyway.
- Update the .V.MTIMES and .V.IMPORTS files to reflect the new state.

While an active session exists, the name of the most recently created package is displayed in the upper part of the tool's window.

8.6.4 Evaluation

After advancing his changes into his private repository, the user commands "Eval,," to evaluate the model, and thus compile his modules. (Or compile and link, or document-process, depending on the nature of the package.)

As a convenience, the "Advance & Eval" command does an advance followed by an evaluation. (This is slightly faster than the two commands in series, because the ordinary "Eval,," command begins by checking that the user doesn't need to do an advance ...)

Also as a convenience, the "Rev & Adv & Eval" command does a revise imports followed by an advance followed by an evaluation. The revise imports specification used is the one named in the "Revise with .." command to the right.

Used normally (no option key), the "Eval,," command evaluates the "test" component of the current (just advanced) model. Evaluation

feedback is presented in the typescript. Messages are preceeded by the location in the model (root model or sub-model) whose evaluation produced them. Error messages are preceeded by "->".

If commanded with the option key down, "Eval,," brings up a dialog that allows the user to evaluate a package other than the current one, evaluate a component suite other than "test", and/or demand that a certain value from the evaluation be displayed.

To evaluate something other than the current model's "test" component, the user edits the top part of the dialog. It appears with "Package" set to the current package in the private repository (the package that would be evaluated with normal use of the "Eval,," command) and with "Component List" set to "test". The user can alter either text field, either to evaluate something other than his current work, or to evaluate a top-level component of the model other than "test", or to evaluate a suite of top-level components. (A suite is entered as a sequence of blank-separated names.) After any alteration of the form, the user begins the evaluation by commanding "Evaluate" in the dialog.

To see the value of an expression in a model, one uses the lower half of the dialog and its "Evaluate and See Value" command. The expression-value to display is described by filling in either the "UID" or the "Session Directory File Name" field, the "Character Location" field, and the "Character Length" field. This is most conveniently done by using the Typed Selection facility of Trestle: select the expression in an editor's window (an editor that understands about the typed selection, e.g. Facades or Ivy) and click the "Fill In From Selection" command in the dialog. If the user turns on the dialog's "Exact Match Only" boolean, then only an expression that precisely matches the character location and length will be displayed; otherwise, any expression contained within that character sequence will be displayed.

8.6.5 Checkin

To checkin a package, the equivalent of an old shippackage (or make install), the user commands "Checkin..".

This command brings up a Ivy-style form. The form contains the name of the package, a list of components to evaluate, the current comment, and a statement about the package's checkin actions, if any. The form also contains a "Checkin" button.

The user can (perhaps) alter the component list and comment and then use the "Checkin" button. Or he can examine the situation and dismiss the form.

An outline of the action of Checkin:

- See if any source files in the session directory have been changed since the last time they were copied to the private directory. If so tell the user that "You have changes that haven't been 'advanced" and exit.
- Evaluate the package's listed components with a special form of evaluation that remembers the derived objects produced. This is so that the checkin can copy those objects to the public repository.
- Perform a Checkin via VestaUserLevel.
- Remove the checked-in sources from the session directory.

8.7 Package Comments

Packages in repositories may have comments associated with them. A package's comment is a (presumed fairly short) user-supplied string that describes or annotates the package.

As packages are generated by Advance and by Checkin, they are given comments. Rather than have the comment supplied on each call, the comment is stored in the session directory.

The pending-comment may be set via the sub-form that is presented at Checkout or CreateDisconnected or MergeDisconnected time, or via the "Change Comment.." command in the "Cmds~" pop-up menu.

At checkin time, the Checkin dialog displays the comment and offers the user an opportunity to edit it.

8.8 Checkin Actions

Associated with a package, there may be a suite of "checkin actions." These are specal actions that should be performed when the package is

checked in: typically they will involve extra-Vesta activities, such as copying executables into /vesta/out.

A package has checkin actions to be performed if its root model contains a top-level component named "preCheckin" or "postCheckin." The preCheckin component is evaluated before the actual checkin is performed; the postCheckin component is evaluated afterward.

8.9 Disconnected Development

Vesta supports the notion of "disconnected development." This allows the use of a private repository in support of the development cycle, without any connection with a real family in a public repository.

There isn't a direct analogy between disconnected development and something in the getpackage-shippackage world, because the latter doesn't need the notion: "disconnected" in the getpackage world translates to "just in a directory somewhere." But in Vesta, evaluation requires that the model be in a repository, hence disconnected development.

A disconnected "session" is created by the command "Create Disconnected.." in the "Cmds~" popup. It is analogous to the "Checkout.." command, taking the name of the package from a sub-form.

An outline of the action of Create Disconnected:

- See if the current state is "inactive" and complain and exit if not.
- Look at the current session directory. If it appears to contain valuable files, then complain and exit.
- Using the package name in the "Package to Create in Private Repository" field and the user's private repository, perform a CreateDisconnected via VestaUserLevel. Catch any errors and appropriately display them.
- Update the tool's appearance to reflect its new state.

Later, the user may wish his development to become "connected," that is to look like a development cycle initiated with a "checkout" action. This is accomplished with the "Merge Disconnected.." command in the "Cmds~" popup.

An outline of the action of MergeDisconnected:

- Checkout the supplied full package name into the private repository indicated by the supplied "mergee". Complain if the checkout fails.
- Assign the supplied "mergee" as the private-repository partner of the checked out package.

The "checkout" performed by MergeDisconnected does not create a family for the user if he supplies an RNPN of the form "family.1". A NergeDisconnected must be done into an existing family.

A CreateDisconnected followed by a MergeDisconnected is essentially equvalent to a Checkout. The only inherent difference is that the development fork in the private repository is named "disconnected" rather than "checkout." A probable difference is that if the user does a CreateDisconnected, and then a Fetch & Advance, the source files in the package will be newly timestamped, whereas on a Checkout, the original timestamps are preserved.

8.10 Running an Executable

The "Run,," command in UseVesta enables the user to run the executable associated with an "application" model.

The Run dialog (obtained with the option key, see 8.3.3, page 142), enables you to

- Enter the package name of the model.
- Enter the name of the executable within the model.
- Enter command line arguments that should be appended to the run line.
- Indicate whether to start the executable under MM debugger or under the Vulcan debugger.
- For the latter, whether a "-r" (do not stop early in execution) flag should be added.
- And then command the execution.

The dialog is initialized with a package name of the current package and with an executable name equal to the current package's family.

If you just click the command (no option key) and there is a "current package" then the executable whose name is identical to the family name of the package will be run without arguments and not under the debuggger.

The mechanism uses OS.StartProcess to create an "orphan" process. It manifests a filename for the executable by extracting the binding within the "test" component that's the same name as the package's family (e.g. the binding "easter" in the package "easter.2"). It starts the file under /proj/topaz/bin/DebugProcess or /proj/topaz/bin/vdebug if debugging is requested.

The started process is given its own window by starting it under "nw". The new window is given the name "Run: <executable>" via the -n option of nw.

As a debugging aid, the file /tmp/UseVesta-Run-Recent is symbolic-linked to the executable in the repository if the OS.StartProcess succeeds.

The user's supplied arguments are parsed with the ParseShellCommand interface.

The command locates the executable by scanning the evaluated model. So the model must match the "standard" application model in structure. The example model for "easter" is a standard application model.

8.11 Secondary Commands

8.11.1 Abandon

The "Abandon" command in the "Cmds~" popup allows the client to end a checkout session without checking anything in.

The session is ended without result. Looking at the package, later, in the public repository, one will see that the package is bound to the root-model NIL.

Abandon, like all commands that return the tool's state to "inactive," may delete all the source files from the session directory. (This depends

on the user's profile: see 8.11.7, page 162.) Unlike Checkin, however, since Abandon potentially loses user edits, it requires confirmation, and pops up a subform for that purpose.

Since Abandon sets the root-model of the package family to NIL, it inhibits development along that fork. (A subsequent checkout will succeed, but will retrieve no sources.) This may be appropriate, say if one is killing off an uninteresting forked development branch. In other circumstances, e.g. you checked out a main-trunk package, and then decided that you shouldn't have, the abandoned package and its NIL model may be inappropriate. An alternative to abandoning a checked out package is to use the "Fetch.." command to restore the package's content to match its parent's content (if operating on text.78, do a fetch from text.77), and then do an ordinary checkin, with an appropriate comment.

8.11.2 Re-establishing a Session Directory

The "Connect Session To Old Checkout.." command in the "Cmds~" popup allows the client recover if a session directory is damaged – either by the system or by an inadvertant user action, such as an "rm".

The command brings up a dialog. The user should enter the package name of the checked-out (or disconnected) package that should be associated with the current (inactive) session directory. A checked-out package is entered into the first type-in field ("Checked Out Package to Associate with SD"), a disconnected package is entered into the second type-in field ("Disconnected Package to Associate with SD").

Clicking "Connect" in the dialog will then proceed to make the session directory be a SD associated with the package. The special UseVesta files in the directory (the ".V..." files) will be recreated based on package information found in the repository(s). The source files will be recreated from the latest possible version of the package: either the parent package in the public repository or the latest version in the user's private repository.

You can only connect to a package that you checked out or created as disconnected. There is no way for UseVesta to acertain that a different session directory is not connected to the package; if one is, the user may becme confused later.

8.11.3 Prettyprinting

The prettyprint function of UseVesta enables the user to reformat the models in his session directory.

The "Prettyprint Models,," command is in the "Cmds~" popup. The default behavior of the command is to prettyprint all of the models and sub-models in the session directory. With the option key (see 8.3.3, page 142), the user is presented with a form that will let him restrict the prettyprinting to only the top-level model.

Prettyprinting involves the Vesta evaluator. Because of this, prettyprinting requires that the session directory be reflected in the repository: there must have been an "Advance" after the most recent source changes.

The prettyprint functionality in UseVesta is currently disabled, pending changes to the underlying mechanism in the Vesta evaluator.

8.11.4 Model Recreation

The model recreation function of UseVesta enables the user to recreate, that is reconstruct, a model from the information present in the sources in the session directory.

The "Recreate Models,," command is in the "Misc~" popup. The default behavior of the command is to recreate all of the models and sub-models in the session directory.

With the option key (see 8.3.3, page 142), the user is presented with a dialog that will let him restrict the recreation to only the top-level model, and will let him specify a template file that should be used instead of the original model's body.

There are two template files supplied as a part of the tool. "Application.template" is appropriate for an "application" model that creates an executable. "Library.template" is appropriate for a "library" model that defines an "intfs" and an "impls".

Model recreation proceeds by examining the sources in the directory, together with the model body (or template).

The model body contains some number of specialized comments. These comments are on separate lines, and define replacement operations to the recreation process:

```
(*GENERATE-BEGIN SortedDefs*)
(*GENERATE-END SortedDefs*)
```

The lines between this pair of comments is replaced by a list of directives for the compilation of interfaces. An interface is defined as a source whose name ends in ".def". The interface sources are scanned, assuming that they are Modula source programs, and their IMPORTS statements are used to create an appropriate compilation ordering.

```
(*GENERATE-BEGIN Mods*)
(*GENERATE-END Mods*)
```

The lines between this pair of comments is replaced by a list of directives for the compilation of implementations. An implementation is defined as a source whose name ends in ".mod".

Additionally, a reference to the source "buildingenv" is forced into the imports part of the model's directory statement.

8.11.5 Revise Imports

The "Revise Imports" function of UseVesta updates the UIDs that appear in the IMPORTS part of session directory models.

This capability enables the user to easily arrange to, e.g, update his models to reflect a new version of a public interface.

The "Revise with ..." command names the revise specification that it will use. This spec is the source file in the session directory (and hence in the package) with the suffix ".revisespec," if one exists, or the specification contained in the user's profile (8.11.7, page 162).

The default revise specification (named "all-last.revisespec") updates every import to the latest checked-in package in the trunk of the package in the public repository.

The "Revise.." command is in the "Cmds~" popup. This command presents the user with a form that will let him specify all models versus only the root model, and name his specification file.

In the form, the last three lines are a "choice" item set. The first bulleted item is a type-in region where the user can enter an arbitrary name. The second and third bulleted items provide the names of the two pre-defined specifications "all-last.revisespec" and "mine-last.revisespec" for the user's convenience.

The user should set the "Root Only" boolean as appropriate, supply (type-in) his own name in the first bulleted line or select the second or third bullet, and then click the"Revise Imports" command in the upper-left corner.

A user-created specification file can be tailored in many ways. It can update his imports to reflect the repository state as of a particular date; it can treat some imports specially, perhaps updating them to refer to the user's private repository; it can follow forked development branches.

Before passing a specification to the revision procedure, UseVesta alters it by a) replacing all occurrences of the string "$USER" with the user's logon name b) replacing all occurrences of the string "$PRIVATEREPOSITORY" with the canonical name (expanded out to the full path) of the current private repository, and c) replacing all occurrences of the string "$PUBLICREPOSITORY" with the canonical name of the current public repository. These substitutions enable "generic" specifications: one can write a specification that says "update only packages in my private repository" without having to create a separate specification for every user.

The revise specification syntax:

```
specification  ::=   *spec-entry
spec-entry     ::=   comment | rewrite
comment        ::=   ''#'' *<non-new-line> new-line
rewrite        ::=   pattern => replacement '';''
replacement    ::=   absolute relative
absolute       ::=   +term
term           ::=   identifier | string literal
relative       ::=   *direction
direction      ::=   UPBRANCH [TO term] | DOWNBRANCH term |
                     ADVANCE [FOLLOW (ME | ANYONE)]
```

```
                        [TO (version |  LAST | time)] |
                      PARTNER
time            ::=   string literal | AFTER absolute
version         ::=   cardinal
```

White-space can appear in all the customary places. Comments can appear only where the grammar permits them. Text literals and numbers are defined as in the Vesta language specification.

ReviseImports considers each entry in the IMPORT section of a model and performs a sequence of actions, detailed below. The effect of these actions is either to leave the entry unchanged or to replace its UID portion with a different UID.

At several points in the subsequent description, the phrase "it is an error if ..." appears. In all cases, this is shorthand for "UseVesta displays a suitable error message and ceases to process the IMPORT entry under consideration."

For each IMPORT entry, ReviseImports extracts the UID and calls VestaUID.ToRNPN. It is an error if VestaUID.ToRNPN raises an error. VestaUID.ToRNPN returns an RNPN. For each <rewrite> in the input <specification>, ReviseImports attempts to match the <pattern> portion to the RNPN under consideration. The <rewrite> rules are tried in the order that the <rewrite> rules appear in the <specification>. (Strictly speaking, the <pattern> is matched against the string formed by appending a new-line character to the RNPN under consideration.) If a match occurs, ReviseImports uses the <replacement> part to compute a UID that will replace the one in the IMPORT entry under consideration. If none of the <rewrite> rules matches the RNPN, ReviseImports leaves the IMPORT entry unchanged and continues with the next one, if any.

The replacement part consists of a mandatory <absolute> part and a possibly-empty <relative> part. The <absolute> part is a simple string concatenation of terms, each of which is either a literal or is an identifier defined by the <pattern> part of the <rewrite> rule. (Recall that patterns(7) includes a mechanism for binding pieces of the matched character string to identifiers. Those identifiers can appear as terms in the <absolute> specification.) The evaluation of the <absolute> part produces a string, which is expected to be an RNPN.

The <relative> part is interpreted as a sequence of transformations, each of which converts an RNPN to another RNPN. In every case, the input to

a transformation must be a defined RNPN. (Note that a non-NIL result is not the same as a bound RNPN: the latter is a particular case of the former.) We adopt the notation that the PN portion of the input RNPN is a path P comprising a sequence of path steps P1,...,Pn, where each Pi consists of a branch-version pair <Bi,Vi>. The semantics of the individual transformations is as follows:

UPBRANCH

The output of the transformation has the same RN as the input. The output PN is the path P' = P1,...,Pn-1. It is an error if n = 1.

UPBRANCH [TO term]

The output of the transformation has the same RN as the input. The output PN is the path P' = P1,...,Pk where 1 <= k < n and Bk = term. It is an error if no such k exists.

DOWNBRANCH term

The output of the transformation has the same RN as the input. The output PN is the path P' = P1,...,Pn,Pn+1 where Bn+1 = term and Vn+1 = 0.

ADVANCE

The output of the transformation has the same RN as the input. The output PN is the path P' = P1,...,Pn' where Pn' = <Bn,(Vn)+1>

ADVANCE TO (version | LAST | time)

The output of the transformation has the same RN as the input. The output PN is the path P' = P1,...,Pn' where Pn' = <Bn,Vn'>, and Vn' is chosen based on the construct following "TO" as follows:

- TO version: Vn' is set to 'version'.
- TO LAST: Vn' is the largest number such that P' is a bound name in the specified repository. If there is no such Vn', then Vn' is set to Vn.
- TO "time": Vn' is the largest number such that P' is a bound name in the specified repository and 'info.boundAt' is less than or equal to "time". ("time" is something understandable by TimeConv.ScanTime.) If there is no such Vn', then Vn' is set to Vn.
- TO AFTER absolute: <absolute> is a bound RNPN. Vn' is the smallest number such that P' is a bound name in the specified repository and 'info.boundAt' is greater than 'aInfo.boundAt'. If there is no such Vn', then Vn' is set to Vn.

FOLLOW (ME | ANYONE)

This modifier of ADVANCE says to extend the sequence of package names being considered. The P' do not necessarily stop at a non-bound package. The phrase "The output PN is the path P' = P1,...,Pn' where Pn' = <Bn,Vn'>" in the above is extended to be "either the path P' = P1,...,Pn' or, if some path P" = P1,...,Pn" is Unbound, the path P' = Partner[P"].checkout.n". In other words, if a P is Unbound (checked out), then the revise operation continues to consider the packages partner[P"].checkout.0, partner[P"].checkout.1, etc. If ME is present, then this only occurs if the unbound P" is checked out to the current $USER. Saying "ANYONE" is almost certainly incorrect unless the specification is constrained to operate on a very few family trees: notice that an advance with FOLLOW can easily advance to a package that has been bound in a private repository but not yet evaluated.

PARTNER

The input RNPN must exist and be unbound. It is an error if this is not the case. The output RNPN is 'info.partner'.

After the entire <relative> part (possibly empty) has been interpreted, the final RNPN must be bound. It is an error if this is not the case. The value 'info.value' then replaces the UID in the IMPORT entry under consideration.

8.11.6 RNPN Surrogates

The UseVesta tool, to aid the user's reading of models, maintains a stylized comment, called an rnpn-surrogate, in the entries in IMPORTS clauses.

A entry in an imports clause with an rnpn-surrogate takes the form

```
name (* rnpn-surrogate *) ~ uid,
```

where the white space is the user's. The rnpn-surrogate supplies the rnpn that the uid is the value of.

The rnpn-surrogate takes one of four forms:

- A full rnpn, e.g. /-/com/dec/src/vesta/proj/vestauser.10
- A standard rnpn, e.g. /vesta/proj/vestauser.10
- A package name, e.g. vestauser.10
- A package name with the family elided, e.g. .10

The third and fourth forms are only used in circumstances where the rnpn can be recovered from the information in the imports entry. The family can only be elided if the name in the imports line is of the form "family.v" The repository can be elided only if it is the repository that's the one that gave out the uid.

When UseVesta (the Advance operation) creates an rnpn-surrogate, it creates a "standard rnpn" form. When UseVesta modifies an rnpn-surrogate, it tries to keep it in the form it was, retreating toward "full rnpn" only as necessary to maintain the above rules.

When an IMPORTS entry is modified, only the rnpn-surrogate or the UID is modified. All user whitespace is retained. Thus the user can decide, for example, whether there should be blanks between the "*" characters and the rnpn-surrogate or not.

8.11.7 The Profile

UseVesta can be customized, a bit, by the user.

A number of parameters govern UseVesta's actions. These parameters are initially read in from a *profile file* augmented by a *set of profile modifications*.

The profile file is "UseVesta.profile" in the release. The profile modifications are the file ".UseVestaProfile.sx" in the directory "." or in the user's home directory: tool initialization tries to find the file first in "." and then in home.

The user can alter or substitute for the profile file, or, more easily, store profile modifications. To substitute for the profile, see the discussion of FormsVBT and resources in 8.14, page 169.

The user can also alter the current profile parameters at run-time.

To examine and alter the profile parameters, one uses the "Exhibit Profile.." command in the "Cmds~" popup. The resulting form sub-window displays the current profile parameters: the state of the profile, not the state of the tool.

Command buttons in the profile form enable:

`Reflect Current Tool State -> Here`

Changes the profile parameters in the form to match the values stored in the tool-instance. The profile initially appears with the values that a new tool-instance would have, not those that this tool-instance currently has.

`These Values -$>$ Current Tool State`

Changes the profile parameters stored in the tool-instance to match the (presumably just edited) values in the form.

`Save Profile Info in .UseVestaProfile.sx`

Stores the profile parameters of the form into the named file, thus making the changes permanent.

The file ".UseVestaProfile.sx" will be in the directory "." if such a file was found there upon startup. Otherwise, the file will be stored in the user's home directory.

A typical usage is either to a) alter the stored profile by making changes to the form and commanding "Save Profile Info ..." or to b) alter the tool's state by commanding "Reflect Current Tool State -> Here", making changes, and commanding "These Values -> Current Tool State".

The profile items are these:

Revise Specification

The text-name of the user's revise specification. See 8.11.5, page 156.

Do Not Display Name in Error Report

This boolean governs whether UseVesta supplies the source-name of an offending source when displaying an evaluation error to the user. Since the evaluator only supplied the UID of the offending source, acertaining the actual source name can be expensive.

Try for Partner Package on Visit

This boolean governs whether the UseVesta "Visit" command will, given a visitee of the form "/publicrep/packagename", follow the "partner" field of an unbound package to a private repository, and then follow the "checkout" line of packages to the most recent version of the family.

Print Result of Evaluation

This boolean controls whether a few lines describing the result of the evaluation is displayed whenever UseVesta performs a successful evaluation.

Print useful Evaluation Derived's UIDs

This boolean controls whether, after an evaluation (successful or not), UseVesta prints the names and UIDs of the derived objects created (or examined) during the evaluation.

This can be very useful if you want to see an intermediate result, e.g. you want to examine the output of a Flume call.

The printing of these UIDs has bee disabled in the current UseVesta.

Test For Dangling Imports at Checkin

It is often an error for the user to checkin a package when the package's model's IMPORTS clauses refer to other models that are still in private repositories.

This boolean controls whether or not UseVesta checks for this condition and complains if it's so.

Display Times Short

This boolean controls how UseVesta's time displays are formatted. (Many result-lines in the typescript are time-stamped.)

Times displayed are the result of calling TimeConv.TimeLocalToText with Time.Now() and the value of this boolean.

Display Time In All Evaluation Messages

If this boolean is on, a short "(@12:34)" is appended to many UseVesta messages.

Display Vesta Stack In Evaluation Error Messages

If this boolean is on, then an evaluation error message is preceeded, not only by the model-location of the error, but by a back-trace through the Vesta Evaluator's model-stack.

Preserve SD Content After Checkin
Clear SD Content Before Checkout

This two booleans controls what happens to files in the session directory upon checkout, checkin, and similar actions, e.g. begin disconnected.

The default for both booleans is false. In that state, the session directory is cleared of source on checkin, and tested on checkout to ensure that it is empty of source.

If "Preserve SD Content After Checkin" is true, then the session directory is not cleared of source after a checkin.

If "Clear SD Content Before Checkout" is true, then the session directory is cleared of any source at the start of a checkout.

On Filename Match at Advance with RegExpr ...

This text field and the four radio buttons on the following line control a "tidy" operation that occurs at "Advance" time.

If the supplied regular expression is non-empty, then each (apparent) source file in the session directory is matched against it. The actual text-name to be used in the match (full path versus only the name in the last directory) is controlled by the third and fourth radio buttons. The action to take on a successful match (delete or complain) is controlled by the first two radio buttons.

8.11.8 Visiting Source Files

The "Visit" command displays a source file from the repository. It uses the current selection as the name of the source to display. The selection can be a package name; if so, the model associated with the package instance is displayed. Or the selection can be a UID; if so, that source is displayed. UseVesta will try to find a UID by stripping leading and trailing text from the selection. This enables the user to select an entire line containing a UID (say a line in a system model's directory) and use "Visit".

The "Visit" command in the tool's header creates a new sub-window, while the "Visit" command that appears in such created sub-windows pushes the new display onto the sub-window where "Visit" was clicked: the "Previous" command uncovers the previous display.

The "Visit via $EDITOR" command displays a source file from the repository. But instead of creating a FormsVBT sub-window containing the file, it creates (in a separate Trestle window, regardless of which mouse button it is invoked with) an instance of the user's editor (as indicated by the environment variable $EDITOR) operating on the file.

8.11.9 Fetching

The "Fetch.." command in the "Misc~" popup enables the user to copy a suite of source to the session directory. The command brings up a fill-in form that allows the user to specify:

- "Fetch from:" A package to fetch from. The sources of the package will be copied to the session directory.
- "GetPackage Style ..." Whether this is a getpackage-style fetch operation. If so, files from the "fetch from" source and the session directory are time-stamp compared: new files and newer files are copied, equal-timestamp files are not copied, older files are not copied and a warning given, files in the session directory but not in the "fetch from" are retained with a warning. If not, the session directory must be empty (see deleting current sources, below) and the files are then copied.
- "Delete Existing Sources First" Whether this fetch operation should begin by deleting the current sources.
- "Verbose" Whether verbose output should be supplied. Detailed output is only applicable to the "getpackage style" option. With "Verbose" all files involved are listed with their status, while without "Verbose" only files involved in warnings are listed.

and then to command:

- "Fetch & Advance" to get new sources into an active session, and "Advance" them into the private repository or

- "Fetch Into Idle SD" to get the new sources into the session directory even though there is no active session. (Perhaps the user wishes to just examine some of the sources. An alternative to this fetch would be to use the capabilities of Vesta's Echo clerk and examine the files directly.)

8.11.10 Changing Context

The "Change Context (reps etc).." command in the "Misc~" popup enables the user to redefine his public repository, private repository, and revise specification at runtime.

The user makes a change by editing the values in the dialog (which are initialized to the currnt values) and commanding "Set".

The repositories cannot be redefined if the the tool's state is an active one.

8.11.11 Changing Session Directory

The "Change Session Directory.." command in the "Misc~" popup enables the user to switch a tool instance among session directories at runtime.

The command brings up a dialog with a "Session Directory" field set to the current session directory.

To switch to an existing directory, the user edits the field and then changes session directories by typing a return or by commanding "Switch" in the dialog. The latter dismisses the dialog, the former leaves it up.

To switch to a directory that does not yet exist, one can either create it with "mkdir" in a shell somewhere, and then proceed as above, or one can enter the name of the new directory and use the "Create Session Directory and Switch" command in the dialog. This command will make a new directory and switch to it. It will not make a complete path: if you ask to create and switch to "a/b/c/d/e/f", "a/b/c/d/e" must already exist.

8.11.12 Showing Session Directory

The "Show Session Directory" command in the "Misc~" popup displays a listing of the session directory tree-of-files.

Sub-directory content is displayed indented, to make the tree manifest.

If a source file is has been changed, that is the session directory copy is newer tha the repository copy, and an Advance operation would write it back, then "(changed)" is displayed after the file name.

8.11.13 Record Time Command

The "Record Time" command in the "Misc~" popup writes the current time to the typescript.

8.11.14 Test 'pending' in Session Directory Command

The "Test 'pending' in Session Directory" command in the "Misc~" popup interrogates the state of the session directory. If there is a "pending" command, then UseVesta performs the actions that it would ordinarily do if the user attempted a "real" command while the session directory indicated that a previous command had been interrupted while it was in progress. See 8.13, page 168 for a discussion of this.

8.12 Error Reports

The UseVesta tool catches lower-level error conditions and maps them into error strings suitable for display to the user.

For some lower-level error indications, the implementation investigates (via the low-level interfaces) in order to be able to specify the difficulty more precisely.

All errors are reported in the feedback area. Errors which affected a state-changing action, or which represent a serious condition, e.g. an anomaly in a repository, are also reported in the typescript.

8.13 Interrupted Commands

When it begins an operation that affects repository (and hence session directory) state, UseVesta writes intentions information into the session

directory so it can recover after an abort or crash. This information is written into the file ".V.PENDING".

When UseVesta associates a tool-instance with a session directory, either upon start-up or upon a "Change Session Directory" command from the user, it checks the ".V.PENDING" file to acertain if the session directory contains information about a partially-completed command. If it does, UseVesta brings up a large pop-up form that contains discussion of the problem. This pop-up invites the user to ask for a "Compare." The latter invokes an examination of the session directory and the repository(s), based on the command that was interrupted, and will bring up a second form that suggests a course of action. [These two forms are filled in with info while they are visible: ignore the initial confusion.]

Several UseVesta commands check for partially-completed commands, and complain if the session directory is in such a state. If the user gets such a complaint ("Session Directory Contains 'Pending' Command Info!"), he can trigger the pop-ups that offer discussion and analysis via the "Test 'pending' in Session Directory" command in the "Misc" command pop-up.

8.14 Resources

Vesta supplies a facility, Bundle, that looks up package-related data files within the repository.

FormsVBT supplies a facility, Rsrc, that looks up package-related data files in the Unix file space.

UseVesta uses both these facilities. It first uses Rsrc to find a file of the user's. Failing to find one, it uses Bundle to get the data file from the repository.

To substitute a file for any UseVesta package data file, the user inserts his file, with the same name as the package's file, into the Rsrc path list.

The UseVesta Rsrc path list is

- vestausertools/forms/UseVestaDir in ".", the directory that UseVesta was started in
- vestausertools/forms/UseVestaDir in the user's home directory

8.15 Special Commands

Several special or testing commands are placed in the "Beware~" popup.

Warning: The commands in "Beware~" may be obscure or flakey. Several operate in event time even though they are long-running.

- The "Empty Sources" command deletes all of the source files in the session directory.
- The "Create SD From /proj.." command takes a package in /proj and copies its sources to the session directory. The goal is to automate the initial creation of a Vesta package from existing sources. The command brings up a fill-in form. The user supplies 1) the full package name 2) a "Go" filter: sources that do not match this regular expression are ignored 3) a "NoGo" filter: sources that do match this filter are ignored 4) booleans that affect the operation: "Erase First" clears the session directory of source before copying, "Report Filtering Etc" produces a verbose typescript. Bugging the "Create" commmand in the form then executes the copying operation.
- The "Test Alert Late in VUL" and "Do Not Test Alert Late" commands toggle a boolean in the VestaUserLevel implementation. If the boolean is on, then Thread.Alerted is tested (and Alert raised as appropriate) very late in the execution of VestaUserLevel procedures, such as Checkout. This makes it easier to debug alerts.
- The "Show Selection" commands displays the current selection as a Text.T and as a TypedSelection.SourceAndPosition. This can aid the user in finding the character position of a token in a file.

8.16 Startup

One creates a UseVesta tool instance via "UseVesta [-Ssessiondirectory]" in a shell.

There should be a Vesta server running on the workstation prior to starting UseVesta. (Start a server via "startvesta,, and wait for the log-file window or icon to appear.)

The -S argument specifies a session directory. If it is omitted, the UseVesta instance will initially be connected to the session directory "."

If there is an active checkout session's data in the session directory, then the UseVesta instance's repositories will be those associated with the session. If there is an active disconnected session's data in the session directory, then the UseVesta instance's private repository will be that associated with the session.

If the public repository is not set by the above, it will be the environment variable VPUBLIC, if present, or else "/vesta/proj".

If the private repository is not set by the above, it will be the environment variable VPRIVATE, if present, or else "/user/$USER/vesta/private".

9. ManageReps

9.1 Introduction

ManageReps is a window-based and FormsVBT-based tool that enables the user to deal with Vesta repositories: create and delete them, obtain listings of families, information about packages, etc.

It relies on lower-level interfaces, VestaRep and VestaRepmgmt, to actually perform much of the work.

9.2 Look and Feel

The ManageReps tool is built on the FormsVBT interface, and so its basic user-interface paradigm is that of FormsVBT. ManageReps, additionally, incorporates several "standard" displays and gestures.

9.2.1 Over-all Structure

The tool's window is broadly divided into four sections:

- An upper section containing a title and informational status (the name of the repository that ManageReps is currently connected to), commands, and a feedback line.
- A region where sub-windows displaying particular requested information appear.
- A typescript sub-window in the middle that displays the stdout-like output of the tool. (Unlike UseVesta, there is little such output from ManageReps.)

- A region below the typescript where transient sub-windows are displayed, a la Ivy.

9.2.2 Pop-Up Menus

Some command buttons in the commands region are really place-holders for pop-up menus. These pop-ups are located at the top right hand side of the region, and the displayed names end with a "~": "List Reps~", "Alter Rep~", and "Switch Rep~".

9.2.3 Sub-windows

Some of the tool's command buttons will cause a sub-window to appear. The sub-window may display data to the user, or may be a form that enables some complicated action, akin to Ivy's "Find" sub-window.

Commands that bring up a form-fill-in sub-window, a la Ivy, have names that end with "..".

Where a sub-window appears depends on the mouse-button that is used to bug the command:

- The left button causes a sub-window to be inserted into the tool's window, just above (display sub-windows) or below (forms-fill-in sub-windows) the typescript area.
- The middle button causes a FormsVBT "popup" window to be displayed "over" the tool's window.
- The right button produces a new Trestle window. The window will appear "near" the tool's window, and will be in the same tiling/overlapping domain as the tool's window.

All of these sub-windows contain a "Shut" button that will cause them to go away: see 9.10, page 178.

These sub-windows are separate from one another and several (even several instances of the same generic subwindow) may appear at the same time. [This is different from how naive tools utilize FormsVBT popups.]

9.2.4 Typescript

The typescript subwindow contains a listing of the output of the tool. Many user interactions, especially those that modify system state, record their actions in the typescript subwindow.

9.3 Learning About Repositories

The "" pop-up anchor provides a way to acertain the names of existing public and private repositories. Within the pop-up, "Public" lists the current public repositories, "My Private" lists the private rpositories of the user, and "All Private" lists the private repositories of all users.

Within a private repositories listing sub-window, the user will probably find the "Grep" command useful: see 9.10, page 178.

9.4 Switching Repositories

The ManageReps tool's attention is always directed to one repository: the repository named in the title at the top-left of the tool's window.

The user chages that repository with the "Switch Rep~" command.

The Switch Rep command brings up a menu with four commands. The first, "To /vesta/proj", switches to the standard SRC public repository. The second, "To /user/<user>/vesta/private", switches to the standard private repository of the user. The third, "To Selected Name", switches to the repository (public or private) named by the current selection. The last, "Refresh Current Display", updates the display of the current repository to reflect its current state.

ManageReps tries to keep the top sub-window displaying the content of the current repository. So, unless that sub-window has been shut, a repository switch will refill the sub-window with a display of "Public rep <name> as of <date>" or "Private rep <name> as of <date>". If there is no sub-window displaying the current repository, a repository switch will create one.

When ManageReps is given a family or package name (without a repository prefix), it uses the current repository. This can be confusing

after a switch: if you are looking at, say, the family 'fred" in repository /private, the subwindow shows you a tree, with "fred.1", "fred.2", etc. If you now switch to the repository /different, select "fred.2" and ask to see that package (say with the Show command), ManageReps will try to display /different/fred.2, which is probably not what you had in mind.

9.5 Creating Repositories

To create a repository, one clicks the "Create Rep.." command and fills in the form.

The form contains a "Repository Name" field. It should be set to the name of the repository to be created. The standard name of a private repository is /user/$USER/vesta/private if the user ("$USER") has only one, and /user/$USER/vesta/<something mnemonic> if the user has several.

The form contains a "Repository's Home Replica" field. It should be set to exactly "src".

Then a public or private repository can be created by clicking the "Create Private Rep" or the "Create Public Rep" (are you sure?) button.

9.6 Deleting Repositories

To delete the current (private) repository, one brings up the "Alter Rep ~" popup and invokes the "Delete.." command. (As a small safety precaution, you can only delete the repository that you are looking at.)

This command brings up a form that displays the name of the repository to be deleted and asks for confirmation.

Commanding "Delete" will (attempt to) delete the repository. There are a number of checks made on the repository: it must be private, owned by the user, and not contain any outstanding checkout sessions. If the delete operation fails, an appropriate explanation is presented.

9.7 Looking at Things: Repository Data

Several forms of a "Show" command enable rudimentary browsing within a repository. The "Show" command in the tools' header takes the current selection, parses it (crudely) to see whether it is a repository name, a family name, or a package name, and then brings up a new sub-window containing a view of the repository content, the listing of the family-tree, or a description of the package (e.g. who created it).

The "Show Rserved" command in the tools' header brings up a display of the packages that are checked out (by anyone) from the public repository.

Within a sub-window created by the command, there is a "Show" command. It operates similarly to the "Show" command in the header, except instead of creating a new sub-window, it pushes its display onto the sub-window where the Show button was clicked. The new display may be "popped" via the "Prev" button: see /xrprevprev.

Within the sub-window that displays the families within the current repository, the "Show Last" command shows the last package in the main trunk of the selected family.

9.8 Looking at Things: Sources in Repository

The "Visit" command displays a source file in the repository. It uses the current selection as the name of the source to display. The selection can be a package name; if so, the model associated with the package instance is displayed. Or the selection cvan be a family name; if so, the model associated with the last-bound package instance in the main trunk of the family is displayed Or the selection can be a UID; if so, that source is displayed. ManageReps will try to find a UID by stripping leading and trailing text from the selection. This enables the user to select an entire line containing a UID (say a line in a system model's directory) and use "Visit".

Akin to "Show", the "Visit" command in the tool's header creates a new sub-window, while the "Visit" command that appears in such created sub-windows pushes the new display onto the sub-window where "Visit" was clicked.

The "Visit via EDITOR" command is similar, but it brings up a new window (using the editor named via the $EDITOR environment variable) on the source file.

9.9 Looking at Things: Learning About a UID

The "Examine UID.." command brings up a dialog. The dialog enables the user to type-in a UID and ask for information about it; or to ask for information about the selected UID.

The information is displayed in the dialog. It includes the length of the object in bytes, its create-time, and, if it appears to be text, the first few lines of its content.

9.10 Commands Within a Sub-Window

Several command buttons within a (transient) sub-window aid the user.

The "Show" sub-window command has already been described in 9.7, page 177.

The "Grep" command creates a sub-window (which covers the current one) that contains exactly the lines of the current sub-window that contain the current selection as a sub-string.

The "Prev" command "pops" the sub-window, displaying the sub-window underneath, if any. (For example, restoring to view the sub-window covered by a use of "Grep".)

The "Shut" command removes the sub-window and any sub-windows "underneath" it, and re-allocates the vertical space among the remaining sub-windows.

9.11 Error Reports

The ManageReps tool catches lower-level error conditions and maps them into error strings suitable for display to the user.

For some lower-level error indications, the implementation investigates (via the low-level interfaces) in order to be able to specify the difficulty more precisely.

All errors are reported in the feedback area. Important errors (e.g. ones which seem likely to be a repository problem, rather than a simple user typing error) are also reported in the typescript.

9.12 Resources

Vesta supplies a facility, Bundle, that looks up package-related data files within the repository.

FormsVBT supplies a facility, Rsrc, that looks up package-related data files in the Unix file space.

ManageReps uses both these facilities. It first uses Rsrc to find a file of the user's. Failing to find one, it uses Bundle to get the data file from the repository.

To substitute a file for any ManageReps package data file, the user inserts his file, with the same name as the package's file, into the Rsrc path list.

The ManageReps Rsrc path list is

- vestausertools/forms/ManageRepsDir in ".", the directory that ManageReps was started in
- vestausertools/forms/UseVestaDir in ".", the directory that ManageReps was started in
- vestausertools/forms/ManageRepsDir in the user's home directory
- vestausertools/forms/UseVestaDir in the user's home directory

9.13 Startup

One creates a ManageReps tool instance via "ManageReps [-b]" in a shell.

There should be a Vesta server running on the workstation prior to starting ManageReps. (Start a server via "startvesta" and wait for the log-file window or icon to appear.)

The -b flag specifies that some displays (particularly lists of families and packages) should use a FormsVBT "Browser" sub-window. Without the flag, a formatted-by-ManageReps TextArea is used.

The inital repository interrogated by ManageReps is the one specified by the environment variable VPUBLIC, if present, or else "/vesta/proj".

10. Omissions and Known Deficiencies

The Vesta world offers only basic facilities right now; lots of functionality needs to be added. There are also quite a few things that don't work the way you'd like.

The descriptions below contain many references to postings on src.vesta. The references are to the most informative posting, not necessarily the earliest posting on a given topic. Use Postcard's "Browse Discussion" capability to explore the entire history of a topic.

10.1 Language

- Caching interacts with non-functional functions. Copying files to release directories is tricky to do in a functional language. You call the **Copy** function for its side effect on the file system, but Vesta does not know this. If Vesta finds that you've done the exact same **Copy** before, it won't repeat it.

  One work-around, which depends upon a quirk in the current implementation of Vesta caching, is to comment out the **Copy** call and evaluate the model. Then uncomment the **Copy** call and evaluate again. This should trick Vesta into performing the **Copy**.

  We could change the language to eliminate this sort of problem. We might allow you to declare a function "nonfunctional" so that it would be reevaluated every time the model is evaluated. (hanna, src.vesta, 24 April 91.)

- One error leads to many error messages. When a compilation fails, Vesta tells you about it, then proceeds to tell you that later functions that depended upon the results of the compilation have failed, too. Why doesn't Vesta give up sooner and spare you all of the redundant error messages? According to the Vesta Language definition, the result of evaluating the Vesta program

```
LET
  f ~ LAMBDA x, y IN x
IN
  f(1, error("oops"))
```

is 1, not **ERROR**, even though both arguments are evaluated before the call to f. Maybe these semantics should be changed, and **ERROR** made contagious. In that case, evaluation could stop along any branch that returned **ERROR**, although other branches could continue to evaluate. For example, in evaluating

```
f1(error(), f2(), f3())
```

the calls to `f2` and `f3` might still be evaluated, but `f1` would not.

Or maybe, instead of changing the language semantics, we would just suppress "superfluous" error messages caused by **ERROR** values, if we could decide which those were. (hanna, src.vesta, 4 January 91.)

A related language bug: The result of evaluating

```
{ a ~ 3, error("hello") }
```

is an error message plus the binding

```
{ a ~ 3 }
```

It would be better if the result was

```
{ a ~ 3, anon ~ ERROR }
```

or even simply

```
ERROR
```

- Use the binding-constructor comma with care. Ideally, you would use semicolons only where necessary and use commas everywhere else. This would give a model that's as fast as possible to evaluate.

The problem is, when you make a mistake and use a comma where a semicolon was needed, the error message you get may be quite obscure. Here are some guidelines to help you strike a balance.

A common error is to use a comma following your top-level **intfs** function. (mann, src.vesta, 18 Feb 91.) Unless your model is very unusual, you'll lose only a tiny bit of performance by using semicolons after all of the top-level elements in your model.

Another common error is to use a comma after a **Compile(Foo.def)** that is followed by a **Compile(Baz.def)** where **Baz** imports **Foo**. UseVesta will figure out a compilation order and put in the semicolons for you; it requires that your model include stylized comments so it can tell what lines to replace.

Using commas in your **impls** function, between the compiles of implementation modules, can't cause a problem. When Vesta has a parallel evaluator, these commas should make your model evaluate much faster. Be sure to use these commas.

- Syntax for function application is too permissive. The lack of a comma or semicolon following **Compile(A.def)** in

```
LET M2 IN {
    Compile(A.def)
    Compile(B.def)
    };
```

causes a confusing error message, because **E1 E2** is legal syntax for a function application. This is a typical problem with expression languages. In a statically-typed language this problem is usually caught at compile time, but in Vesta it is only caught at function application time.

It would be possible to require that arguments to procedure calls be lexically enclosed in parens or in braces. With this more restrictive syntax, Vesta would complain sooner in the example above, and would produce a more understandable message. A Vesta model pretty printer would also help detect this error. (jdd, src.vesta, 21 May 91.)

- Binding constructors don't complain about duplicates. Both comma and semicolon use merge rather than union to build the result binding, so no error is raised if the left and right arguments contain

binding elements with the same name. This makes it harder to diagnose duplicate-name problems in large "umbrella" models like buildingenv and release. Paul McJones thinks that changing from "merge" to "union" semantics would not require changing very many models. (mbrown, src.vesta, 12 June 91.)

10.2 Evaluator

- No automatic reconstruction of deriveds. Originally, Vesta was designed to perform automatic reconstruction of deriveds in case weeding was too aggressive. This process is called recipe evaluation. Vesta's recipe evaluation capability is now disabled, because it was not fast enough to be useful in reconstructing more than a few deriveds, and usually it needs to reconstruct many deriveds if it needs to reconstruct any.

- Vesta's pretty printing of opaque values is poor. At present you often see evaluator messages like

```
-> Vulcan.Prog: bad argument: opaque
```

  when it would more informative to see something like

```
-> Vulcan.Prog: invalid argument: <Vulcan interface: Ts>
```

  (ellis, src.vesta, 10 May 91.)

- Evaluator starts too many bridges. At the start of every evaluation, the evaluator tries to ensure that all bridges it has ever used are currently running. This can lead to obscure error messages if some now-irrelevant bridge can't be started (e.g. because it has been weeded.) There is no inherent reason why the evaluator should start bridges as it does; it was implemented this way for simplicity (this implementation avoids attempting to restart a buggy bridge more than once per evaluation, which otherwise might be difficult to arrange.) (hanna, src.vesta, 21 May 91.)

- Evaluator dumps core if stack overflows. The Vesta evaluator does not perform any checks to see if the evaluation stack is getting too

deep. So if you write an infinitely (or deeply) recursive model, the evaluation stack will exceed implementation limits and the Vesta evaluator will dump core. The most common cause of a deeply recursive evaluation is an attempt to use an old buildingenv that has been weeded; the evaluator now prints a message in this case. (hanna, src.vesta, 6 March 91.)

- No parallel or distributed evaluation. Vesta will eventually evaluate models using multiple threads and multiple machines to process bridge calls in parallel; for now, it performs one bridge call at a time, on the local machine.
- No "vsync". There is no good way for a program to tell when the evaluator has finished writing out the cache or caches it is pickling. A vsync command would solve this problem. The evaluator does write a message in the Vesta log when it has finished writing a cache, so a user can tell.

10.3 Bridges

- Vesta can wedge while aborting an evaluation. This is a low-probability event. It is most likely to occur with the MM bridge, but can occur in certain uses of the Shell bridge as well. After you click Stop (UseVesta) or type control-C (vmake), the evaluation ends up stuck, with nothing appearing to happen. Other processes, unrelated to Vesta, can get stuck as they try to exit. Things grind to a semi-halt with various threads waiting in RPC.Moribund. You should reboot. (levin, src.vesta, 12 May 91.)
- Bug in shell bridge treatment of stdin. If your model says

```
Shell$Sh("foo")
```

the effect is

```
Shell$Sh("foo < '' ")
```

i.e. no stdin is the same as stdin containing the empty string. It would be best to treat reading from an otherwise undefined stdin as an error, but we don't know how to implement this in the shell bridge without modifying the shell (redell, src.vesta, 5 March 91.)

- No csh in shell bridge. Most SRC users are more familiar with csh than sh. However, for technical reasons the shell bridge does not work with csh. Lucille Glassman is facile with sh; maybe you can get her to help you.

- Creating relative links in release directories. If you try to create a relative symbolic link using ln -s under the shell bridge, you'll get an absolute link instead. You'd like to be able to create the relative link, at least under the unsafe shell bridge, but it is not clear how to make it happen. (mann, src.vesta, 3 April 91.)

- Putting bridge working directories in /tmp has problems. Some Vesta bridges create working directories with long hexadecimal names (like 08002b13f90bv27c45adb) in /tmp. The daemon that cleans up /tmp will delete such a directory, but the bridges are careful to keep the daemon from doing this while the bridge that's using the directory is alive. If you feel compelled to clean up /tmp manually, you should kill your Vesta server first; otherwise you may get strange Vesta behavior (huge numbers of messages complaining of being unable to create a temporary directory.) (levin, src.vesta, 6 Mar 91.)

  It appears that the right thing for Vesta to do eventually is to follow the conventions for servers established by smgr, and keep its private files in /machine/srcfNN/admin/servers/vesta. (wobber, src.vesta, 6 Mar 91.)

- Unshipped mm executables are hard to debug. Suppose you are using the mm compiler, and your model includes a test component; evaluating the test component compiles a test program and runs it. If the test program dumps core, how do you find the core file and debug it? The core file is stashed away in a subdirectory of /tmp. Assuming that you can find the core file, use cfa(1) to find the vesta uid of the executable from the core file. Append this uid to"DebugCore " and it should work. (detlefs, src.vesta, 11 March 91.)

10.4 Disk Storage Management

- Advance without Eval makes the default vrweed command delete the most recent cache. An Advance operation makes a new leaf node in your private repository. If you do several advances, without

evaluations, you will build a branch in which the most recent vesta server cache is several nodes away from a leaf. Then, a the default vrweed command is likely to delete that most recent cache, since it hasn't been associated with any leaf or near-leaf: only an evaluation associates cache info with a package. (ayers, src.vesta, 31 December 90.)

- No automatic clean-up of private repositories. Someday we hope to have a standard daemon (say, driven by cron) that gets rid of old stuff in your private repositories. But because of the problem just cited, it is too risky to perform weeding of private repositories via a standard daemon. If you fail to run vrweed, you may consume huge amounts of disk space. So run vrweed regularly.

- Do not weed and evaluate concurrently. During an evaluation, Vulcan may lookup a derived object and discover that it exists. Then vrweed may delete it. Then Vulcan reads it and gets an error. Until this problem is fixed, you should not weed and evaluate concurrently. (hanna, src.vesta, 12 June 91.)

- Very long checkout sessions interact with public repository weeding. For technical reasons, very long checkout sessions may retain many obsolete derived objects in the public repository. Until this is fixed, try to do a checkin every month or so. (levin, src.vesta, 8 May 91.)

- Disconnected sessions interact with public repository weeding. If you perform buildWithOverrides (or do some comparable thing with another public repository umbrella model) from a disconnected session, and want your deriveds to be preserved in a public repository weeding, you must tell the weeding czar. If you forget to do so, and your deriveds are removed by vrweed, you must comment out your test component, evaluate, then uncomment and evaluate. (chiu, src.vesta, 10 June 91.)

- "du." There isn't any real analog of the Topaz **du** command for repositories. **vrweed** can give you a summary number, e.g.,

```
vrweed /user/\$USER/vesta/private -k 0 -n
```

to get the total of all deriveds, or

```
vrweed /user/\$USER/vesta/private -d -mu -k 0 -n
```

to get the total of all deriveds and sources, but it's a little slow and clunky.

- Do not Enable in role vestasrv to clean up a repository. Because of limitations in the implementation of our authentication machinery, everybody at SRC has the right to run as vestasrv. But please refrain from doing so. No one other than a Vesta administrator has the need to run as vestasrv.

  To remove a repository, use the appropriate command in "ManageReps". To remove an object from a repository, use "vrweed". (chiu, src.vesta, 16 January 91.)

- Weeding sources has problems. Today, vrweed complains and stops if it can't find an imported model when tracing the sources reachable from a package version. So if you, for instance, create private versions of buildingenv, it is probably unwise to weed the sources until this vrweed bug is fixed. (chiu, src.vesta, 31 May 91.)

- vrweed message "Non-UID file found while enumerating objects". During a vrweed, you may see a message "Non-UID file found while enumerating objects" in the Vesta log. This message means that your repository contains a form of garbage that vrweed does not yet know how to collect. This garbage is not harmful, apart from the storage it consumes; at some point vrweed will be extended to collect it. If you find that such files are consuming a lot of your space, ask for help from Roy Levin or Sheng-Yang Chiu. (chiu, src.vesta, 22 April 91; levin, src.vesta, 5 August 91)

- Weeding versus aborted evaluation. If you weed your private repository immediately following an aborted evaluation, the weed may do the wrong thing. Weeding with `-k SESSIONS 2` will reduce the likelihood of this problem striking you. (levin, src.vesta, 6 November 91.)

10.5 Repository and Clerk

- One Vesta server per machine. Only one Vesta server may be run on a machine at a time. If you try to start up a second server when the first is running the second server exits with a nice error message.

- All Vesta files are executable. The Vesta clerk regards all files as executable, but few of them are. This can be confusing. (levin, src.vesta, 16 April 91.)

- Obsolete packages live forever. There is no way to delete erroneous or obsolete package names short of deleting the entire repository.

- Disconnected sessions interact with compression. You cannot compress a file that was created in a disconnected session with another that was created in a checkout session, and you get an obscure error message in the attempt. (chiu, src.vesta, 28 March 91.)

- Disconnected sessions interact with checkin. If you do a fetch-and-advance from a disconnected session into a checkout session, and then try to do a checkin, VestaRep complains that you are trying to checkin a UID that wasn't authorized by a public repository. You'll have to touch the offending file so that UseVesta will assign it a new UID.

  It would be better if the user interface issued a warning when you tried to do a fetch-and-advance from a disconnected session into a checkout session. It would also be better if the checkin-time error message avoided obscure terminology such as "session handle". (ellis, src.vesta, 20 August 91.)

- UIDs aren't first-class pathnames. You can pass Vesta UID to Taos as a pathname, and Taos will pass it through to the clerk, who will open the Vesta object as a file. But if you apply Filename.Head to a Vesta UID and pass the result to the clerk, it will deny that it is a directory. (ayers, src.vesta, 13 July 91.)

10.6 Miscellaneous

- Primitive browsing. Browsing facilities are primitive. But improving. See vxref and ExamineEnv.

- Commenting out test components of buildingenv packages. Some people like to keep their test evaluations from one checkin to the next. Unfortunately, test components cause the Vesta server to become huge when evaluating a new buildingenv or release model, because the test components generate big caches. So it is considered polite to comment out your test component when you check in a

library package. When you do this, you give up the cache the next time you check the package out. (hanna, src.vesta, 10 June 91.)

Upcoming performance improvements will soon make it unnecessary to comment out test components.

- Preprocessor output is hard to debug. Debugging preprocessor output can be difficult in Vesta, because the preprocessor output is a derived and does not have a good user-sensible name. Various workarounds are available today. If the preprocessor output you want to see is a Modula-2+ module in buildingenv or the release model, vxref will help you. For instance, the sxrpc package flumes the SxRPCInternal interface; here's how to find one of the results of this fluming:

```
ff> vxref SxRPCInternalClient definition
SxRPCInternalClient definition /v/0haM08002P.Zay/D8ec.vYEWL59KU-7zKbgz.sxrpc
```

You can use the ugly Vesta pathname to visit the file with your favorite editor. The vxref implementation relation is similar, for finding implementation modules:

```
ff> vxref SxRPCInternalServer implementation
SxRPCInternalServer implementation /v/0haM08002P.Zay/D8ec.Ey-g5PhrwQvz7xpd.sxrpc
```

If this avenue is not available to you, you can write a tiny model that performs the preprocessing and then copies the result into the Unix file system. Or, with Vulcan, you can use the debugger's VESTAUID builtin to find the source code of any module in a program. (sclafani, src.vesta, 1 May 91.)

- No Vesta model pretty printer. A Vesta model pretty printer has been on the low priority list for some time. There's nothing difficult here except for comments, which are always exciting. It took a few hours to do the pretty printer for the Vesta language without comments; after a week of trying to get add pretty printing for comments Chris put the pretty printer on the back-burner where it has stayed ever since. (hanna, src.vesta, 21 May 91.)

- Vesta tools dump core if there's no Vesta server. Various tools that use Vesta (Vulcan, UseVesta, vmake, ...) have a tendency to dump core if Vesta crashes or is not running when a tool starts. So, for

instance, if you run UseVesta just after running startvesta, chances are that UseVesta will crash. The Vesta RPC client stubs need to be fixed to provide a more civilized error indication.

- Test-shipping to /tmp has problems. It is common practice in Vesta models for the test component to ship deriveds to subdirectories of /tmp. Echo defines a different /tmp for each machine as a performance hack to avoid having one directory that is heavily write-shared by all machines. It does *not* do this to give every user his own place to write temporary files with assurance that there will be no name conflicts. If you run a Vesta evaluation remotely using dp or the like, you will install files in some other machine's /tmp directory, where you may have trouble finding them and where their names may clash with files created by other users of the same remote machine.

  A different convention would be to create a directory /vesta/out for each user, and perform test ships there. This would require a Vesta evaluation to have access to the name of the user performing an evaluation. At present the server can look at the USER environment variable to find who started it up, but in principle a server can be shared (as it might well be in the dp example.) (mann, src.vesta, 24 January 91.)

- There is no standard replacement for CTime. The Vulcan version of CTime returns the Vesta UID of the program as the version string. (mcjones, src.vesta, 4 January 91.) Tim Mann has invented a CTime replacement for Taos that gives a more meaningful version string, but this CTime has not been institutionalized. If you are interested in Taos CTime, read the taos, bootfile, and vrnpn models.

- Execute permissions not maintained. Every object in the repository has execute permission turned on. But when the VestaUserLevel copies objects into the working directory, it turns the execute permission off. This is an annoyance when a model contains shell scripts. (Ellis, 12 July 91.)

- ReviseImports is not smart enough. Occasionally a buildingenv is checked in before it is ready for widespread use. This creates problems because ReviseImports has no way to know that it shouldn't use the latest. One way to avoid trouble is to import the release model instead of buildingenv, and use the buildingenv from the most recent release. But this can fail, too, e.g. when Vesta's cache format changes between releases. (levin, 1 October 91.)

Index