107

The Vesta Language for Configuration Management

Christine B. Hanna
Roy Levin

June 14, 1993

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth. This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Robert W. Taylor, Director

The Vesta Language for Configuration Management

Christine B. Hanna
Roy Levin

June 14, 1993

Authors' Abstract

Current approaches to software configuration management and system building do not scale up to support large-scale software engineering; common practice involves numerous stopgap measures to work around this shortcoming. The Vesta system is designed to eliminate this problem, by providing (1) a language designed to support complete, concise system descriptions and (2) a novel caching mechanism that permits efficient system building.

The Vesta system uses a functional programming language to describe configurations. This language provides the flexibility and power needed to describe large software components. The system descriptions are specific and complete, and include all of the sources that are used to build the system and all of the instructions that tell how the sources are composed. Only information written down in the description can influence construction of the system. Nevertheless, the descriptions are concise and easy to read and write.

The language evaluator caches the results of evaluating function applications, which are the expensive operations in the Vesta language. Caching in Vesta is automatic and persistent. Because the language is functional and there are no side-effects, caching is conceptually straightforward. Vesta caches the result of all function applications—from those at the leaves (e.g., compiling one source file), to those in the middle (e.g., packaging up a library), all the way to the top. Caching function applications at all levels permits Vesta to build and rebuild large software systems efficiently.

Contents

1 Introduction

The development of large-scale software systems requires effective configuration management—the organization of software elements into systems. Configuration management must cope with a wide variety of development difficulties, including parallel development and testing, portability and cross-platform development, and inter-component consistency control. For small systems, these issues can often be addressed manually; for large systems, automation is essential.

Vesta is a system that provides the automation to solve these problems. Central to the solution is a comprehensive mechanism for describing and constructing a large software system, which is the subject of this paper.

More specifically, the Vesta system is organized as follows:

- The *repository* [Chiu and Levin] names and stores objects.
- The Vesta *language* is used to write *system models* (or *models*, for short) that describe how the components fit together to form a resulting software system.
- The language *evaluator* processes the models to build software systems.
- *Bridges* [Brown and Ellis] provide the connection between the Vesta evaluator and program development *tools*. Vesta is extensible and bridges are the way it is integrated with existing preprocessors, compilers, linkers, etc.

Figure 1 presents a simple example that illustrates the user's view of these pieces. The user provides Vesta with the files `A.c`, `B.c`, and the system model `Model`, and gets back the executable `AB`. The system model describes how `A.c` and `B.c` are compiled and linked to produce `AB`. Figure 2 is an elaboration of Figure 1; it shows how the work flows internally through the components of the Vesta system.

As the example suggests, a Vesta model describes how the components of a software system fit together. Models correspond roughly to the `makefiles` of Unix—both describe how to build a software object from its components. But Vesta models do so far more precisely and comprehensively than `makefiles` do [Levin and Mcjones].

Unlike `make` (and most existing configuration management systems), Vesta uses a real programming language to describe configurations. Other systems, including `make`, use an ad-hoc approach with many system-building concepts hard-wired into the description language. While some of these system description languages have simple control structures, they have nothing like the abstraction facilities provided by functions. Vesta's programming language provides the flexibility and power needed to scale up to describe large-scale software development.

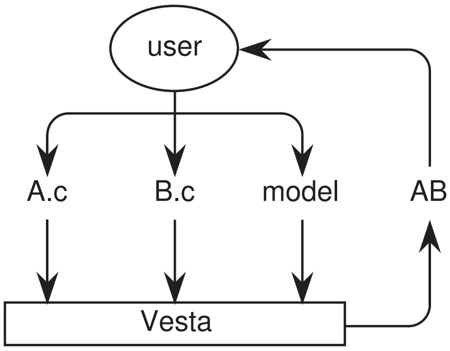

Figure 1: User interaction with Vesta

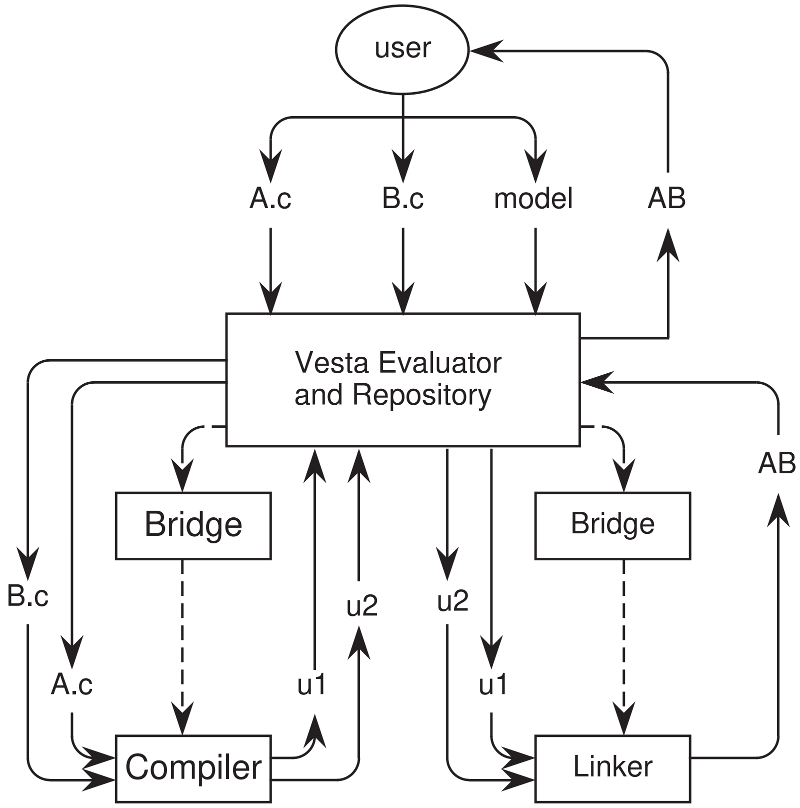

Figure 2: User interaction and workflow through Vesta

The Vesta language is functional; it is essentially typed lambda calculus with a small number of additional features. An operation like "compile" is represented in the language as a function; the function inputs are the source files and the compilation options, and the result is the object file. A software system is built by a series of function applications. Part I of this paper explains the language in detail and shows how these system models are constructed.

Using a functional language simplifies the language evaluator. A software system is compiled and linked whenever a model is evaluated. To do this fast enough, the evaluator must cache the results of expensive sub-evaluations so that next time the results may be reused. It would be quite complicated to cache expression results if the modelling language permitted side-effects, but in a functional language caching is conceptually straightforward to implement. Nevertheless, an implementation that actually works for a large system is quite challenging to construct, which is the subject of Part II of this paper.

Because Vesta underwent an extended trial managing large software systems [Levin and Mcjones], there is considerable experience that supports the workability of this approach. Part III of the paper evaluates that experience as it applies to the language design and implementation, and identifies opportunities for further enhancements.

Part I

Vesta models

2 Organizing software using models

This section presents Vesta's approach to organizing software. It provides background for the rest of the paper. This background is particularly relevant to the discussion of the language, because the language was designed to support the approach described in this section.

2.1 Model properties

All models share two fundamental properties:

- a model logically contains all the "text" of a system
- a model uses functions to describe software construction

The next two sections discuss these properties in detail.

2.1.1 Models logically contain all the system text

A model logically contains all the text used to build a software system. This means that it contains all the program code that goes into the system, as well as all the building instructions that tell how to put the code together. Every element that was used in producing the system is explicitly specified in the model, including the interface modules and/or header files, the software libraries, versions of the compilers, compiler options, and hardware platform. Only information written in the model can influence building. Vesta controls the environment in which compilation and linking occurs so that implicit dependencies not included in the model, such as Unix environment variables, cannot influence building.

Because all the information used to build the system is written in a model, the model gives an exhaustive record of how the system was constructed. This means that exactly the same system may be rebuilt whenever necessary.

A model for a C program logically contains every line of C code that goes into the program, including all the source code in the C libraries. Actually including all this code in the model, however, would make the model difficult to manage and manipulate. Instead, this code is almost always stored in separate *source* objects. A source object is something that is hand-crafted and cannot be mechanically reproduced. Sources are named in the model by a unique repository identifier or UID. To maintain strict control over each source that logically appears in the model, source files are *immutable* and *immortal*; the same UID always refers to the same contents.

System models also logically contain all the building instructions for a given system, including how to build all the tools and libraries used by the application. Of course, it would be impractical for every model to include the construction instructions for the entire environment. Instead, models are modular, and the instructions for constructing different parts of the system are written in different models. Modularity allows a software system to be sensibly divided into subsystems by providing a way to compose the subsystems into a larger system. Models are (almost always) source objects, and one model can reference another.

2.1.2 Functions describe software construction

Models are written to help localize the binding of choices of how a system is built. When a choice is localized, changing it means making a change in a single place. To achieve this, models use highly parameterized functions in which most of the choices are passed as parameters to the functions.

For example, by calling a function that describes an application program with different parameters, one can build the program for a variety of hardware platforms, or with various versions of the compiler and libraries. Moreover, models are generally organized so that one *umbrella* model calls functions in many others, passing the same parameter values to each. This makes it easy to change the values used in all the models by changing only the umbrella; one can build a new system release for a different platform by changing just one line in the umbrella release model. Figure 3 shows the structure of the SRC release umbrella model and some of the models that it invokes; note that the bootfile model and the Vesta model are themselves also umbrellas.

The localization of choices means that the amount of work that one person must do to modify a system is proportional to the size of the change, not proportional to the size of the system. Building a new system release for a different platform is a conceptually simple change that does not require more human effort than changing the appropriate line in the system description. Of course, a substantial amount of machine time may be needed to complete this build.

2.2 Model structure

A model has two sections:

- the *sources* section names the source files that compose the system,
- the *construction instructions* tell how to process the source files to build the system.

Figure 4 shows an outline of the sections of a model.

The sources section has two parts. The first names the source files that are local to the subsystem described by the model. The second (followed by the

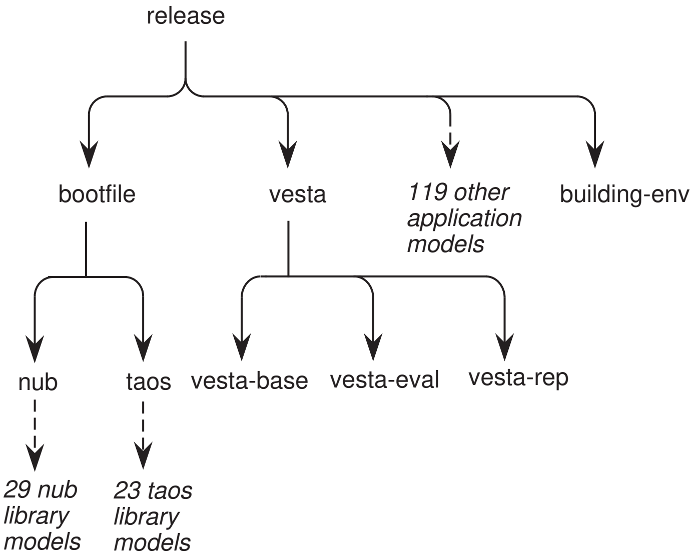

Figure 3: SRC release model

keyword `IMPORT`) names the models that come from the surrounding environment.

The construction instructions section also has two parts. The first, the functions, provides the highly parameterized functions that describe how the sources are composed during building. The second, the testing expression, contains applications of these functions to actually build a system. This is called a "testing expression" because it builds a component for local testing.

Figure 5 shows a simple Vesta model that builds a "hello" program. The details of the model aren't important here; they are discussed in the next chapter. The model is here simply to show the basic model structure.

The sources section in Figure 5 contains two source files. `hello.c` is the source file local to the model; `building-env.v` is an imported model that provides the environment necessary to build the program. Both sources are named by their UIDs, which are abbreviated here with as *uid*-n.

The construction instructions contains two components. The component labeled `build` is one function that defines the compilation and linking of the "hello" program. The second component labeled `test` applies the `build` function, as well as a function from `building-env.v` called `default`, which supplies the default environment in which to build the program.

The `test` expression is used by the developer of this model to test his program using a given version of the environment. This program may also be built

```
DIRECTORY
  ....
  ....
IMPORT
  ....
  ....
IN
```

for a system release. In this case, the `build` function will be called from the release umbrella model, perhaps using a different environment. This is why the model separates the `build` function from the `test` expression; the `build` function may be called from multiple different contexts. A function like `build`, which may be called from outside the model, is written to assume that a building environment has already been established before it is applied.

2.3 Updating a model

Since models and other programming modules are usually source objects, they are immutable. To make a "change" to a source object, a new version of the object must be created. For example, say that a user has the following model, called `lib.v.3`[1]:

```
DIRECTORY
  A.c = uid-1,
  A.h = uid-2,
  B.c = uid-3,
  B.h = uid-4,
IN
  ...
  C$Compile(A.c, A.h),
  C$Compile(B.c, B.h),
  ...
```

If the user wants to "change" A.h, he must produce a new version of A.h with a new UID, and use that new UID in a new version of the model, called `lib.v.4`:

```
DIRECTORY
  A.c = uid-1,
  A.h = uid-5,
  B.c = uid-3,
  B.h = uid-4,
IN
  ...
  C$Compile(A.c, A.h),
  C$Compile(B.c, B.h),
  ...
```

Because a user must explicitly indicate that the original version of a source is to be updated to a newer version, he has complete control over incorporating

---

[1] This is a slight simplification of the naming scheme used for sources in the Vesta system.

other's changes into his own model. For example, the model in Figure 5 imports a particular version of the `building-env.v` model. One user may modify the environment and release a new version of the model, but a second user working on the "hello" model is completely insulated from these changes. The old versions continue to exist, and the second user may continue to import the old versions until he is ready to change to the newer ones. In this way, users are protected from arbitrary changes in their environment. Tools are available in Vesta to help users create updated versions of existing models, both for source objects local to their model and for imported models.

2.4 Evaluating a model

Most configuration management systems do not incorporate the concept of a specific system configuration. Instead, they use descriptions that refer to mutable files whose contents may change. Their building tools must determine whether any of the mutable files named in the description changed, and then what to do about it. With `make`, for example, the rule is that if file Y depends upon file X, changing X requires Y to be recomputed. If a `makefile` records that the file `A.c` depends upon a header file `A.h`, then `A.c` must be recompiled when `A.h` changes.

Vesta cannot use such an approach to rebuilding. Vesta lacks the very idea of a "change" since Vesta's models are immutable. As discussed in Section 2.3, if a new version of `A.h` is produced, then a new version of the model, `lib.v.4`, may be created to incorporate the new file. Source objects in a model don't change because a model describes a particular immutable state.

Objects in Vesta don't change, but they do stay the same, and the Vesta evaluator takes advantage of this. In particular, the evaluator notices when two functions applications are the same, even in different models; this occurs when the function and the values of its parameters are identical. For example, consider the models `lib.v.3` and `lib.v.4` from Section 2.3. The application

```
C$Compile(B.c, B.h)
```

is the same in both models because the function and parameters are identical. If the result of this function was computed when evaluating `lib.v.3`, the same result can be reused during the evaluation of `lib.v.4`. Reusing the result of a function already evaluated is the job of the *caching* machinery, discussed in detail in Part II.

On the other hand, in `lib.v.4`, the function application

```
C$Compile(A.c, A.h)
```

is different from the similar one in `lib.v.3`, since the parameter `A.h` has a different value between the two applications. When the evaluator first analyzes `lib.v.4` it decides that this result doesn't yet exist, so the function application is performed. In general, a function application is performed when the result of that application doesn't already exist in the cache.

2.5 Constructing the environment

A model's construction instructions describe the entire software system. Evaluating those instructions builds the entire software system. Each time the system is built the entire model is evaluated and whatever must be built, is built.

In the "hello" model in Figure 5, the instructions for constructing the environment are executed when the `building-env.v$default` function is applied. Every time the "hello" model is evaluated the entire environment is built. This means that instead of evaluating a 4–line program (`hello.c`), the evaluator analyzes a system with many thousands of lines (the code for the libraries and the compiler and linker, and for `hello.c`). This may seem unnecessarily aggressive, but caching makes the parts of this that have been done before very fast.

Since the environment is built upon every application, a consistent build is always produced. A user may write a new model specifying a new component of a library, even the lowest level interface, and know that every function application with that component as a parameter will be evaluated and a consistent environment will result. Also, a tool maintainer can build a new version of a tool, and incorporate it in the environment, knowing that all necessary processing with that new version will be performed. (There is a separate way for a tool maintainer to assert that one version of a tool is upward compatible with an older version, so that all results produced using the old tool may be reused; details are given in the paper on Vesta bridges [Brown and Ellis].)

2.6 Repository functionality

The Vesta repository provides management and naming of the different versions of objects; details are given in the Vesta repository paper [Chiu and Levin]. There are two kinds of objects, *source* and *derived*. Source objects were introduced in Section 2.1.1; they are hand-written objects. Derived objects are those mechanically produced, from sources or other deriveds. Like source objects, derived objects are named by UIDs.

There are four repository functions that are needed for the rest of this paper:

- `NameSource() -> UID`: This function produces a new source UID. The only requirement for a source UID is that it be unique. UIDs produced by `NameSource` are distinct from those produced by `NameDerived`.
- `NameDerived(inp:  LONGINT) -> UID`: This function produces a new derived UID. The derived UID is completely determined by the value of `inp`.

If `NameDerived` is called twice with the same value for `inp`, it will produce the same UID. UIDs produced by `NameSource` are distinct from those produced by `NameDerived`.

- `Write(uid: UID, t: Text)`: This function takes a UID and a Vesta text value and associates the UID with the text.
- `Read(uid: UID) -> Text`: This function takes a UID and returns the Vesta text value associated with that UID.

3 The Vesta language

The language is typed lambda calculus with a few additional features needed for configuration management. It does not include many traditional features of general-purpose functional languages, such as explicit support for recursion and data abstraction. This section gives a brief overview of the features that the language does contain.

The language is a purely functional, modular language. It is composed of expressions which when evaluated produce values.

The type system is simple, with a few basic types: boolean, integer, text, function, opaque, binding, and list. Lists are like LISP lists; bindings are sets of name-value pairs.

Opaque types are associated with language *bridges*, which provide a way to invoke compilers and other tools from a model, and which can extend the language's type system. A bridge is the intermediary between the evaluator and an individual compiler or tool. Bridges produce values with opaque types; only the bridge that produces a value of a specific opaque type can manipulate it.

The language is dynamically typed; that is, types are associated with values instead of with names and expressions. Type-checking occurs as the model is evaluated: that is, as the software system is being compiled and linked.

Even without static type checking, the language is strongly typed: an executing model cannot breach the language's type system. The expected types of parameters to language primitives are defined in the language documentation, and actual parameters to primitive functions are checked when the model is evaluated. Bridges cooperate in the implementation of strong typing by type-checking the parameters to the functions that they export.

The basic scoping mechanism is static (or lexical). The language also has dynamic scoping for free variables in appropriately-defined functions. Dynamically scoped names are not lexically distinct from statically scoped ones.

There are a just a few kinds of expressions. Sets of name-value pairs (or bindings) are introduced into the environment via a `LET` expression. The value paired with a particular name is selected from a binding with a selection expression `binding$name`. Functions are defined with the `FUNCTION` expression and used in function application. There is a traditional `IF` conditional. The

`DIRECTORY` expression resembles `LET`, but introduces name-value bindings into the environment in which each value represents an object in the repository object store.

There are also about 50 built-in functions, which perform arithmetic and boolean operations and provide basic manipulation on texts, lists, and bindings.

The language supports higher-order functions: functions which return functions.

The language provides a way of defaulting actual parameters at function application. The language also permits defaulting of *formal* parameters at the point of function definition (which is simply another way of saying that it supports dynamic scoping).

The language discussion that follows emphasizes why the language design turned out as it did. It focuses on the configuration management demands that the language had to satisfy. Each subsection presents one demand and discusses why it is important. The *Language details* portion of a subsection describes the Vesta language features that must satisfy the demand. This approach does not present the language features in the most orderly fashion and it omits some of its fine points. The Appendix contains a semi-formal specification of the language semantics.

3.1 A model describes a specific, complete configuration

A model refers immutably to particular versions of source files and imported models. Once a model is created it never changes.

A model describes the construction of an entire software system, including the construction of the environment used by the system. To make it feasible to write and modify such descriptions, the Vesta language is modular: a system description can be broken into several lexically separate parts that build on each other. Modularity also supports abstraction; one programmer can build upon the work of another without needing to understand all of its details.

Language details: DIRECTORY and UID expressions

> In the language, each model is a module: a lexically separate piece of text, stored in the repository as a source object and named by a UID. The DIRECTORY expression is the only top-level expression of a model, and it may appear only as the top-level expression. Figure 4 in Section 2.2 shows the structure of the DIRECTORY expression.
>
> The sources section has two parts: one for objects local to the model's component and one for objects that come from its environment. These two parts are evaluated identically by the language interpreter, but other tools that process models may treat them differently.

> The only expressions that may appear in the sources section are sequences of explicit bindings separated by commas. The left-hand side of an explicit binding is an identifier and the right-hand side is a UID: a unique repository identifier.
>
> Each UID names a source file. Sources are either models or non-models; these two cases are evaluated differently. If the UID names a non-model, the UID is evaluated by reading the source using the repository's `Read` function (see Section 2.6); the Read returns a text value. If the UID names a model, the model is read, parsed, and evaluated. The result of evaluating a model UID is an arbitrary Vesta value.
>
> The UID may appear only in the sources section of a DIRECTORY expression, on the right-hand side of an explicit binding. (This restriction makes it easy for tools to find all UIDs in a model.) In the body of the model, the repository object is referenced by the name to which it's bound in the sources section.
>
> The result of evaluating the sources section is a Vesta binding, which becomes the initial environment for evaluating the construction section.

3.2 All dependencies are captured in the model

The language permits a precise description that captures all of the dependencies of the system-building operations. For each operation, the output is completely determined by the inputs of the operation; these inputs must all be explicit in the model. This approach is supported by a functional programming language, since the system-building operations are pure functions with no side-effects; the dependencies of the operation are the parameters to the functions.

Functions are at the core of the language; they provide a powerful abstraction mechanism in which a user who wants to invoke a function needs to understand only the function signature, not the details of the function body. There are several different kinds of functions that may be used in a model. *Bridge functions* supply the basic system-building operations like compilation and linking. *Primitive functions* permit the manipulation of texts, lists, and bindings. *User-written functions* let a user create his own abstractions by packaging together related calls on other functions.

Language details: FUNCTION expression and closure value

> A FUNCTION expression defines a function. The following model contains a single FUNCTION expression which processes a Unix man page:

```
DIRECTORY
  Printing.man = uid-1
IN {
  doc = FUNCTION ComputeManPage IN
    { Printing.1 = ComputeManPage(Printing.man) }
  }
```

> The result of evaluating a FUNCTION expression is a closure value, which contains the traditional pieces: the function's formal parameters, the function body, and the static environment needed to evaluate the function body.

3.3 Models provide a single, uniform naming mechanism

In traditional programming environments there are several different name spaces that a programmer must understand and manage. In a Unix environment the name spaces include file system path names, environment variables, linker symbols, target names in `makefiles`, etc. The rules for modifying these name spaces vary, and interaction between them can be confusing.

Vesta provides a single, uniform naming mechanism, so that a programmer needs to use only a single mechanism, with the familiar language features of name binding and scoping.

Bridges also benefit from Vesta's naming scheme. In most configuration management systems, a tool like a compiler or linker must understand how to access many different name spaces, and must make rules about how the name spaces interact. With Vesta, all the names that a tool references are interpreted in Vesta's name scope.

Models add names to the environment using bindings, and bindings are the basic structuring mechanism for the values in models. Many models chiefly manipulate bindings, since configuration management for a large system requires moving around a lot of names along with their values. Bindings provide convenient manipulation of large groups of names and their values.

Language details: binding expression and value

> A binding expression is a list of binding components enclosed within curly braces, { and }. A binding expression evaluates to a binding value, which is a set of name-value pairs.
>
> A binding component may be either an explicit binding or an arbitrary expression. The left-hand side of an explicit binding is a name, followed by an equals sign, and the right-hand side is an arbitrary expression. An explicit binding evaluates to a binding that contains

a single name-value pair. If instead the binding component is an arbitrary expression, the expression must evaluate to a binding value. Consider the following binding expression:

```
{ Log = M2$Prog((M2$Compile(Log.mod))),
  LogUtilities() }
```

The first binding component is an explicit binding. The second binding component is a function application, which must return a binding value.

If two binding components are separated by a comma, then each is evaluated in the initial environment of the expression to produce two binding values, `b1` and `b2`. The resulting binding value is produced by merging the two bindings, after removing all elements from `b1` for which an element with the same name exists in `b2`.

Semicolons are also permitted in bindings; the result of evaluating:

```
{x; y}
```

is equivalent to evaluating the expression

```
{x, LET x IN y}
```

So, for example, the result of evaluating the binding:

```
{ a = 3, b = 4; a = plus(a, b) }
```

is a binding value with two name-value pairs: `<a, 7>` and `<b, 4>`.

Bindings are "opened" by the syntactic construct LET. The LET expression adds new name-value pairs to the environment in which the LET body is evaluated. In the expression `LET E1 IN E2`, it's an error if `E1` doesn't evaluate to a binding.

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

3.4 Users can add new bridges as needed

Bridges are not written in the Vesta language, but they add functionality that is accessible from the language. Bridges extend the basic functionality of the Vesta evaluator by providing new functions.

Users can add bridges for new compilers or other language processers whenever such tools are needed. A bridge is loaded dynamically during an evaluation. When the bridge is loaded, it tells the Vesta evaluator the names and signatures of the functions it implements. When one of these functions is called during an evaluation, the evaluator invokes the bridge to execute the function.

The language supports bridges by providing an extensible type system. Bridges cooperate in the implementation of strong typing by type-checking the parameters to the functions that they export.

The language also supports versioning of bridges and tools. The types, functions, and instances of the types exported by a tool are versioned; details are given in the bridge paper [Brown and Ellis].

Language details: opaque types and the bridge primitive

> Each bridge has a single opaque type associated with it. A bridge is responsible for defining, manipulating, and type-checking values with its opaque type. Because the contents of an opaque value are up to the bridge, a bridge may actually define various opaque types for its internal use; the bridge then checks that the proper opaque types are passed to its functions.
>
> For example, consider the following (erroneous) bridge call:
>
> ```
> M2$Prog((M2$Compile(Log.def), M2$Compile(Log.mod)))
> ```
>
> The `M2$Prog` function is passed two files: a compiled interface module and an object module. The M2 bridge uses distinct opaque types for these two values. However, the `Prog` function accepts only object modules for linking. When presented with this parameter list, the bridge signals a type error.
>
> The `bridge` primitive function makes a bridge and its facilities available for use in a model. The bridge function returns a binding which defines the available facilities; typically, a bridge will export functions and default parameters which may be used (or overridden) when one of the functions is applied.
>
> Note that there are exactly two ways to define a function: one is to write `FUNCTION`, and the other is to use the `bridge` primitive.

3.5 Models are concise

A model provides a complete description of a software system, in which all the dependencies of any given operation are captured. There is a tension between providing a complete description while also making it easy to create new models and read or modify existing ones. A naive approach to writing a complete description would produce a large, unwieldy model that would be difficult to read or modify.

The problem of producing concise models is addressed by certain features of functions and function applications, highlighted below.

3.5.1 A typical function and its application

Here is a typical function and its application; it will be used throughout the rest of the section, although the signature of the function will change as different language features are discussed.

Consider the function `Compile`; its inputs, such as the source files and options, are parameters. One possible signature of the Modula-2+ `Compile` function is:

```
FUNCTION M2-Source, M2-Imports, M2-Options
```

where `M2-Source` is a text, `M2-Imports` is a list, and `M2-Options` is a binding. The Modula-2+ bridge type-checks these parameters when the `Compile` function is applied.

Consider the compilation of the Modula-2+ source file `Val.mod`. In Modula-2+, one module can use definitions from other modules via the `IMPORT` statement. (Don't confuse the Modula-2+ `IMPORT` statement with the Vesta `IMPORT`.) The Modula-2+ `IMPORT` statement inside Val.mod is:

```
IMPORT VestaUID, VestaValue, VestaETypes, Env, Expr,
  Globals, Err, Prim, Location, EvalFriends, EHandle, Cache,
  VestaLog, VestaAST, Bridge, Lex, TagV;
```

To compile this file, a Vesta invocation of `Compile` must list all the imported modules, in addition to the name of the source to be compiled and the name of the compilation options. The Modula-2+ bridge names the imported modules using the convention that a Vesta name is the Modula-2+ module name appended with the string ".d":

```
M2$Compile(
  Val.mod,
  ( VestaUID.d, VestaValue.d, VestaETypes.d, Env.d, Expr.d,
    Globals.d, Err.d, Prim.d, Location.d, EvalFriends.d,
```

```
        EHandle.d, Cache.d, VestaLog.d, VestaAST.d, Bridge.d,
        Lex.d, TagV.d ),
      M2-Options )
```

This invocation uses positional notation to match actuals parameters to formals parameters. Positional matching requires order in the actuals, and so the actuals are ordered in a list. Positional matching is useful when there are a small number of actuals, but it becomes clumsy and error-prone when the number of actuals grows. The next two sections discuss how to make invocations of `Compile` more compact.

3.5.2 Defaulting actual parameters

As shown above, there are three parameters to `M2$Compile`: the source to be compiled, the imports of the source, and the compilation options. For different invocations of `M2$Compile` the source and imports will be different, but the compilation options will usually stay the same. It would be useful to be able to specify the compilation options in one place in a model and then omit naming the compilation options on each call to `M2$Compile`. The language supports this with the defaulting of actual parameters by name.

If an actual parameter is not supplied for a formal at a function application, the Vesta evaluator will look in the current scope to see if there is a definition `with the same name` as the formal. With actual defaulting, the call to `M2$Compile` can be rewritten as:

```
    M2$Compile(
      Val.mod,
      ( VestaUID.d, VestaValue.d, VestaETypes.d, Env.d, Expr.d,
        Globals.d, Err.d, Prim.d, Location.d, EvalFriends.d,
        EHandle.d, Cache.d, VestaLog.d, VestaAST.d, Bridge.d,
        Lex.d, TagV.d ))
```

The function definition names three formal parameters, but this application supplies only the first two actuals. The supplied actuals are matched by position. The remaining formal, `M2-Options`, is looked up by name in the scope of the function application; it is an error if there is no definition. Thus, `M2-Options` can be bound once for a number of calls to `Compile`.

The invocation of `M2$Compile` has gotten a little shorter and easier to read and write, but the inclusion of the imports is still unwieldy, and the user is simply repeating information already contained in the Modula-2+ source module. Listing the imports again is error prone since a user may not keep the two lists synchronized. The next section discusses how to eliminate the need to list the imported interfaces.

3.5.3 Defaulting formal parameters

The language further simplifies function definition and application by supporting *implicit parameterization.* Implicit parameterization allows clients to omit formal parameters from the function definition and then omit the corresponding actual parameters from the function application. The function body therefore contains *free variables*, which are names referenced in the function that are not one of the formal parameters of the function and are not otherwise defined in the body of the function. A free variable is bound either from the static environment of the function or from the dynamic environment that invokes the function; the static environment takes precedence over the dynamic environment.

The signature for `M2$Compile` indicates the possibility of free variables as follows:

```
FUNCTION M2-Source, M2-Options ...
```

The token `...` is part of the language syntax, and the three dots appear literally in the function definition. The token `...` indicates that formal parameters are omitted from the function definition.

Since only two formal parameters appear explicitly in the function definition, at most two actual parameters can appear in the application. Assuming that the actual for `M2-Options` is defaulted as in the previous section, the application of `M2$Compile` reads:

```
M2$Compile(Val.mod)
```

This is a clear improvement on the bulky actual parameter list of the previous section. The function application now explicitly mentions only the interesting parameter; everything else is bound by other means.

In particular, the free variables in the function body are the imported interfaces of `Val.mod`. Because these implicit parameters are determined when the function body is applied, there is no need for the user to bundle up the imports ahead of time in a list, as in Sections 3.5.1 and 3.5.2. Each import is a separate implicit parameter to `M2$Compile`.

As described above, if a free variable is not statically bound, and its enclosing function signature contains `...`, the free variable is bound to a value in the environment of the function application. Such a free variable is looked up in this environment when the function body is executed. To get access to the function application environment while evaluating the function body, the language uses *dynamic scoping* to bind such a free variable occurrence.

Note that the implicit formal parameters may change from one invocation of the function to the next. The only implicit formals that are looked up are those used along the current execution path; there may be others on other code paths that are not referenced. Also, implicit formals may depend on another parameter; in the example above the implicit formals are the imports of the module `Val.mod`; if `M2$Compile` is invoked with a different source module its imports would probably be different.

In summary, omitting the formal and actual parameter list makes models quite a bit shorter and easier to read, and requires a user to write less redundant information.[2]

Language details: scoping and function application

> The basic scoping mechanism is static (or lexical); that is, free variables in a function get their values from the environment in which the function is defined. To implement static scoping, the Vesta evaluator maintains a static scope.
>
> Typically, in a statically scoped language, each use of an identifier is associated with the innermost lexically apparent definition of that identifier. Because the language has binding values, however, the place where a given identifier receives its value cannot be determined in general until the model is evaluated.
>
> If a function is defined with `...` in its signature, dynamic scoping as well as static scoping is used for the identifiers in its body. With dynamic scoping, free variables in a function get their values from the environment in which the function was called. To implement dynamic scoping, the Vesta evaluator also maintains a dynamic scope. When a free variable in a function defined with `...` is looked up in the environment, the static scope is searched first, then the dynamic scope.
>
> We considered having a syntactic means of determining whether a name was to be looked up in the static or dynamic scope. This is the approach that Common Lisp, for example, takes when mixing static and dynamic scoping. We decided that this approach would be too annoying to model-writers, because many names are bound in the static environment and many others in the dynamic environment. For example, a C module being compiled and linked usually comes from the static environment, but many of its include files and libraries come from the dynamic environment. Also consider the following model:

---

[2] Implicit formal parameters have two traits that make them unlike regular formal parameters: they cannot be identified until the function body is evaluated, and they may change for each application of a function. Because of this some readers may prefer the more precise explanation of looking up free variables in the dynamic scope. We think that the implicit formal parameter approach is more intuitive, however, and that is why we have presented it here.

```
DIRECTORY
  main.c = uid-1,
IMPORT
  building-env.v = uid-2,
IN {
  build =
    FUNCTION ... IN {
      main = C$Prog((C$Compile(main.c), libc.a)) };
  test =
     LET building-env.v$default() IN build()
}
```

The function application `building-env.v$default()` returns the library `libc.a`. In the `build` function, `main.c` comes from the static environment, and `libc.a` comes from the dynamic environment.

3.5.4 Bridges generate names

Names may be introduced into a scope in two ways:

- A user may write the name in the model
- The name may be returned as part of the result of a bridge function application.

This section discusses the second of these two. Consider the compilation of interface module `Expr.def`:

```
M2$Compile(Expr.def)
```

This bridge call returns a single-element binding, with the name `Expr.d` and the value of the compiled interface module. In doing this, the bridge is constructing a binding containing a name that does not appear in the Vesta model; it is manufacturing a new name derived from its input. Since a programmer who uses the bridge needs to understand what names it can generate, the rules appear in the bridge's documentation.

Because bridges can introduce names into a model, a user cannot systematically replace all occurrences of one name with another and still be certain that the program means the same. This is because all of the names don't appear textually in the model. To make a name substitution, the user must take into account the rules that the bridge uses to generate new names.

3.6 Intermediate results are conveniently packaged

Some language processors produce results that will subsequently be analyzed by other language processors. The results from the first processor need to be bundled together for convenient analysis by the second processor.

For example, a Remote Procedure Call (RPC) stub generator produces several files that need to be compiled, some for use by the client and some for use by the server. Depending upon which options are used, varying numbers of files are produced. The RPC stub generator could return a binding that contains each of the files, but this would be clumsy for the client to handle since the output files may vary in number. The most convenient way to process the output files would be for the RPC stub generator to return two functions, one that compiles the server files and one that compiles the client files.

Language details: closure value

> A closure value is the result of evaluating a function definition. A closure is a first-class value; it may be passed as a parameter and returned as a result. The language provides higher-order functions which can take functions as parameters and return functions as their result. The RPC generator can directly exploit this feature.

3.7 The type system supports program clarity

The type system of a programming language can be both a help and a hindrance to a programmer. A type system can make a program easier to understand, since a programmer must follow a certain disciplined style imposed by the type system. Also, type checking detects certain kinds of programming errors that could cause incorrect program execution. On the other hand, a type system may be so restrictive that the programmer must strain to express certain ideas in the language; in the worst case a type system can prevent a programmer from formulating some meaningful ideas.

As discussed in this section's introduction, the Vesta language is strongly and dynamically typed. During the design of the language we explored some language variants that incorporated static type-checking. We discovered that they were inherently much more complex; moreover, they still couldn't describe structures that frequently arise in real systems with existing programming languages.[3] More important, however, is that dynamic checking is early enough, from the user's perspective, for a language whose "runtime" actions are the construction of software; errors will be detected at roughly the same time as the compilers detect their own static type errors.

---

[3] The interested reader is invited to reflect on the interaction of static typing and the use of *include* files in C and related languages. What is the type of a C program module, parameterized by the include files? How does one arrive at the answer statically, i.e., before compiling?

The lack of static typing does introduce some difficulties. In particular, the place where a given identifier receives its value cannot be determined in general until the model is evaluated. For example, consider the following:

```
LET {
  M2-Optimize = FALSE,
  building-env.v$default() }
IN
  build()
```

By textual inspection of the model, it appears that the value of `M2-Optimize` will be `FALSE` when the function `build` is applied. But the function application `building-env.v$default()` may return another binding for `M2-Optimize` which will override the explicit one. If the language had static typing, the user could find the declared result type of `building-env.v$default()`, and he would therefore know whether the function returned a binding for `M2-Optimize`.

Part II
Implementation

4 The Vesta evaluator

The Vesta evaluator analyzes system descriptions, or models, written in the Vesta language. As these descriptions are evaluated the software system is built. This section gives a brief overview of the design of the evaluator.

The Vesta evaluator has very different performance requirements from an evaluator for a general-purpose functional language. In the Vesta language the expensive operations are applications of bridge functions such as compilation or linking. Other parts of a Vesta model are comparatively inexpensive. This means that an interpreter is fast enough to process models. An interpreter evaluates the expressions in the model as it evaluates the model; type-checking is done during model evaluation.

The interpreter uses lazy evaluation; that is, a value is computed, or *unlazied*, only when it is needed. Furthermore, a lazy value is unlazied only once, not every time that the value is needed. A *lazy value* is, in effect, a spontaneously created closure; that is, the value is an expression together with the current environment at the time the lazy value was created.

Lazy evaluation is used by the Vesta evaluator as a performance enhancer. It is useful in the implementation of the Vesta evaluator but is not a necessary component of the language definition. There are no language features that depend upon lazy evaluation.

Lazy evaluation improves performance when models are evaluated. It is common for a model to import a substantial name scope (i.e., a binding) from another model, such as a set of related interfaces and implementations. The importer then typically uses only a subset of the imported names, so generating values for all of them would be unnecessarily time-consuming (though semantically correct). Instead, the evaluator assigns lazy values to the elements of the binding, thereby deferring the computation of those values until individual names are referenced and eliminating the computation of values that are never referenced.

4.1 Caching the result of expression evaluation

The language describes software systems in terms of source objects only. In principle, the Vesta evaluator works by evaluating the entire tree of models imported by the model it is working on, rebuilding all their components from source. But obviously it would be far too time-consuming to do this on every evaluation. Building the compilers alone could take several hours and several more to build the libraries.

To speed up evaluation, the Vesta system caches the previous results of evaluating expensive expressions; the evaluator stores the expression and the result in a *cache entry.* Before an expensive expression is evaluated, the evaluator checks to see if a cache entry for that expression already exists; if so, the expression is not reevaluated. In Vesta models, function applications (that require bridge calls) are by far the most expensive expressions to evaluate, so only the results of function applications are cached.

Two function applications that use the same function and the same inputs produce the same result. Because the language is functional, caching function applications is straightforward. A function application can be characterized completely by the function and its explicit and implicit parameters. Because side-effects are not permitted in the language, the only output of a function is its result. If the language were non-functional, the caching mechanism would have to cope with other outputs in addition to the application result.

Cache entries are grouped into separate *caches.* Caches are saved in persistent storage (the Vesta repository) so that they are available across different instances of the Vesta evaluator. Caches are also automatically shared among users so that one user can benefit from the work that another user has performed.

Vesta caches the result of all function evaluations, from those at the leaves of the dynamic call graph (say, compiling one source file), to those in the middle (say, packaging up a library), all the way to the top. (A few primitive function applications are not cached since it is not time-consuming to recompute their results.) This multi-level caching scheme permits Vesta to build and rebuild large software systems efficiently since it maximizes the amount of work that does not need to be redone.

Vesta caching has a superficial resemblance to the *memoizing* technique for functional languages [Hughes] [Keller and Sleep], in that both store a function application and its result value in a table for later reuse. However, memoizing is manual, since a programmer must specify which functions are to be saved, and ephemeral, since memo tables don't last across different instances of the language evaluator. Vesta caching is automatic and persistent.

The next three sections discuss the implementation of persistent caching. Section 5 describes the basic format of a cache entry, which is independent of the values in the cache entry. Section 6 examines the problems that arise when particular values occur in cache entries, and presents solutions for these problems. Section 7 describes how separate cache entries are grouped together into caches and how cache lookup is performed during evaluation.

5 Cache entry structure

This section presents the basic structure of a cache entry and how a cache entry is produced during a function application. This basic structure is independent of the values that occur in the cache entry; the next section (Section 6) describes the special handling required by certain values in a cache entry.

A cache entry corresponds to a function application in the body of some model. There may be many cache entries corresponding to the same syntactic function application in a model, because a function can be applied in different environments.

A cache entry is the triple:

```
<primary key, secondary key, result value>
```

A *cache hit* occurs when the evaluator considers a function application,

```
F(Args)
```

computes a primary and secondary key, and finds that they match the first two elements of some cache entry triple. The final element is then the result value of the function.

The cache is designed never to hit incorrectly, although it may sometimes miss unnecessarily. The contents of the primary and secondary keys are chosen to minimize unnecessary cache misses. An unnecessary miss may occur when the keys contain superfluous information that is not used when computing the function application result. If only the superfluous information in the key is different, then a cache miss will occur even though a hit would have been appropriate. This issue will be discussed in detail in Sections 5.1 and 5.2.

The primary and secondary keys are separated to facilitate cache lookup. The primary key may be computed at the point of the function application; it is used to find a set of potential cache entries. The secondary key contains the information needed during cache lookup to choose among the set of potential cache entries. See Section 7.3 for details of cache lookup and use of the primary and secondary key.

5.1 The primary cache key

For the function application

```
F(Args)
```

the primary key is computed from the body of `F` and the actual parameters, which include `Args` and any defaulted parameters corresponding to explicit formal parameters. The primary key doesn't capture all of the function's parameters since it doesn't include the implicit parameters, also known as the function's free variables (see Section 3.5.3). These are the province of the secondary key, whose construction is described in the next section.

The primary cache key is chosen to quickly narrow the choices of potential cache entries. The primary cache key doesn't contain just `F` without `Args` because this would produce too many potential cache entries after lookup based on such a broad key. For example, in a given model there are many applications of the `Compile` function, but there are usually very few different cases of applying `Compile` with the same source object.

As mentioned at the beginning of Section 5, the primary cache key is chosen to minimize unnecessary cache misses. The primary cache key contains the body of `F` but not the static environment of `F`. If the static environment of `F` were included it might cause unnecessary cache misses since many elements of the static environment of `F` may not actually be used when `F` is applied. For example, consider the following model:

```
DIRECTORY
  AVax.def = uid-1,
  AAlpha.def = uid-2,
IN {
  intfs = FUNCTION target ... IN
    IF eq(target, "VAX") THEN
      M2$Compile(AVax.def)
    ELSE IF eq(target, "ALPHA") THEN
      M2$Compile(AAlpha.def)
    ELSE
      error("target value is incorrect")
    END
  }
```

The static environment of the function `intfs` contains both `AVax.def` and `AAlpha.def`. For any given application of `intfs`, however, at most one of these names will actually be used.

The static environment of `F` needn't be included in the primary cache key because the names that are actually used from it will appear in the secondary cache key. This is discussed in the next section.

5.2 The secondary cache key

The secondary cache key is created from the implicit parameters of the function application. Recall from Section 3.5.3 that implicit parameters are the *free variables* in a function body. A free variable in a function is a name referenced in the function that is not defined in the body of the function and is not one of the formal parameters of the function. In the following function:

```
FUNCTION M2 ... IN {
  M2$Compile(A.def),
```

```
  M2$Compile(B.def)
  }
```

the function's explicit free variables are `A.def` and `B.def`. When the function is applied, many more implicit free variables may be discovered that are defaulted parameters to the calls to `M2$Compile`.

The free variables and their values are used to construct the secondary cache key. As shown in this example, the free variables can't be determined by a static examination of the function body (see Section 3.5.3), so the evaluator discovers them incrementally as it interprets the body. Roughly speaking, for each free variable it encounters, the evaluator adds a name-value pair to the secondary cache key. (A more precise description of the secondary key construction appears in Section 7.3.)

Two different applications of the same function may have different free variables. Consider the following function:

```
f = FUNCTION target ... IN
  IF eq(target, "VAX") THEN
    M2$Compile(AVax.def)
  ELSE IF eq(target, "ALPHA") THEN
    M2$Compile(AAlpha.def)
  ELSE
    error("target value is incorrect")
  END
```

If the function application is:

```
f ("VAX")
```

the set of free variables includes `AVax.def`, while for

```
f ("ALPHA")
```

the set of free variables includes `AAlpha.def` instead.

Like the primary cache key, the secondary cache key is also chosen to minimize unnecessary cache misses. Unfortunately, there is one case in Vesta where the secondary key may cause an unnecessary miss. This may occur when the free variable is a composite value (either a list or a binding). In this case, the entire value is recorded as the free-variable value. This simple technique is easy to implement, but it can lead to unnecessary reevaluation. For example, consider

```
g = FUNCTION ... IN
  M2$Prog((M2$Compile(A.def)))
```

The name `M2` is a free variable in the function `g`; it is a binding that contains (at least) the functions `Prog` and `Compile`. On a subsequent application of `g`, if any component of `M2` is different then the application will be reevaluated instead of getting a cache hit. This will happen even if the different component was never used in `g`, say the `Library` routine. A more careful alternative scheme for recording dependencies is briefly discussed in Section 10.5.

There's one more wrinkle about free variables and the secondary cache key: free variables may actually be bound by the function's static environment. For example, consider the model from Section 5.1. For any given evaluation of `intfs`, at most one of the names `AVax.def` and `AAlpha.def` will be encountered by the evaluator as a free variable, and its value will come from the static environment. For any given application of `intfs`, only the value that is actually used will contribute to the secondary key.

5.3 Determining free variables

As discussed in Section 5.2, each cache entry contains a list of free variables that were accessed during the corresponding function application. This section elaborates on how a free variable list is computed.

A free variable for a given function application may be accessed within the body of that function, or it may be accessed within the body of some function called by that function. Consider the fragment:

```
ship = FUNCTION dir ... IN
  ShipM2Prog(dir, "Bundle", Bundle);
buildAndShip = FUNCTION dir ... IN
  LET M2 IN {
    Bundle = Prog((Compile(Bundle.mod), SL-basics));
    ship(dir) };
test =
  LET building-env.v$default() IN buildAndShip("/tmp")
```

`ShipM2Prog` is a free variable in the function `ship` because it is referenced there but not defined. `ShipM2Prog` is implicitly referenced in `buildAndShip` because it is an implicit parameter to the function `ship`, and since it is referenced but not defined in `buildAndShip` `ShipM2Prog` is a free variable there also. (The name `ShipM2Prog` appears in the binding returned as the result of the function application `building-env.v$default()` in `test`.) In general, the discovery of a free variable during an evaluation may require that free variable to be added to several free variable lists.

To implement this requirement, the evaluator interconnects cache entries as follows. Each dynamic function application has an associated cache entry. The cache entries are threaded with parent pointers in a structure that reflects the dynamic call graph of the evaluation. Each cache entry points to the cache entries of the functions that called it.

Now, when a name is looked up in the environment, the evaluator maintains free variable lists by the following simple algorithm:

1. Is the name bound in the current function body? If so, the algorithm terminates.
2. Add the free variable to the list (secondary key) of the cache entry associated with the current function body.
3. Set the current function body and its cache entry by following the parent pointer, then go to step 1.

5.4 Use of fingerprints

The primary and secondary cache keys were previously described as being "computed from" various values. Since these values are potentially large, the reader may have suspected that some sort of hashing technique is employed to reduce the keys to a bounded size. Indeed this is the case—the evaluator uses a particularly convenient kind of hashing code called a *fingerprint*.

Fingerprints provide small, fixed-sized identifiers for values, as described by Broder [Broder]. Because values are unbounded in size and fingerprints are fixed-sized, many values can map to the same fingerprint, as is the case of any fixed-length hash. Unlike ordinary hashing, however, such an occurrence is designed to be extremely uncommon with fingerprints of modest size. As a result, if two fingerprints are equal, the values they stand for are the same, with overwhelmingly high probability.[4]

Using fingerprints in cache entries instead of values saves both space and time. A fingerprint comparison is faster than performing a structural comparison of the values, except for the very smallest values. Replacing values with their fingerprints saves space in a cache; this saving is substantial since values can get quite large.

6 Values in cache entries

The previous section (Section 5) describes the basic structure of a cache entry, which is independent of the values that occur in the cache entry. This section

---

[4] As a rule of thumb, the probability of collisions grows quadratically with the number of values compared, and linearly with their size. In our current implementation, a rough estimate of the probability of collision is $n^2 * m/2^{95}$ where n is the number of values considered, and m their average size.

examines the individualized treatment that is required by certain values that may appear in cache entries.

The Vesta evaluator supports lazy values, error values, and closure values (among others), and these values can lead to problems when they appear in cache entries. As discussed in this section, these problems are solved by deleting or fixing up the cache entries after they are built but before they are available for lookups, so that these lookups can proceed efficiently.

This section describes lazy values in the result value or the secondary key of a cache entry; error values in the result value; and closure values in the result value or secondary key. This section also discusses the related case where the result of evaluating a model (which can be an arbitrary value) appears in the primary or secondary cache key of a cache entry.

6.1 Lazy values in the result value

As described in Section 4, lazy evaluation is used during model interpretation. After a model is completely evaluated, there may be some lazy values produced during evaluation that were never unlazied. These lazy values were not needed to compute the result value of the model, but they may appear as the result value of some cache entry. For example, consider the following model:

```
libs = FUNCTION ... IN {
  filelib = file.v$lib();
  memorylib = memory.v$lib() };
build = FUNCTION ... IN
  LET M2 IN {
    log =
      Prog((Compile(Log.mod), libs()$filelib, SL-basics)) };
test =
  LET building-env.v$default() IN build();
```

After `test` is evaluated, there will be a cache entry for `libs()`. The result of `libs()` will be a binding, but since the `memorylib` component was never used during the evaluation, it will be bound to a lazy value in the result.

A lazy value is represented as an unevaluated expression tree together with the environment current at the time the lazy value was created; typically such a value is quite large. Because of this, cache entries with lazy values in their result are not persistently cached. At the end of an evaluation, the cache is scanned and any cache entry with a lazy value in its result is discarded. In this example, the cache entry for `libs()` would not be retained because the value of the `memorylib` component in the result is lazy.[5]

---

[5] At one time, the evaluator did save cache entries with lazy components in the results. In the cache entry a lazy value was converted to an *unavailable* value, which included the

6.2 Lazy values as free-variable values

To restate, lazy values are not saved as results in the cache because lazy values are too large. Should lazy values be permitted in the free variable section? Unlike the result value, free variables are represented as fixed-sized fingerprints, so the size of the free-variable value wouldn't matter.

Due to the low cost of doing so, the Vesta evaluator chooses to let lazy values appear in the secondary keys. The low cost is important because these cache entries seldom cause hits during subsequent evaluations. A hit would require exactly the same lazy value; since the lazy value contains the entire environment at the time it was created, even an irrelevant difference in the environment will produce a different lazy value (fingerprint) and cause a cache miss. This wouldn't be a problem if the evaluator included only the relevant environment values in the lazy value, but determining what's relevant requires an essentially complete evaluation of the lazy value's expression, defeating its purpose.

Here's a concrete example, derived from the one above in Section 6.1:

```
DIRECTORY
  Log.mod = uid-1,
  Log.man = uid-2,
IMPORT
  file.v = uid-3,
  memory.v = uid-4,
  building-env.v = uid-6
IN {
  doc = FUNCTION ... IN
    { Log.1 = ComputeManPage(Log.man) };
  libs = {
    filelib = file.v$lib();
    memorylib = memory.v$lib() };
  build = FUNCTION ... IN
    LET M2 IN {
      log =
        Prog((Compile(Log.mod), libs$filelib, SL-basics)) };
  test = LET building-env.v$default() IN {
    build();
    doc() };
  }
```

---

model name and an evaluation path to the expression that produced the lazy value. If an unavailable value was used during an evaluation, the evaluator would re-analyze the model to recreate the original lazy value, and then unlazy it. After implementing this approach we discovered that it was not particularly efficient; the cost of the re-evaluation often exceeded the benefits of caching. It was more efficient to save cache entries only when the result values were completely evaluated.

The binding `libs` is a free variable in the application of `build`. The binding `libs` has two fields: `filelib` and `memorylib`. Only `filelib` is used in this model; `memorylib` is therefore bound to a lazy value. That lazy value contains a number of definitions in its environment; including the binding for `Log.man`. Every time the user evaluates the model with a different `Log.man`, the `build` function will be reexecuted also, because the free-variable value `libs` is different from before.

How might the evaluator improve this situation? One possibility is to unlazy any lazy free-variable values in cache entries. This works well if the value will be needed later anyway. But if the value isn't needed, the evaluator ends up doing a potentially large amount of useless work. In this example, an entire library might be constructed (from scratch) needlessly.

The current Vesta evaluator tries to steer a middle ground between leaving lazy components lazy and completely unlazying them. It unlazies the top-level components of composite values, but leaves other lazy values unevaluated. This heuristic approach works in many cases but is not really satisfactory; for a better solution see Section 10.5.

6.3 Errors in the result value

The evaluator doesn't produce cache entries for function applications that encounter errors during their interpretation. This rule applies recursively to function applications that call the function application that got the error.

Why not? In a functional language, won't a subsequent re-evaluation just get the same error? There are two reasons, both reflecting the practical realities of using a functional language in an environment that is sometimes nonfunctional.

First, some functions produce error messages that aren't part of their formal result. Such messages are really harmless side-effects in the sense that the side-effects don't alter the result of evaluation. But if the function applications were cached, on a subsequent re-evaluation the cache hit would prevent the user from seeing the messages again—an unfortunate situation if the original messages were lost.[6]

Second, a function evaluation may fail because of a transient system problem. In the ideal, of course, this can't happen. But in reality, resources are finite and occasionally become exhausted. For example, the temporary file space available to compilers is limited, and an application of the `Compile` function may therefore fail because of inadequate temporary space. Obviously, caching this result is foolish—the user will want to retry the compilation when adequate disk space has been restored.

---

[6] One could eliminate this problem by making the error messages *be* the result of the function, so they would be retained in the Vesta cache. But, while intellectually satisfying, this approach has additional practical complications that aren't worth surmounting.

6.4 Models as free-variable values

As in the previous examples, one model references another by simply naming it. So, in the first model, a DIRECTORY expression binds the second model's name to a UID that names the object in the repository. During evaluation, when the name of the second model is encountered in the body of the first, the second model is read, parsed, and (lazily) evaluated.

The value obtained by evaluating a model may appear as a free-variable value. For example, in the model to build the Vesta server:

```
DIRECTORY
IMPORT
  vestarep.v = uid-1,
  vestaeval.v = uid-2,
  vestabase.v = uid-3,
IN {
  intfs = FUNCTION ... IN {
    vestabase.v$intfs();
    vestarep.v$intfs();
    vestaeval.v$intfs() };
  etc.
  }
```

the models `vestabase.v`, `vestarep.v`, and `vestaeval.v` are all free variables in the function `intfs`. To check for a cache hit on `intfs()`, the evaluator would appear to have to read, parse, and (lazily) evaluate all three models. That's a significant amount of work, and reduces or eliminates the usefulness of getting a cache hit.

Instead, the evaluator uses a small trick that gives a substantial performance benefit. The entries in the free-variable list of a cache entry are augmented with an additional Boolean, so that each list member is a triple:

```
<free var name, free var value, is-a-model>
```

When "is-a-model" is TRUE, the free variable value is the repository UID of a model, or, more precisely, the fingerprint of the repository UID. Fingerprinting the UID is logically just as good as fingerprinting the (lazily) evaluated model for two reasons: (1) models are immutable, and (2) a model is evaluated in the empty environment and so gets no free variables from the surrounding environment. This UID substitution is much faster, since the model doesn't have to be read or interpreted at all.

This trick dramatically reduces the evaluation time of a model that references many other models when only a few of these references are different from an earlier evaluation.

6.5 Models in the primary cache key

A similar trick is used when a model is part of the primary cache key. Specifically, when the evaluator sees the syntactic form:

```
Model$Component(Args)
```

it creates two cache entries with different primary cache keys. The first is a regular cache entry; the cache key is a function of the values of `Model$Component` and `Args`. The second is a *special-key* cache entry; its primary cache key is computed from only the UID of `Model`, the name of `Component`, and the value of `Args`.

Consequently, if there is a cache hit on the special cache entry, the `Model` needn't be read, parsed, and evaluated. Even if `Model` is different, there's a chance that the value of `Model$Component` isn't different, and so there's still the opportunity to get a cache hit on the regular cache entry.

This simple optimization reaps performance benefits for the same reasons, and in the same situations as described in the previous section.

6.6 Closures in the result value

The language supports first-class functions, which may return other functions as their results, and it is important to be able to cache these result functions. However, saving closures in the result value of a cache entry poses special challenges.

A closure contains an expression body, the formal parameter names, and the static environment. The static environment may contain lazy values. Before the cache entry is saved at the end of evaluation, the lazy values in the static environment are completely evaluated. This is done since lazy values are usually quite large and would take up too much space in the cache. (Also see the discussions of lazy values in Sections 6.1 and 6.2.)

There is one exception to unlazying all the lazy values in the static environment. If the lazy value is produced by evaluating a model, than the value is not unlazied but is converted back into the model's UID. This technique is necessary because completely evaluating a model may require a great deal of extra evaluation. One model usually contains a construction component and also imports other models, which in turn import other models which contain construction components and import other models, etc. Completely evaluating a model and its environment could end up causing a complete build of every model in the system!

To make it possible to save closures in result values, the language definition imposes a restriction on the contents of the static environments of closures: only names that appear literally and are free in the body of the function definition are included in the static environment of the closure. For example, consider the following model:

```
DIRECTORY
  Text.def = uid-1,
  Text.mod = uid-2
IN {
  intfs = FUNCTION ... IN
    M2$Compile(Text.def),
  impls = FUNCTION ... IN
    M2$Compile(Text.mod)
  }
```

The functions `intfs` and `impls` are evaluated in the same static environment; this environment contains definitions for both `Text.def` and `Text.mod`. The restricted static environment of the `intfs` closure contains only `Text.def`, since that is the only name in the static environment which appears literally in the body of the function. Likewise, the restricted static environment of the `impls` closure contains only `Text.mod`.

This rule means that

```
(FUNCTION IN body)()
```

is not equivalent to

```
body
```

because the former may have fewer free variables when evaluating `body`, and this can make a difference if `body` calls functions that take implicit parameters. This is an unusual restriction, but it was found to be necessary to make closures work well with caching.

A closure's static environment is restricted for three reasons:

- The unrestricted static environment can be quite large and would require too much space to store in a cache in many cases.
- Every value in the static environment needs to be unlazied before the closure can be cached; if the static environment contains superfluous entries, unnecessary evaluation can occur.

- When a closure appears in the result value of a cache entry, every name in its static environment is a potential free variable for the cache entry.[7] If there are superfluous names in the static environment then these superfluous names will also appear in the cache-entry free variable list. A different value for one of these superfluous names would cause an unnecessary cache miss.

Even the restricted static environment saved in a cached closure may be an overestimate of what the function body actually requires. Consider the model:

```
DIRECTORY
  Text.def = uid-1,
IN {
  intfs = FUNCTION ... IN
    LET {M2; flume()} IN {
      Compile(Text.def);
      etc. };
  }
```

The static environment of `intfs` will contain a binding for `Text.def` since `Text.def` appears literally in the body of `intfs`. Once the function `intfs` is applied, however, it may turn out that `Text.def` is not a free variable of `intfs`, if the function application `flume()` returns a binding for `Text.def`.

Although the restricted static environment may contain an overestimate of what is actually used from the static environment, this has not been a problem in practice. Given the way models are currently written, the conservative static environment usually produces an environment that is close to the ideal.

6.7 Closures as free-variable values

As mentioned above, a closure is composed of an expression body, formal parameter names, and a static environment. This means that when a closure is added as a free-variable value, its static environment is included.

When a closure appears in the secondary cache key this key includes the closure's static environment, restricted as described in the previous section.

---

[7] To see why the names in the closure's environment are listed as free variables in the cache entry, consider the fragment:

```
LET {x = 3} IN {
    g = FUNCTION IN { h = FUNCTION IN x };
    g() }
```

`x` is in the static environment of the closure bound to `h`. According to the rule above, all the names in the closure's environment are on the cache entry's free variable list. Since `x` is defined outside the body of `g`, `x` is on the cache entry's free variable list for `g()`. If this were not the case, on a subsequent cache lookup the value of `x` might be different but there would still be a cache hit on the old entry for `g()`, and this would be incorrect.

Because the static environment is included in the key, an evaluation may occasionally miss unnecessarily; if the closure environment contains a definition that is not used in the function application, and that definition is different on a subsequent cache lookup, a miss occurs even though there's no need to reevaluate the expression. Consider the model:

```
DIRECTORY
  AVax.def = uid-1,
  AAlpha.def = uid-2,
IN {
  privateIntfs = FUNCTION target ... IN
    IF eq(target, "VAX") THEN
      M2$Compile(AVax.def)
    ELSE IF eq(target, "ALPHA") THEN
      M2$Compile(AAlpha.def)
    ELSE
      error("target value is incorrect")
    END;
  intfs = FUNCTION target ... IN
    LET privateIntfs(target) IN {
      etc. }
  }
```

The closure `privateIntfs` is a free variable in the application `intfs()`. The static environment for `privateIntfs` contains both `AVax.def` and `AAlpha.def`. Depending upon the value for `target`, only one of these interface modules will actually be used by any given application of `privateIntfs`. But if either of them is different in a later evaluation, there will be a cache miss on the function application for `intfs`.

This simple approach to handling closures as free-variable values usually performs well, but it does produce unnecessary misses on occasion. It is difficult to do better and not slow down cache lookup noticeably.

7 Caches and cache lookup

The previous sections have discussed cache entries, without considering how the cache entries are organized so that accesses will balance speed and accuracy. Since cache entries are stored persistently (in the repository), disk I/O is required to read them. Storing each cache entry in a separate file would yield an impractically slow evaluator, so related cache entries must be *clustered* into *caches*. During an evaluation, various caches are accessed as needed and at the end of an evaluation a new cache is created.

A high cache hit rate is important for performance. If missing in the cache causes a moderate-sized compilation, that costs 1,000 times more than a hit. The miss rate is also important to users regardless of how long the compilation takes; users hate to see a module recompiled if they know the compilation is redundant. Still, searching through all the cache entries ever generated is clearly too slow, so entries must be clustered in such a way that only the likely ones are searched.

This section examines how cache entries are grouped into caches, how caches are created, and how cache lookup occurs during evaluation. There is also a discussion about how cache entries are purged from caches. The section concludes by describing several flaws in the current design.

7.1 Caches used during cache lookup

A cache is associated with a particular version of a model, and each model has exactly one cache, once evaluated. Each cache is stored in a file in the repository under a UID that is computed from the model's name; given the name of a model, its cache can be found. Because a cache is associated with a model, a cache produced by one user may be used by another; this allows one user to take advantage of another's work.

During evaluation, a list of currently interesting caches, as explained below, is maintained. Before each function application a cache lookup occurs in each interesting cache until a cache hit occurs or until misses occur in all caches.

During evaluation, a cache is labeled as one of two types. The first type of cache is an *ancestor cache*, associated with the model that is the immediate ancestor of the root model that the user is evaluating. The second type of cache is an *imported cache*, associated with some model that is imported directly or indirectly by the root model. Note that the cache type is not an inherent property of a cache, but a property that the cache possesses during a particular evaluation. The importance of these two cache types will become apparent in Section 7.2 below.

The list of interesting caches grows and shrinks during an evaluation. Initially, the list contains only the ancestor cache. During the evaluation, an imported cache is added to the list whenever the evaluator evaluates a new imported model, and is removed from the list when the evaluator finishes with the model. In other words, the list of interesting caches contains those caches in whose models evaluations are currently proceeding, including the ancestor cache.

7.2 Organizing cache entries into a cache

When a model is evaluated, a single new cache is created for the root model. The imported cache and the ancestor caches are not modified; any new cache entries created during the evaluation go into the new cache for the root model.

Also, any cache entries from the ancestor cache (but not from imported caches) for which a hit occurred are copied into the new cache. Consequently, the new cache for the root model contains all the entries that were created during that evaluation, or would have been created if cache hits had not occurred in the ancestor cache during the evaluation. At the end of an evaluation the new cache is saved to persistent storage.

7.3 How cache lookup works

To evaluate a function application, the evaluator performs the following steps.

1. Evaluate the function parameters.
2. Consider the function expression. If it is of the form `Model$Component`, then compute the special key (see Section 6.5) and perform a cache lookup (described in detail below). If the lookup gets a hit, return the result value from the cache entry as the result of the function application.
3. Evaluate the function (which, in the case of a miss in step 2, will cause `Model` to be read, parsed, and evaluated).
4. Construct the static and dynamic environments for the evaluation of the function body, and bind the actual parameters to the formals.
5. Perform a cache lookup (described below). If the lookup gets a hit, return the result as in step 2.
6. Evaluate the function body using the environment constructed in step 4.

In detail, the cache lookup proceeds as follows.

A The evaluator first computes a primary cache key by fingerprinting the function and its parameters. This computation is different for the two lookups in step 2 and 5 above, as described in Sections 5 and 6.5.

B For each cache on the list of interesting caches, the evaluator indexes (i.e., hashes into) the cache using the primary key computed in step A. If this produces one or more cache entries, then step C is performed on each of the entries until either a cache hit occurs or the entries are exhausted. If a cache hit occurs, the lookup terminates with a hit; otherwise, the evaluator repeats this step on the next member of the list of interesting caches. If no members remain, the lookup terminates with a miss.

C The evaluator extracts the secondary cache key from the cache entry under consideration. This is the free variable list (Section 5.2) containing triples of the form `<name, value, is-a-model>`. For each triple on the list, the evaluator looks up `name` in the environment; if it is undefined, this cache entry is a miss. Otherwise, the `value` field is compared to either the UID bound to `name` (if `is-a-model` is TRUE) or the fingerprint of the value of `name` (if `is-a-model` is FALSE). An equal comparison causes the evaluator to repeat this step on the next triple of the free variable list. If there are no more entries on the free variable list, a cache hit occurs. An unequal comparison causes the cache entry to miss, and the lookup algorithm continues with the next cache entry per step B.

7.4 Purging cache entries

Once a new cache is created at the end of an evaluation, it is never modified. Individual cache entries are not purged from a cache, but the entire cache and all its cache entries are deleted at the same time.

This approach is appropriate given the way cache entries are organized into caches. The clustering approach for cache entries was chosen to simplify purging them. A cache is associated with a model, and a cache contains only the cache entries that are needed when evaluating its model.

The cache is therefore just another derived object that is created during the evaluation of the model. Deciding when to delete a cache (or any other derived object created during the evaluation) is not under the purview of the evaluator, but is an administrative issue. The Vesta repository paper [Chiu and Levin] includes a discussion of how and when this deletion occurs.

7.5 Bridges and repository caching

The preceding description of caching discussed the techniques used by the evaluator. There is an independent caching mechanism, *repository caching*, that bridges can exploit to improve on the evaluator's scheme.

During the evaluation of most bridge functions, the bridge will produce one or more derived objects to be stored in Vesta's repository, under a name based on the bridge function and its parameter. This is done using the repository's `NameDerived` function (see Section 2.6). To implement repository caching, the bridge computes the name to be given to the derived, and checks in the repository to see if the object already exists.

For a bridge to perform repository caching, it must be able to determine all the inputs to a function efficiently without actually performing the function. In some bridge functions this is easy and repository caching is a performance accelerator; in others the bridge must actually perform most of the function to determine the inputs and so bridge caching does not pay off. For example, when compiling a Modula-2+ module, only the beginning of the module must be read and parsed to determine the collection of necessary compiled interface modules. In comparison, to determine the inputs for compiling a C file, all of the C header files must be read and parsed. This may take a large fraction of the total compilation time, and so repository caching may not improve performance.

The evaluator doesn't do repository caching itself because it would need to know all the parameters used to create a derived object, and it is the bridge that determines this in a language-specific way. Repository caching is entirely optional; each bridge can implement it for the functions that can benefit.

7.6 Problems with the current design

There are two problems with the current approach to grouping cache entries into caches. The first is that the cache miss rate is higher than it should be. The second is that, given today's memory costs, memory usage is too great for Vesta to be a viable commercial product. Each of these is discussed in this section.

Because cache entries are associated with root models, cache misses can occur even though a valid cache entry exists in some cache. Consider the following two models umbrellaOne.5 and umbrellaTwo.8. Both models import the same version of `lib.v` and the same version of `building-env.v`. Both evaluate the functions applications `lib.v$intfs()` and `lib.v$impls()` in the same `building-env.v` environment. The cache entries from these two applications in umbrellaOne.5 would provide hits for the same applications in umbrellaTwo.8 if these entries were available during cache lookup. But the cache entries from the evaluation of umbrellaOne.5 are stored in the cache for umbrellaOne.5, and the current design does not provide a way for the evaluation of umbrellaTwo.8 to find them.

```
umbrellaOne.5
--------------
DIRECTORY
IMPORT
  building-env.v = uid-5,
  lib.v = uid-6,
  etc.
IN
  intfs = FUNCTION ... IN {
    lib.v$intfs();
    etc. };
  impls = FUNCTION ... IN {
    lib.v$impls(),
    etc. };
  test = LET building-env.v$default() IN {
    intfs(); impls() };
```

```
umbrellaTwo.8
--------------
DIRECTORY
IMPORT
  building-env.v = uid-5,
  lib.v = uid-6,
  etc.
IN
   intfs = FUNCTION ... IN {
    lib.v$intfs();
    etc. };
  impls = FUNCTION ... IN {
    lib.v$impls(),
    etc. };
  test = LET building-env.v$default() IN {
    intfs(); impls() };
```

In this case, repository caching can prevent the bridge from actually having to redo the compilations in `lib.v$intfs()` and `lib.v$impls()`. This is not an ideal solution, since bridges don't always implement repository caching. For example, if the models are for C programs instead of Modula-2+ programs, repository caching won't save much time even if it is implemented; Section 7.5 explains this point in detail. Also, while repository caching may eliminate individual compilations, an evaluation could run much faster with cache hits at internal function applications. For example, if `lib.v$impls()` contains a thousand compilations, the 1000 repository cache hits will take longer than getting one internal cache hit on the application of `lib.v$impls()`.

The second problem with the current design is excess memory usage; the prototype used more memory than would be reasonable for a commercial system today. In the current design, caches are read for every model that is used during the evaluation. Many times there are no useful cache entries in a cache, but the cache is read anyway because there is no simple way to discover in advance that there are no useful cache entries.

Both these problems must be addressed to make the Vesta system performance commercially acceptable. Section 10.4 suggests a different approach.

Part III

Experience

8 Performance

This section argues that the run-time performance of the Vesta system is comparable to that of the Unix utility `make` [Feldman] when `make` is used in a way that avoids inconsistent builds. To support this claim, this section compares the times to build several typical Modula-2+ programs using the two systems. The timings were taken on a Firefly personal workstation [Thacker *et al.*] [8]. The relevant information about each test program is given in Table 1.

The time discussed in this section is from the end of the user's source file edits to the completion of the build, minus the time spent in the compiler and linker (which are the same in each case). For Vesta the time is divided into two parts: the time to create a new immutable model ("Advance") and the time for the evaluator to interpret the new model. For `make` the time is divided into three parts: the time spent in the `edsel` and `imc` utilities that are used with `make` to avoid an inconsistent build for Modula-2+ programs, plus the time for `make` to process the `makefile`.

These timing divisions highlight the fact that `make` and Vesta provide different guarantees. Vesta creates a consistent software system every time since it rebuilds whatever parts of the environment are required. Systems built with `make` do not build the environment when the client program is constructed; instead, libraries and other environment components are built earlier and simply used during the construction of the client program. In Vesta terminology, the environment components are treated as source objects during the system construction. The `make` building style requires additional tools to perform checks to verify a consistent build. The tools described here are specific to Modula-2+, but similar tools have been used with other languages.

The `edsel` utility used with `make` for Modula-2+ programs is similar to `makedepend` [Brunhoff] in functionality and performance, but is set up to run automatically from the `makefile` during each build. `Edsel`, in conjunction with `make`, causes the necessary recompilations to be performed for modules that are listed in the `makefile`. Typically, these are the modules in the `makefile` that are local to the system being built.

The Inter-Module Checker (`imc`) handles Modula-2+ libraries that are linked into the client program but are not built by the client program's `makefile`. Because the libraries are not necessarily built in the same environment as the client program, they can be accidentally built with different interfaces. The `imc` checks interface consistency of all the modules that go into a system and issues

---

[8] Although the Firefly is a multiprocessor, neither Vesta nor `make` takes explicit advantage of its multiprocessing capabilities.

program name	*total source lines*	*number of modules*
Vesta server	103,295	195
loupe	24,336	177
vrweed	4,563	3
hello	6	1

Table 1: Program information

program name	*make (edsel+imc+make)*	*Vesta (Advance+Vesta server)*
Vesta server	1:26	1:10
loupe	:59	:39
vrweed	:20	:08
hello	:07	:08

Table 2: Complete build

an error if they are not built with the same interfaces. Note that the `imc` can not produce a consistently built program; it merely can issue an error message if it determines that a consistent program cannot be built. It is then up to the programmer to determine which modules are inconsistent and what needs to be rebuilt.

Tables 2 and 3 give timings for a complete build of several programs. A complete build is required, for example, when some basic library interface is modified or a new compiler is introduced. The `make` timings were taken after all the deriveds in the program were deleted, which is much faster than depending on `edsel` to discover the need for recompilation. (To see this, compare the `edsel` times reported in Table 3, where all deriveds are deleted, with the `edsel` times reported in Table 5, where the deriveds cannot be deleted.) Tables 2 and 3 show that, except for the smallest builds, Vesta is 19% to 60% faster than `make` plus `edsel` and `imc`.

Tables 4 and 5 give timings for a build after a change is made to a single implementation module. In this case, Vesta is 46% to 67% faster than `make` plus `edsel` and `imc`.

Table 6 gives the percentage of the total build time spent in Vesta facilities, which includes time spent in Advance and the Vesta server. The percentage of the total build time in Vesta ranges from 1% to 16%.

program name	*edsel*	*imc*	*make*	*Advance*	*Vesta server*
Vesta server	:19	:19	:48	:32	:38
loupe	:10	:30	:19	:21	:18
vrweed	:08	:10	:02	:06	:02
hello	:05	:01	:01	:06	:02

Table 3: Details of the complete build

program name	*make (edsel+imc+make)*	*Vesta (Advance+Vesta server)*
Vesta server	2:03	:56
loupe	1:12	:39
vrweed	:24	:08

Table 4: Single-module change

program name	*edsel*	*imc*	*make*	*Advance*	*Vesta server*
Vesta server	1:11	:19	:33	:32	:24
loupe	:24	:30	:18	:22	:17
vrweed	:13	:09	:02	:06	:02

Table 5: Details of the single-module change

program name	*complete build*	*single-module change*
Vesta server	1%	16%
loupe	1%	15%
vrweed	4%	6%
hello	10%	

Table 6: Percentage of total build time spent in Vesta

9 Comparison with other systems

9.1 Cedar System Modeller

Many of the ideas in Vesta originated in the Cedar System Modeller, and Cedar's System Modelling Language (SML) and the Vesta language have many similarities [Lampson and Schmidt]. The Cedar System Modeller was used in a single-language environment to maintain software written in Cedar.

The Vesta language and SML share a number of basic features. SML is functional and modular, and strongly typed. Bindings are commonly used. First-class functions are supported. System descriptions are written in terms of source objects. System descriptions are complete; they describe all the components that go into building the system.

While there are similarities between the two languages, the System Modeller supports a description language with an elaborate and complex type system. Types in SML are first-class values, and SML implements static type-checking instead of dynamic type-checking like the Vesta language.

The SML type system permits an elegant system description in which a Cedar interface is a type, and the corresponding Cedar implementation module is an instance of that type. With SML, static type-checking provides earlier warnings about certain kinds of mistakes in a system configuration earlier in the construction process. Unlike the Cedar System Modeller, though, the Vesta system was designed to work in a multi-language environment that could incorporate all existing programming languages. We discovered that in this environment static type-checking was not feasible; see Section 3.7 for details.

The SML evaluator caches the results of compilation and linking, but does not cache the results of other function applications. The SML approach to caching is similar to that used by `make` and the `make` variants described below. While this approach provides adequate performance when evaluating small and medium-sized software systems, the overhead becomes significant when evaluating large systems with thousands of software modules.

9.2 Make

`Make` and systems similar to it are perhaps the most prevalent software building systems [Feldman]. `Make` has been implemented with slightly different features by a number of vendors, but this section does not need to distinguish between the different versions.

The `make` system is not integrated with a version control system. To use `make` with one of the common version control systems such as RCS [Tichy] or SCCS [Rochkind], the files to be compiled must be checked out into a local building area. With Vesta, files do not have to be checked out to be compiled.

The Vesta language has a well-defined semantics that programmers should find familiar, especially programmers with knowledge of functional languages.

In contrast, `make`'s semantics are more operational. To understand a complex `makefile`, a reader must often imagine the actions `make` might take at each step.

With `make`, the user describes the software system in terms of the source and object modules that go into the system. The user must think in terms of derived object names, which becomes complex when a system can be built in various ways for multiple platforms, instruction sets, etc. With Vesta, the user describes his system in terms of only the source objects used to build the system.

With `make`, dependencies for each activity must be explicitly listed; if a user omits a dependency an object might not be rebuilt when it needs to be. Consequently, language-specific generators are often used to generate a `makefile` with the correct dependencies, but for safety the `makefile` may need to be generated before every build and running the generator is often time-consuming. With Vesta, the system will determine the dependencies during each evaluation of a model. There is no opportunity for a user to fail to rebuild, as with `make`; in Vesta, an object will always be built if it hasn't been created previously.

Make depends upon environmental factors, such as Unix environment variables, that are not specified in the `makefile`. For example, Unix `make` relies on the user's PATH variable to include the right directories. With Vesta, all dependencies are explicitly specified in the model.

With `make`, only file dependencies are taken into consideration when `make` decides whether to recompute a derived. This means, for example, that changing a compilation switch does not cause rebuilding. Also, `makefiles` can't prevent compilers from using environment variables; since the settings of environment variables are not written down in a `makefile` this is another unrecorded dependency.

`Make` uses file dates to determine what to rebuild; if an object predates the modified date of any object it depends on, that object is rebuilt. Whether or not an object is rebuilt depends upon the time stamps of the object and its dependencies, not on whether or not any of the modules in the configuration change. This approach is sufficient as long as progress involves all modules in a system moving forward in time. But consider the following scenario. A user is working on system P, which imports version 5 of interface I from the public area. The user successfully builds system P, but during testing he discovers an error with version 5 and switches back to using version 4 of interface I instead. Since version 4 is older than version 5, `make` will not rebuild any of the modules in system P that import I. As with the other problems with `make`, the user is required to work around this limitation by hand.

`Make` does support recursive invocation of itself to permit decomposition of a large system into several smaller makefiles. The mechanisms for passing a large number of parameters from one `makefile` to another is clumsy, but this does provide a limited form of `makefile` modularity.

9.3 Make variants

Several other systems exist that provide the same basic features of `make`, but include additional features that attempt to correct certain of `make`'s problems. The `imake` [Moraes] system was created to deal with building the X Window system; ODE-II [Open Software Foundation] was created to build OSF software; ClearCase [Atria Software] is a commercial product that extends `make`.

ODE-II and ClearCase provide a way to integrate with a version control system. In both cases, compilations can occur without having files checked out to a local area.

All three systems, `imake`, ODE-II, and ClearCase, use an extended form of `make` as their description language. In all three cases the software system is described in terms of the object modules that go into the system, as with `make`.

The three systems express dependency information differently. In ODE-II, all dependencies must be explicitly listed. On the other hand, `imake` uses a dependency generator that works for C code only; presumedly dependencies for other languages need to be explicitly specified. ClearCase does not require build dependencies to be explicitly specified, but build order dependencies of derived objects must be specified. Vesta also requires build-order dependencies to be specified, but the way this is done differs. In ClearCase, the dependency is specified using object files and build rules; in Vesta the order dependency is specified using scoping and implicit parameters.

With `imake` and ODE-II, only file dependencies are taken into consideration when deciding whether to recompute a derived. ClearCase takes into account file dependencies as well as the "command line" that invokes the language processor. However, ClearCase's use of the command line doesn't include environment variables that the processor may access from the environment. Also, a ClearCase user must perform special actions so that tool versions are included in the dependencies.

ODE-II and `imake`, like `make`, use file dates to determine what to rebuild. ClearCase does not; an object is rebuilt when any object it depends upon is different than before.

None of the three systems provides modularity equivalent to Vesta's. They do provide a common file that stores the common rules or definitions in one place that may be referenced from separate `makefiles`. These rules therefore needn't be duplicated in each `makefile` and a change to the common file will affect all the `makefiles` that include it. For example, `imake` supports building for different machines by allowing machine dependencies (such as compiler options, alternative command names, and special make rules) to be kept separate from the descriptions of the objects to be built.

10 Future directions

The Vesta system was in use at SRC for more than one year. During that time several areas were identified for future work. Some of this work, described in Sections 10.1, 10.2, and 10.3, is needed to improve system usability. Other work, described in Sections 10.4, 10.5 and 10.6, is needed to reduce memory usage and increase speed.

10.1 Form models

The Vesta language offers extreme flexibility in writing models. However, as in all languages, flexibility provides the opportunity for both good and bad structure. We'd like to allow proven structures to be written more easily without taking away the "escape hatch" that is necessary to do something unusual.

After writing models for more than a year, we noticed that several basic patterns for model-writing began to emerge. To make it easier to write new models, form (or template) models should be developed. Such form models should be quite useful because in a typical model there is no new structure; models of the same type often differ only in the lists of files in the system.

Form models will reduce what a new user must understand about the Vesta language. Currently, most users use the "copy-and-edit" approach to writing models; a user will find a model for a similar library or application, copy the model, and edit the appropriate parts. Form models will codify this procedure so that a user can be certain of using an reliable format.

10.2 Vesta language debugger

Users often write models with bugs, and the models are often complex enough that finding the bugs can be difficult. A debugger for Vesta models would help users understand what goes wrong when they write incorrect models.

Most of the features that users want in a Vesta debugger are standard ones provided by general-purpose debuggers. Also, users have requested one non-standard feature; they want to ask: "Where was this name bound to its current value?". This is important because where names are bound is not always textually apparent in models, since functions can return bindings and names can be generated by bridges.

It is difficult to imagine a similar debugging facility for use with `make`, because of `make`'s lack of formal semantics. Traces of `make`'s actions are usually all that is available.

10.3 Browsing system builds

The main browsing tools available in software development environments today are mainly language-specific use/definition tools. Future work for the Vesta system includes a browser that can answer language-independent questions about building sets of models, although it knows nothing about language-specific semantics.

Common queries might include:

- Where is identifier XYZ defined/used?
- Where is the value V used?
- What library supplies the implementation of Foo?
- Evaluate expression E in a given scope.

Another area for work would be to integrate language-specific browsers with the Vesta browser.

10.4 Grouping cache entries into caches

Section 7.6 described two problems with the current approach to grouping cache entries into caches: the cache miss rate is too high and the memory usage per server is too great. A solution is briefly sketched here; it uses a *cache server*.

A cache server is responsible for storing cache entries and for providing a cache lookup function. Cache entries are kept in files on disk; all cache entries for a given primary key are grouped into a single file called a PKFile. PKFiles are read as needed during cache lookup. The cache server maintains an in-core hash table of new cache entries that haven't yet been written to disk, or of cache entries that were recently used through a cache hit.

A PKFile file is a hash table; cache entries are assigned to a bucket based upon the values of the free variables that are common to every cache entry in the file. PKFiles are organized so that the whole file needn't be read during a cache lookup. Hash buckets that are read are kept small enough so that performance is not adversely affected.

With the cache server, all cache entries for a given primary key are available to every evaluation. This eliminates the current problem where an evaluation does not find an applicable cache entry even though one exists in some cache somewhere. Note that although caches are never updated in today's scheme, PKFiles will be updated as new cache entries are generated.

Cache servers will need a lot of memory, but moving this functionality to a server means that client workstations will not incur the high memory usage required by caching today. If necessary, there may be more than one cache server, with each server handling a portion of the cache entries.

10.5 A fine-grained dependency scheme

As described in Section 5.2, the secondary cache key is constructed from the free variables in the function body. This scheme is not ideal when a free variable is a composite value (i.e., a list or binding) or a closure. This section discusses improvements to the current approach.

With a list or binding value, the function may depend only on one component of the list or binding even though the free variable names the entire value. With a *fine-grained dependency* scheme, the entry on the secondary cache key will change. Recall that the current information is:

```
<free var name, free var value, is-a-model>
```

The new secondary cache key entry will be:

```
<free expression, free expression value, is-a-model>
```

This allows an entry on the secondary cache key to be a component of a composite value instead of the entire composite value.

To make this fine-grained scheme work, we must be able to determine which composite value each component value belongs to, if any, and what expression selects the component value from the composite value. The evaluator maintains the necessary information during evaluation. The expression is compact, since it is to be stored as the "name" in a secondary cache key; for a binding component, for example, a typical expression will be `binding$component1$component2`.

As discussed in Section 6.2, a lazy value may be a component of a free variable value. Currently, if the lazy value isn't completely unlazied it may later cause unnecessary cache misses, but if the lazy value is completely unlazied that may cause unnecessary evaluation. The fine-grain dependency scheme solves this problem: the only cache key entries are those components that are actually used by the function, which will already be completely simplified.

10.6 Parallel distributed evaluation

Currently a software system is completely built on the workstation which initiated the build. Distributing the building in parallel to a network of machines will decrease the total evaluation time.

There has been a fair amount work in the underpinnings of distributed evaluation, such as how to pass evaluation context efficiently between machines. The difficulty in doing distributed building in Vesta is determining when remote evaluation is warranted and how to schedule the remote machines. Practical considerations include expected length of compilation, "warmth" of file system caches on remote machines, and resource availability.

11 Conclusions

This paper introduces Vesta, a novel system for configuration management. It focuses on the modelling language used to describe configurations, and on the evaluator that processes models to build software systems.

A simple functional language is all that is needed to describe large and complex systems. This language can be used to produce exhaustively complete descriptions of real software systems in which all dependencies are explicitly captured. These descriptions contain all the sources and building instructions that go into the result, including the tools and libraries used in building the system.

Models written in the Vesta language are compact, and easy to read and write, for three major reasons. First, the language makes it easy to manipulate large groups of name-value pairs. Second, the language provides a way of defaulting formal parameters at the point of function definition. Third, the language supports defaulting of actual parameters at the point of function invocation.

The language allows for the dynamic integration of existing compilers and tools. Each new tool extends the basic types and primitives provided by the Vesta language. The language makes it easy to add a new tool by providing a single naming mechanism by which a tool can access its inputs.

Given the performance characteristics of model evaluation, a simple interpreter is adequate to evaluate system descriptions. While the interpreter itself can be quite simple, Vesta's complete system descriptions do place heavy demands on persistent caching of function applications. In particular, caching must be performed for function applications at all levels of the dynamic call graph. Caching at the leaves only (which correspond to compilation and linking) would not produce acceptable performance when evaluating complete descriptions. Since cache lookup is used heavily by the evaluator it must be carefully implemented to make evaluation efficient.

Performance measurements show that the run-time performance of Vesta is comparable to that of `make`. This is in spite of the fact that Vesta offers stronger consistency guarantees for the resulting software system.

12 Acknowledgements

Butler Lampson, John Ellis, John DeTreville, Andrei Broder, and Jim O'Toole provided useful advise and assistance on the language design and implementation. Greg Nelson and Martín Abadi wrote the initial version of the language specification; a slightly modified version appears in the Appendix. Many adventurous users withstood the vagaries of the prototype implementation.

A Language semantics

This section is a definition of the core of the Vesta language, by means of an explicit interpreter that maps an expression and an environment into a value.

The following are values in the language:

- ERR, NIL, TRUE, and FALSE
- pairs of values
- bindings and closures
- texts, integers, and UIDs
- bridge values.

The notation for pairs and lists is Lisp-like; that is, (x,y) is short for CONS(x, CONS(y, NIL)).

A *binding* is a total map from names to values. Bindings map all but a finite number of names to the distinguished value ERR. If b is not a binding, b(N) is defined to be ERR.

If b1 and b2 are bindings, define (b1 ELSE b2) to be the binding defined by the rule that for any name N,

```
(b1 ELSE b2)(N) =
  IF b1(N) = ERR THEN b2(N) ELSE b1(N) END
```

If either b1 or b2 is not a binding, b1 ELSE b2 is defined to be ERR.

The function Eval takes an expression E and two bindings, called the static and dynamic environments, and produces a value. It is defined by cases; if E is not handled by one of the cases below Eval returns ERR.

For any name N,

```
Eval(N, s, d) = (s ELSE init ELSE d)(N)
```

where init is an initial binding whose contents are a parameter to the language definition. The initial bindings contain the predefined values ERR, TRUE, and FALSE, and the built-in functions.

The $ operator in the language extracts the value associated with a name in a binding. That is:

```
Eval(E$N, s, d) = Eval(E, s, d) (N)
```

A comma-separated list surrounded by round parentheses represents a list:

```
Eval( (E_1, ..., E_n) , s, d) =
  ( Eval(E_1, s, d), ..., Eval(E_n, s, d) )

Eval( () , s, d) = NIL
```

A semicolon-separated list surrounded by curly braces represents a binding formed by evaluating a list of bindings in order, adding each one to the static environment before evaluating the next one, and returning as a result the "ELSE" of all of them. It is convenient to define first the case in which there are just two bindings in the list:

```
Eval({E; F}, s, d) = f ELSE e
  where e = Eval({E}, s, d)
    and f = Eval({F}, e ELSE s, d).
```

Notice that curly braces are placed around E and F in the recursive evaluations, so that if they contain semi-colons, the rule will apply to them again. This gives two ways of evaluating an expression like {E; F; G}, but since they produce the same result it doesn't matter.

If commas are used instead of semicolons, then each binding is evaluated in the initial environment; that is, the bindings are evaluated "in parallel" instead of sequentially:

```
Eval({E, F}, s, d) = f ELSE e
  where e = Eval({E}, s, d)
    and f = Eval({F}, s, d).
```

The comma has higher precedence than the semicolon, so that for example { E, F; G } is short for { {E, F } ; G }.

An explicit binding is created by listing name-value pairs with an equals sign between the name and the value:

```
Eval( {N=E}, s, d) =
  (FUNCTION x: IF x = N THEN e ELSE ERR END)
    where e = Eval(E, s, d)
```

The "base case" for the rules above is that if "E" does not have the form "F; G", "F, G", or "N = E", then

```
Eval({E}, s, d) = Eval(E, s, d)

Eval({}, s, d) = NIL
```

As a consequence, curly braces can be used to override the precedence rules; round parentheses cannot be used for this purpose since (X) is a list of length 1.

The language has a LET construct that is conventional except that the binding is a value:

```
Eval( LET B in E, s, d) =
  Eval(E, Eval(B, s, d) ELSE s, d)
```

The language has an IF construct that is conventional:

```
Eval(IF T THEN E ELSE F, s, d) =
  IF Eval(T, s, d) = TRUE THEN Eval(E, s, d)
  ELSEIF Eval(T, s, d) = FALSE THEN Eval(F, s, d)
  ELSE ERR END
```

The DIRECTORY expression is unique to the Vesta language. It is the top-level expression in a model, and is used to introduce specific versions of the source files that compose the software system:

```
Eval(DIRECTORY D1 IMPORT D2 IN E, s, d) =
  Eval(E, Eval({D1}, s, d) ELSE Eval({D2}, s, d) ELSE s, d)
```

A UID is a unique repository identifier which names another model or a source file. A UID is evaluated differently depending upon whether it is a model or a regular source file:

```
Eval(UID, s, d) =
  IF IsModel(UID) THEN Eval(Parse(Read(UID)), NIL, NIL)
  ELSE Read(UID) END
```

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 parameter. IsModel is a predicate that is true if and only if its parameter names a Vesta model. Parse takes a text as a parameter and returns an expression; it returns ERR if its parameter cannot be parsed into a legal expression.

A *closure* is a quadruple

```
(formals, body, environment, type)
```

where formals is a list of names, body is an expression, environment is a binding, and type is either STATIC or DYNAMIC.

There are two forms for constructing closures: one leads to type=STATIC and one leads to type=DYNAMIC. The syntactic distinction is that for the latter case, the list of formals is followed by a literal "...". Note that "..." is literally part of the Vesta language, and not an ellipsis in this description.

```
Eval(FUNCTION formals IN E, s, d) =
  (formals, E, s', STATIC)

Eval(FUNCTION formals ... IN E, s, d) =
  (formals, E, s', DYNAMIC)
```

where s' is s restricted to the set of variables that occur free in E but not in formals; this set fs is defined by induction over the structure of E:

```
fs(N)                    = N
fs(E$N)                  = fs(E)
fs((E1,...,En))          = Ui fs(Ei)
fs({E;F})                = fs(E) U fs(F)
fs({E,F})                = fs(E) U fs(F)
fs({N=E})                = fs(E)
fs({E})                  = fs(E)
fs(LET B IN E)           = fs(B) U fs(E)
fs(IF T THEN E ELSE F)   = fs(T) U fs(E) U fs(F)
fs(FUNCTION F IN E)      = fs(E)
fs(FUNCTION F ... IN E)  = fs(E)
fs(F A)                  = fs(F) U fs(A)
```

Finally there is application, which is denoted by juxtaposition. There are two cases, corresponding to the application of a static or dynamic closure, which differ only in the way they set the dynamic environment for the evaluation of the body of the closure:

```
Eval(F A, s, d) = Eval(body, s', d')
  where Eval(F, s, d) = (formals, body, env, STATIC),
        actuals = Eval(A, s, d),
        s' = BindArgs(actuals, formals, s, d) ELSE env,
        d' = {}

Eval(F A, s, d) = Eval(body, s', d')
  where Eval(F, s, d) = (formals, body, env, DYNAMIC),
        actuals = Eval(A, s, d),
```

```
            s' = BindArgs(actuals, formals, s, d) ELSE env,
            d' = (s ELSE d)
```

The function BindArgs(a, f, s, d) returns a binding of the actuals a to the formals f. Any unbound formals are bound from the environment. The actual may be a binding, list, or other value:

```
    BindArgs(a, f, s, d) = BindArgs(NIL, f, a ELSE s, d)
      where a is a binding

    BindArgs(NIL, NIL, s, d) = {}
    BindArgs(CONS(a, al), CONS(f, fl), s, d) =
      {f=a, BindArgs(al, fl, s, d)}
    BindArgs(NIL, CONS(f, fl), s, d) =
      {(s ELSE d)(f), BindArgs(NIL, fl, s, d)}

    BindArgs(a, f, s, d) = BindArgs(CONS(a, NIL), f, s, d)
      where a is neither a binding nor a list.
```

References

[Atria Software] Atria Software, Inc. *ClearCase Concepts Manual*, 1992.

[Broder] Andrei Broder. "Some applications of Rabin's fingerprinting method", R. M. Capocelli, et al. (ed), *Sequences II: Methods in Communication, Security, and Computer Science*, Springer-Verlag, New York, 1991.

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

[Brunhoff] Todd Brunhoff. *makedepend* manual page, X11R5 software distribution.

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

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

[Hughes] R.J.M. Hughes. "Lazy memo functions", *Proc. Conference on Functional Programming and Computer Architecture*, Nancy, 129-46, LNCS, 201. Spinger Verlag.

[Keller and Sleep] R.M. Keller and R. Sleep. "Applicative caching", *ACM Transactions on Programming Languages and Systems*, 8(1), January 1986.

[Lampson and Schmidt] Butler W. Lampson and Eric E. Schmidt. "Practical Use of a Polymorphic Applicative Language", *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, January, 1983.

[Levin and Mcjones] Roy and Paul R. McJones. "The Vesta Approach to Precise Configuration of Large Software Systems", Research Report 105, Digital Equipment Corporation Systems Research Center, June 1993.

[Moraes] Mark Moraes. "An Imake tutorial", X11R5 software distribution.

[Open Software Foundation] Open Software Foundation, Inc. *ODE-II Developer's User Guide*. Version 1.0, June 1991.

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

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

[Thacker *et al.*] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite, Jr. “Firefly: A Multiprocessor Workstation”, *IEEE Transactions on Computers*, 37(8), August 1988.

[Tichy] Walter F. Tichy. “Design, Implementation, and Evaluation of a Revision Control System”, *Proceedings of 6th International Conference on Software Engineering*, IEEE, Tokyo, September, 1982.

[Wirth] N. Wirth. *Programming in Modula-2*, 2nd edition, Springer-Verlag, New York, 1982.