108

Bridges: Tools to Extend the Vesta Configuration Management System

Mark R. Brown
John R. Ellis

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

Bridges: Tools to Extend the Vesta Configuration Management System

Mark R. Brown
John R. Ellis

June 14, 1993

John R. Ellis is a member of the research staff of the Xerox Palo Alto Research Center.

©**Digital Equipment Corporation 1993**

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of the Systems Research Center of Digital Equipment Corporation in Palo Alto, California; an acknowledgment of the authors and individual contributors to the work; and all applicable portions of the copyright notice. Copying, reproducing, or republishing for any other purpose shall require a license with payment of fee to the Systems Research Center. All rights reserved.

Authors' Abstract

Vesta is a highly flexible configuration management system that supports large-scale development. Vesta provides a repository of immutable objects and a functional programming language for writing concise yet complete descriptions of configurations.

A Vesta bridge is a set of related functions and data types provided by tool builders to a Vesta environment. For instance, a C bridge might include a function for compiling C sources and a function for linking compiled C sources into executable images.

Vesta has supported development on a significant scale. The Vesta prototype included several low-aspiration bridges that encapsulated existing tools without modification; these bridges were straightforward to write. Vesta also included one high-aspiration bridge, Vulcan, a compiler server based on abstract-syntax trees. Vulcan gained both functionality and performance from its integration with Vesta. Both types of bridge benefited from Vesta's single, uniform naming facility that replaced *ad hoc* name spaces of traditional environments.

Bridges themselves are described and configured within Vesta. This allows tool builders to provide consistent collections of tools, control their evolution, and manage their installation using Vesta.

Contents

1 Introduction

Most successful software development environments include an extension mechanism for adding new tools. Extensibility allows an environment to evolve as user requirements change and as new technology comes along.

This paper describes *bridges*, an extensibility mechanism of the Vesta configuration management system. We make two main points about Vesta bridges. First, bridges are a practical mechanism, allowing existing tools such as compilers, linkers, and shells to be incorporated at low cost into the Vesta system. Second, the bridge mechanism offers both improved performance and increased functionality to new tools written specifically for Vesta.

Before introducing bridges we must introduce Vesta. Vesta represents our attempt to build a configuration management system that is highly flexible and extensible yet supports large scale software development. We were inspired by the Unix system [Kernighan and Pike], a well-known software development environment whose strength is its flexibility and extensibility. We felt that three aspects of Unix contribute to its strength:

- The Unix file system makes no assumption about file format, and its hierarchical naming provides a rudimentary mechanism for large-grained modularity, parameterization, and version control.
- Programmable shells such as `sh`, `csh`, and `make` provide low-overhead, simple mechanisms for combining tools to solve specific problems.
- Unix tools are ordinary application programs that follow a few conventions for the use of facilities like standard input/output streams, argument passing, search paths, and exit status.

The Unix system is widely used commercially, but it suffers severe problems as an environment for large-scale software development. The very flexibility of Unix is its Achilles heel—because Unix imposes so little structure, programmers must rely on the disciplined use of conventions to keep a large system organized and consistent.

Thus it is now commonplace to augment Unix with a configuration management system. Typical configuration management systems, however, enhance Unix in only a single dimension by supplementing the file system. In Vesta we have augmented Unix in all three dimensions:

- Vesta supplements the Unix file system with the Vesta repository, a hierarchical, versioned store of immutable objects. Like Unix files, Vesta repository objects are uninterpreted byte sequences.
- Vesta programmers use a functional programming language to describe software systems concisely yet completely, to a degree not possible with other systems. The restriction to a functional language enables Vesta to describe large software systems entirely in terms of source objects.

- Vesta formalizes a common class of programming tools, functional tools, with its notion of bridges.

A programming tool is *functional* if it computes a function of its inputs. Nearly all existing tools used for mechanical software construction are functional. For instance, a C compiler computes an object file from a C source file, a collection of header files, and a list of compiler switches; the content of the resulting object file is completely determined by the compiler and its inputs.

A Vesta *system model*, written in the Vesta language, describes a software system that can be built using functional tools. For example, the system model of an application program specifies how to compute the application's executable program image from a collection of source objects using functions such as a compiler and a linker.

A Vesta *bridge* is a set of related functions and types provided by tool builders to a Vesta environment. For instance, a C bridge might include a function for compiling C sources, a function for making link libraries from collections of compiled C sources, and a function for linking compiled C sources and libraries into executable images.

Vesta is self-describing—the bridges themselves are described and configured within the system. This allows tool builders to provide consistent collections of tools, control their evolution, and manage their installation using Vesta.

Vesta has supported development on a significant scale. Vesta was used for one year to maintain and evolve a system of 1.4 million lines of code written in several languages by more than 25 programmers; most of the code pre-dated Vesta and was implemented using traditional Unix tools. The system included a kernel, a remote procedure call facility, a distributed file system, a window system and user-interface toolkit, a text editor, a compiler and linker, and dozens of other applications (including Vesta itself). Using Vesta, the system was ported from VAX to R3000 machines and from SRC's experimental operating system to OSF/1. We know of no other advanced configuration management system that has been used on this scale.

Here is a summary of what we learned about bridges from this experience:

- Vesta's bridge mechanism does provide practical extensibility at low cost to tool builders. We incorporated existing tools such as compilers, linkers, and shells into bridges without modification. We wrote a new bridge specifically for use with Vesta; this bridge gave new and improved functionality and performance, compared with the conventional tools it replaced.

- Bridges benefited from being embedded in a real programming language rather than the crippled description languages of systems like `make`. Vesta provided bridges with a single, uniform naming facility that replaced the *ad hoc* name spaces of traditional environments. Bridges also benefited from the built-in data types of a programming language and from the ability to define new types.

- While `make`-based environments often require language-specific tools to compute dependency sets for correct incremental builds, Vesta freed our bridges of this concern.
- The Vesta bridge abstraction addressed a major issue that is not dealt with coherently in other configuration management systems: the persistent storage and versioning of tools. The ability to describe and configure bridges using Vesta was extremely valuable as tools evolved.

The Vesta approach to tool integration and extensibility centers on its functional language and immutable object store; tools integrate with Vesta by using the language's name space and data types and by extending the language with new functions and types. Most other research projects and recent commercial products for large-scale configuration management are based on a more data-centric approach to tool integration and extensibility, using entity-relationship or object-oriented databases. Quoting from a recent paper [Thomas]:

> Over the past few years, there has been an evolution from the use of file systems as data repositories for [software engineering environments] to the use of data/object bases. PCTE's [object management system] exemplifies this evolution. The additional richness of the data models of these data/object bases allows explicit representation of relations between the objects manipulated in the [software engineering environment], including dependency information of interest for Configuration Management.

We believe this data-centric approach to be a tarpit. Dependency information can be expressed much more concisely, understandably, and maintainably in a language, as in Vesta, than as a data structure. The value added by expressing source code in a richer data model is unclear, while the costs incurred are immediately evident—the need to modify or replace a large set of tools that were designed to use files. Despite hundreds of man-years devoted to these data-centric systems, they are only starting to progress beyond toy examples, and the work to date has not addressed the hard details of issues like tool integration for consistent building.

Here is how the remainder of the paper is organized. Bridges are an integral part of the Vesta environment, so Section 2 of this paper provides necessary background information on Vesta. Section 3 details the design goals for Vesta bridges, and Section 4 describes our design to meet the goals.

Sections 5 and 6 illustrate the design space for bridges. Section 5 shows how easy it is to incorporate existing tools into bridges. It introduces the idea of *encapsulating* an existing tool to make a bridge, and illustrates a way to add a new tool to Vesta without writing a new bridge. Section 6 describes the Vulcan bridge, which is more highly integrated with Vesta than other bridges and gains significant advantages from the integration.

In Section 7 we make some final comments on Vesta's range of application and how our work on Vesta bridges relates to other research.

2 Vesta background

This section contains the information about Vesta that is necessary for a proper understanding of the subsequent sections. We don't cover any topic in depth. Two companion papers describe Vesta's repository [Chiu and Levin] and language [Hanna and Levin], while a third paper explains how Vesta solves specific problems in configuration and release management [Levin and McJones].

2.1 System models, objects, and repositories

Vesta *system models* (or *models*, for short) describe software systems completely yet concisely. A system model is complete because it names the specific version of each source object that contributes to the system and it describes precisely how the results (usually either libraries or executable program images) are generated from the source objects. Only information written in a model can affect building; other information, such as header files, libraries, and tools not controlled by Vesta, cannot affect building. A system model is concise because it is written in a programming language that supports functions and function parameter defaulting.

Vesta formalizes the distinction between source and derived objects. A *derived object* contains a result that Vesta has produced by executing a system model. A *source object* contains something that Vesta cannot reproduce. For instance, a source object may contain a source program typed by a user, or it may contain an object code library imported from another organization. A system model is a specialized type of source object. Internal to Vesta, both source and derived objects are named with unique identifiers (UIDs).

Vesta enforces the immutability of all source objects. Once a source object has been created, it cannot be modified. Instead, a new version can be created.

Because system models are complete and source objects (including system models) are immutable, the execution of a system model is reproducible: it always yields the same results. A derived object can always be reproduced from source if necessary. Therefore Vesta can (and does) manage derived objects. In this way Vesta solves the "repeatable build" problem.

A Vesta *repository* is essentially a container for source and derived objects. Programmers share their work through one or more repositories.

A repository is accessible to Unix programs as a read-only file system. For instance, any Vesta UID can be converted to a valid (though obscure looking) pathname. Source objects also have user-sensible pathnames that reflect the versioning of source objects in the repository.

2.2 The language

As described in the introduction, programmers write system models (descriptions) in the Vesta language; a bridge is a collection of functional tools that are callable from the language. So an appreciation of bridges requires some understanding of the language.

The Vesta language is functional—values are produced by repeated application of functions. Vesta allows the naming of intermediate results, but does not allow variable assignments or other side effects. The language contains a small set of built-in functions, and programmers write new functions using the `FUNCTION` construct. Tool builders provide additional functions by writing bridges in traditional systems programming languages.

The language is dynamically typed: Each value carries a representation of its type during the evaluation of expressions, and programs do not contain type declarations. The predefined types are:

Boolean	–`TRUE` or `FALSE`
Integer	–an integer
Text	–a sequence of bytes (often, but not always, ASCII text)
List	–a LISP-style list
Binding	–a mapping from identifiers to values
Function	–a function

Bridges can define new types.

The language is also strongly typed: All built-in functions produce a special `ERROR` value if passed an argument with an incorrect type. For instance, the value of

```
PLUS(1, TRUE)
```

is `ERROR`.

Though the language is dynamically typed, it provides the same guarantees of correctness as a statically typed language would. Static typing in a language like Ada ensures that all type errors in a program are detected during the construction of the program, before it ever executes. Vesta provides the same guarantee—if the evaluation of a system model succeeds, there are no type errors in the program constructed by the evaluation.

Description of a complex system involves the manipulation of large sets of named values. Vesta facilitates such manipulation by making naming environments first-class citizens called *bindings*. A binding is a set of name-value pairs in which no name is repeated. Bindings are central to the expressiveness of the language.

In the following example, the values of the two Vesta language expressions are bindings, and the two bindings are equal:

```
{ opt_level = 3, debug = FALSE }

{ debug = EQ(0, 1), opt_level = PLUS(1, 2) }
```

The language provides a number of built-in operators and functions for manipulating bindings and extracting component values. For example,

```
b$x
```

yields the value paired with the name `x` in the binding `b`. And

```
{ b1, b2 }
```

yields the union of the two bindings b1 and b2.

The `LET` expression allows a binding to be opened as a name scope:

```
LET binding IN expression
```

The value of a `LET` is `expression`, evaluated in a naming environment in which each name defined in `binding` is bound to its corresponding value. For instance, the value of

```
LET { a = 1, b = 2 } IN PLUS(a, b)
```

is `3`. `LET` is similar to Pascal's `WITH`, except that the binding can be the result of evaluating an arbitrary expression:

```
LET Env_default() IN build(p, q, r)
```

In this case, the names contained in the binding may not be apparent from a local inspection of the surrounding text. This unusual feature, naming environments as first-class citizens, is crucial to the modularity and conciseness of system models.

Another unusual feature of the Vesta language is implicit formal parameters. The Vesta fragment

```
plus_x_y = FUNCTION ... IN PLUS(x, y)
```

defines a function with no explicit formal parameters. The "..." in the function definition means that names in the function definition that are not bound by an enclosing `LET` (`x` and `y` in this example) should be considered formal parameters and looked up in the function caller's environment. Thus the value of

```
LET { x = 1, y = 2 } IN plus_x_y()
```

is `3`.

Implicit formals are a practical necessity for concise system models. Models tend to be composed of highly parameterized sets of functions that deal with thousands of names, and it wouldn't be practical to list all the functions' free variables as explicit formals. (A companion paper [Levin and McJones] contains several examples of real models.) Also, bridge functions need to access a large and infinitely varying set of implicit formal parameters. For example, when a C compiler function is applied to a C source, it must get values for all headers included directly or indirectly by the source. It would be impossible for the compiler function's definition to list all these as formal parameters, since they vary with the source. And it would be infeasible for programmers to bundle the set of headers into a binding passed to the compiler, since that would make models much, much larger.

2.3 The evaluator

The Vesta *evaluator* is an interpreter of system models (Vesta language programs). When interpreting a model the evaluator builds the software system described by the model. An interpreter (rather than a compiler) is sufficient because the expensive part of evaluating a system model is generally the calls to bridge functions.

The evaluator performs caching of function calls. That is, when the evaluator is preparing to call a function, it determines whether or not it has already called the same function with the same actual parameters (both explicit and implicit). If it has, it returns the result of the earlier call instead of performing the call again. Vesta's use of a functional language makes caching simpler.

The evaluator's cache is persistent—it is stored on disk to survive restarts of Vesta. The cache is also shared between users of a common repository. Persistent, shared caches make it practical to describe large software systems entirely in terms of source objects, instead of in terms of a few source objects and the derived forms of imported libraries.

The Vesta evaluator's ability to support large descriptions expressed in terms of source objects is a direct benefit to all bridges, because it guarantees consistent evaluation. In particular, Vesta bridges have no need for specialized tools that compute dependency sets, as are often used in systems based on **`make`** (e.g. imake's **`makedepend`**).

2.4 System environment for the Vesta prototype

The Vesta prototype is written in Modula-2+ [Rovner]. A significant benefit of using Modula-2+ is its well developed remote procedure call (RPC) facility, which simplifies the construction of programs that consist of communicating processes. Vesta makes heavy use of Modula-2+ RPC's ability to marshall an instance of an opaque reference type by creating a remote surrogate instance, called a *network reference*.

The prototype is built on the Taos operating system [McJones and Swart] and Echo file system [Hisgen *et al.*]. Taos is a Unix variant that supports standard Unix applications while also providing threads and RPC support. Echo is a distributed Unix file system providing single-site consistency in spite of the distributed implementation.

The prototype runs on Firefly multiprocessor workstations [Thacker *et al.*], which contain VAX processors.

3 Bridge design goals

The design of the Vesta bridge mechanism is motivated by the following goals. We want the bridge mechanism to meet these goals directly when possible. In other cases the bridge mechanism should help individual bridge writers meet the goals.

- *Language-integrated*: A bridge function should present an interface that exploits language facilities. For instance, rather than defining a `Compile` function with a single parameter containing the compiler's command-line options as a text string, it is better to define the individual options as separate parameters with appropriate data types like Boolean and Integer. This way the Vesta language mechanisms for parameter defaulting will be helpful to users of `Compile`. Similarly, bridges should take structured values (lists and bindings) as parameters and return them as results when appropriate.

- *Unbounded*: There should be no designed-in limits on the number of bridges or on the number or complexity of functions provided by a single bridge. Naturally, the capabilities of the machine Vesta runs on will impose practical limits.

- *Persistent*: To give repeatable behavior, bridges must be stored persistently and immutably.

- *Versioned*: To support evolution, bridges must be versioned. It must be possible to introduce a new version of a bridge function that applies to instances of types produced by earlier versions of the bridge, and to produce new versions of bridge types that can only be manipulated by corresponding new versions of bridge functions. Vesta must support the use of both old and new versions of a bridge in the same evaluation.

- *Dynamically loaded*: To ease evolution, bridges must be loaded dynamically rather than statically bound into Vesta. This capability is especially important to bridge developers. Dynamic unloading of bridges that have been idle for a long time is desirable, to avoid having the environment grow without bound.

- *Concurrent*: Vesta aims to serve a large number of concurrent users from a given repository. Bridges must not introduce bottlenecks that limit the number of users that a repository can support. In fact, bridges must give the Vesta evaluator the flexibility to compute bridge functions concurrently for a single user, if the evaluator chooses to implement parallel (and possibly distributed) evaluation.

- *Functional*: A bridge function must compute a true function of its input parameters. For instance, environment variables and file system state outside Vesta's repositories must not influence the results of a bridge function.

- *Multiple platforms*: To allow system models to be as platform-independent as possible, bridge functions should be parameterized by the target instruction set and operating system and should hide incidental differences between platforms. A bridge must be capable of evolving to support new target instruction sets and operating systems as necessary. The instruction set or operating system that is running a bridge should not affect the signatures (i.e. the argument and result types) of bridge functions.

- *High performance access to small values*: To support Vesta's performance goals, Vesta must store instances of both built-in types and bridge-defined types in its persistent caches and retrieve such values efficiently. The performance requirements are not easy to characterize precisely, but storing and retrieving Vesta values, including instances of bridge-defined types, must be more efficient than storing and accessing files in a file system. A typical instance of a bridge-defined type would be a tiny file, only a few hundred bytes.

4 Bridge design overview

4.1 Process structure

Each Vesta user runs a process called the Vesta server. This process contains a Vesta evaluator and code to access Vesta repositories. There is no repository server per repository—instead, file sharing is synchronized using the file system. (Nothing rules out doing it the other way.)

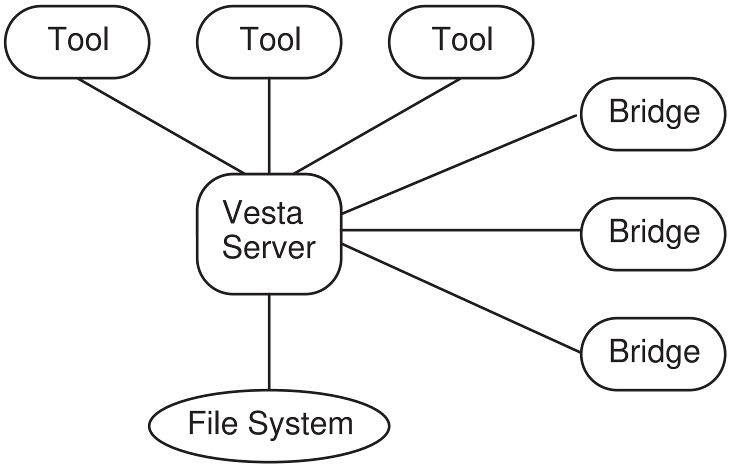

Figure 1: Vesta's process structure

Each Vesta bridge is implemented as a program. To start a bridge, Vesta starts a Unix process running the corresponding program. The bridge process communicates with the Vesta evaluator and the Vesta repository using remote procedure calls.

Vesta tools provide user interfaces to the facilities of the Vesta repository (check-in, check-out) and evaluator (build). The tools run as separate processes that communicate with the Vesta server using RPC.

These initial design choices fully or partially address the first six goals enumerated in the previous section:

- *Language-integrated*: Because the bridge communicates with the evaluator using RPC, it is straightforward to communicate complex typed values to and from the bridge.

- *Unbounded*: Since the Vesta evaluator runs each bridge as a separate process, the number of bridges is effectively unbounded.

- *Persistent*: Since bridges are programs, Vesta can store bridges using the mechanisms it uses for storing programs in general. It follows that bridges are stored as persistently as Vesta requires.

- *Versioned*: Since bridges are programs, Vesta's facilities aid in storing multiple versions. Because the Vesta evaluator runs each bridge as a separate process, it can run old and new bridge versions at the same time.

- *Dynamically loaded*: The Vesta evaluator runs a bridge on demand by starting a process. The evaluator and the bridge only need to agree on their common RPC interfaces. Similarly, the Vesta evaluator is free to kill off bridge processes to implement dynamic unloading.
- *Concurrent*: Since a bridge process is associated with a user's Vesta server, it is evident that bridges don't create any bottlenecks limiting the number of concurrent users. The Vesta evaluator is free to start multiple bridge processes in order to implement parallel or distributed evaluation.

An additional advantage of running each bridge as a separate process is the high degree of isolation it provides. The crash of a defective bridge cannot bring down a user's Vesta server.

4.2 How bridges extend the set of functions

The Vesta language includes `BRIDGE`, a built-in function that creates bridge functions. `BRIDGE` takes a single executable program as a parameter. It starts that program as a bridge process (if the program is not already running) and returns a binding containing the functions and default parameter values exported by the bridge.

That's the extent of Vesta's support for bridges at the language level. Vesta also defines an RPC interface that bridges must export, and a separate RPC interface for bridges to call. The remainder of this section is an overview of these two interfaces as bridges use them to define new functions. Section 4.3 deals with how bridges define new types.

A bridge exports two procedures called by the evaluator via remote procedure call. Here are the Modula-2+ signatures for these procedures:

```
PROCEDURE GetBinding(): List;

PROCEDURE Eval(
    env:      Environment;
    function: Text
    ): Value;
```

As part of executing the `BRIDGE` primitive, the evaluator starts a bridge process and then calls `GetBinding` in the RPC interface exported by the new bridge process. `GetBinding` returns a list of name-value pairs. A value that represents a bridge function encodes the signature of the function (i.e. the list of named parameters and whether or not the function has implicit parameters) but not the body—that's private to the bridge. Other values, such as integers and booleans used as default parameter values, are passed across in their entirety.

To call a bridge function, the evaluator calls `Eval` in the RPC interface exported by the bridge process. The function parameter to `Eval` is the function name returned by the initial call to `GetBinding`.

The evaluator exports several procedures to be called by bridges; the rest of the procedures described in this section are all exported by the evaluator. A bridge function invoked by `Eval` obtains its parameters by calling the evaluator's `Lookup` procedure:

```
PROCEDURE Lookup(
    env:  Environment,
    name: Text
    ): Value;
```

The `Environment` value is the same one the evaluator passed to `Eval`, and the `Text` value is name of the parameter the bridge is looking up.

`Lookup`'s result, of type `Value`, is a network reference that identifies a value stored in the evaluator. A `Value` can represent any Vesta language value: Boolean, Integer, List, Binding, etc. A bridge is generally expecting a specific type for each of its parameters, and wants the actual value, not a handle. So after performing a `Lookup`, the bridge calls the evaluator again to obtain a specific type of value. If the `Value` does not have the expected type, the bridge issues an error. Here are two typical examples of the type-specific procedures available to the bridge:

```
PROCEDURE IntegerV(
    v: Value
    ): INTEGER RAISES {WrongType};

PROCEDURE ListV(
    v: Value
    ): List (*of Value*) RAISES {WrongType};
```

Similarly, if the bridge needs to construct a `Value` to return as the result of a bridge function, it calls a type-specific constructor procedure in the evaluator:

```
PROCEDURE NewInteger(
    int: INTEGER
    ): Value RAISES {};
PROCEDURE NewList(
    list: List (*of Value*)
    ): Value RAISES {};
```

We chose this technique for providing remote access to Vesta values on the basis of its simplicity. The Modula-2+ RPC system does the hard work of maintaining the correspondence between handles in the bridge and values in the evaluator. The Modula-2+ RPC transport is very fast [Schroeder and Burrows], so performance is acceptable even though it takes several RPCs per parameter. With a slower RPC transport, it would be more important to minimize the amount of communication; a different remote interface to Vesta values would need to be designed.

The Vesta types Integer and List just used as examples are representative of how the evaluator interface supports most Vesta types, but the type Text is special. For efficiency reasons, the evaluator uses two representations for text (byte sequence) values. The first text representation is simply a reference to a sequence of bytes in the evaluator's memory. The second text representation is the UID of a repository object (source or derived). The second representation allows the evaluator to communicate a large text to a bridge efficiently, by sending its UID. This avoids the evaluator reading the object into memory and copying the object's contents across the process boundary using RPC. For the same reason, a bridge can return a large text to the evaluator more efficiently by passing the UID.

A bridge must be prepared for either text representation. So to read a text-valued parameter, a bridge must first query the evaluator to determine the representation, then call the appropriate procedure:

```
TYPE ValueClass =
    (TextVC, UidForTextVC, (*others*))
PROCEDURE Class(
    v: Value
    ): ValueClass RAISES {};
PROCEDURE TextV(
    v: Value
    ): Text RAISES {WrongType};
PROCEDURE UidForTextV(
    v: Value
    ): UID RAISES {WrongType};
```

Given a UID, the bridge can obtain the bytes of the repository object by calling a separate RPC interface to the repository, much like reading a Unix file using `open` and `read` system calls.

Having obtained its parameters, a bridge function is free to construct and return any type of Vesta value. A common result type is Text, used for instance to represent an object code file produced by a compiler. Since these texts are large, it makes sense to store them in the repository as derived objects.

To support a simple form of caching by bridges, derived objects are named with unique identifiers that a bridge computes. The name of a derived object produced by a bridge function encodes the function and all of the function's parameters (plus the name of the function result if the function produces several derived objects as results.) A bridge function can first compute the name of its result object and check to see if it exists. If the object already exists, there is no need to compute it again; the bridge function just converts the derived object's UID to a Value and returns it. We call this technique *repository caching* because it makes direct use of the repository as a cache.

A technical challenge in repository caching is how to encode a function and all of its parameters into a fixed-length UID. We solve this problem with fingerprints [Broder]. A *fingerprint* is a fixed-length encoding of an arbitrary byte string. Many distinct byte strings map to the same fingerprint, but such an occurrence is designed to be extremely uncommon. In the 96-bit fingerprint implementation used by Vesta, a rough estimate of the probability of collision is $n^2 * m/2^{95}$ where n is number of byte strings considered and m their average length. We chose a 96-bit fingerprint on the assumption that a function and its parameters could be encoded in 2^{10} bytes and that a repository would contain fewer than 2^{24} derived objects. This puts the probability of fingerprint collision at 2^{-37}, which is comparable to the undetected bit error rate of a typical computer network. Using longer fingerprints would allow for larger repositories.

The evaluator simplifies repository caching for bridges by exporting a procedure that hides the details of how different types of Vesta values are fingerprinted:

```
PROCEDURE ToFingerprint(
    v: Value
    ): Fingerprint RAISES {};
```

4.3 How bridges extend the set of types

Just as bridges can extend the Vesta language with additional functions, they can also extend the language with additional types. Each bridge defines exactly one new type, which is *opaque*—only that bridge may create values of its type and only that bridge's functions may operate on those values. Like lists and bindings, opaque values may contain other Vesta values. The contents of an opaque value are up to the bridge that created it. The bridge represents an opaque value to Vesta as a varying-length string of bytes and varying-length sequence of Vesta values.

Opaque bridge types offer advantages over Vesta's built-in types in some situations. Naturally these types offer the advantage of information hiding. In addition, they allow the bridge to represent its values efficiently and conveniently. For example, suppose a bridge written in Modula-2+ wanted to create values with four integer-valued components. It could use a Vesta binding with four elements, or it could define a Modula-2+ record with four fields and store the binary representation of that record in an opaque value. Not only is the opaque value's representation smaller (since it doesn't include the binding-component names) but it is notationally more convenient and more efficient for the bridge implementation to access, using Modula's record accessors rather than procedure calls to the evaluator interface described in Section 4.2. Section 6.1 discusses several applications of bridge opaque types.

To construct an opaque value, a bridge calls the Vesta evaluator:

```
PROCEDURE NewOpaque(
    value:          BridgeOpaque;
    pickledRep:     Text;
    embeddedValues: List (* of Value *)
    ): Value;
```

The value parameter is a reference to the bridge's representation of the opaque value. Its details are hidden from the evaluator—the `BridgeOpaque` type is a network reference. The `pickledRep` and `embeddedValues` parameters are a representation of the opaque value, for the evaluator to write into a persistent value cache.

When the evaluator needs to construct an opaque value from its cached representation (after a cache hit), the evaluator calls the bridge:

```
PROCEDURE Unpickle(
    pickledRep:     Text;
    embeddedValues: List (* of Value *)
    ): BridgeOpaque;
```

This mechanism is designed to be incremental; unpickling one value does not require unpickling all of its embedded values.

A bridge can effectively provide multiple opaque types by using a tagged union in its pickled representation. For example, the first byte of the pickled representation could be a tag that identifies which of the unioned types this value is, with the rest of the bytes following. Since the pickled representation can have a varying number of uninterpreted bytes and a varying number of Vesta values, providing multiple opaque types in this way doesn't waste any space.

4.3.1 Opaques versus deriveds

Opaque values themselves are intended to be rather small, no more than a few hundred bytes. To make larger values (thousands of bytes or more), the bridge can write large sequences of bytes into a derived repository object and record the UIDs of those objects in the opaque values.

Why have both bridge opaque values and derived objects? Why not have just one of the concepts?

Derived objects are much like traditional files, so we know from experience that they are a good way of representing a bulky compiler output such as object code. As discussed in Section 4.2, it is much more efficient to pass large texts by reference to a repository object than by value. So derived objects appear to be necessary.

Trying to use derived objects to represent the small data structures typically represented with bridge opaque values would be hopelessly inefficient. The bridge cannot afford to perform a random disk read per opaque value. The Vesta evaluator provides faster access to the opaque values produced by a single function call by clustering them in the persistent function cache.

A functional difference between opaque values and derived objects is that derived objects can be named and read outside of a system model's evaluation, whereas Vesta values can only be accessed during an evaluation and cannot otherwise be named. Thus a bridge that wants to record some information for later use outside of an evaluation must write that information in a derived object.

For example, a conventional debugger accesses its debugging information outside of evaluation, so all the debugging information produced by the compiler and linker must be written to derived objects. This is mildly unfortunate, since some of that information may be duplicated in bridge opaque values.

An alternative design, relying entirely on bridge opaque values, might have significant advantages. The debugger would generate an appropriate Vesta expression at the start of a debugging session and then run in the context of the expression's evaluation. This would allow the debugger to provide the user with additional information. For example, in a program containing multiple instances of some generic module, the debugger could name a particular module instance by displaying the particular invocation of the compiler in the system model that produced it. We did not implement this design because the high start-up overhead of the first version of the Vesta evaluator made the approach unattractive.

4.4 Versioning issues

The discussion to this point has suppressed some important details related to versioning.

In the list returned by `GetBinding` (Section 4.2), each bridge function is associated with a *version string*. This version string is the function's unique identifier. From Vesta's point of view, two bridge functions with the same version string are the same function.

Control of function versioning is important because of Vesta's persistent cache. Caches hold values previously computed by functions, including bridge functions. A call of one bridge function will never get a cache hit on a value computed by a different bridge function, even if all the function parameters are identical.

Suppose that today's bridge fixes a bug in some function that caused yesterday's bridge to crash (as opposed to returning an incorrect value.) If the bridge maintainer assigns the same version string to today's function as yesterday's, then a call to today's function can benefit from cache hits in cases where yesterday's bridge worked.

If today's bridge fixes a bug in some function that caused yesterday's bridge to returning an incorrect value, the bridge maintainer assigns a new version string to the fixed function. This will force cache misses: ignoring values produced by yesterday's bridge and calling today's bridge.

As with bridge functions, it's important that a new version of a bridge be able to access the opaque values produced by a previous version. We accomplish this by having each bridge export to Vesta a *bridge class*, which is just a string chosen by the bridge implementor identifying that bridge. The bridge class is returned by the GetBinding function. The Vesta evaluator labels each opaque value with the class of the bridge that produced it. If an implementor thinks a new version of a bridge is compatible with an old version with respect to its opaque types, he simply arranges for the new and old versions to export the same bridge class. This mechanism ensures that the evaluator never passes an opaque value to a bridge that isn't prepared for it.

5 Bridges for existing tools

5.1 MM: A simple bridge

The first bridge we wrote for Vesta was a bridge to our existing Modula-2+ compiler, `mm`, and its associated program construction tools. MM is a good example of a simple bridge.

5.1.1 MM$Compile function

A simplified Vesta function signature for the MM Compile function is:

```
Compile = FUNCTION
    source       (*Text*),
    Env_inst_set (*Text*),
    Env_platform (*Text*),
    ...          (*interfaces*)
                 (*returns Binding*)
```

(We include parameter and result type information as comments; recall that Vesta does not allow type declarations.)

`Compile` has three explicit parameters. `source` is the Modula-2+ source text of the module to be compiled. `Env_inst_set` is the instruction set for which the compiler should produce object code, e.g. "`VAX`" or "`Alpha`". `Env_platform` is the operating system for which the compiler should produce object code, e.g. "`Ultrix`" or "`OSF-1`".

In addition to these explicit parameters, `Compile` has implicit parameters represented by the "..." in the function signature. These implicit parameters are the compiled Modula-2+ interfaces imported by `source`. For example, if `source` says `IMPORT Xxx`, the `Compile` function obtains the value bound to the identifier `Xxx.d` in the scope where `Compile` was called. These parameters cannot be named in the signature of `Compile` because they depend upon the value of `source`.

In a typical call of `Compile`, only the source parameter is explicitly supplied:

```
MM$Compile(Hello.mod);
```

Here the source object `Hello.mod` is bound to `Compile`'s first formal parameter `source`. The `Env_inst_set` and `Env_platform` parameters default to values from the environment of the call. Values for these parameters are generally established in one place in a model and never provided explicitly in a call of `Compile` or other bridge functions.

The `Compile` function reads the module header of its source parameter to determine the module type (interface or implementation) and name. If the module header says `DEFINITION MODULE Xxx`, the `Compile` function returns a binding of the form

{ `Xxx.d` = <*compiled interface*> }

where the value <*compiled interface*> is a text containing the compiled version of the definition module. By returning the compiled interface in a binding using the "`.d`" naming convention, `Compile` makes it easy to supply the compiled interface as an implicit parameter to other `Compile` calls.

If the header says `IMPLEMENTATION MODULE Xxx`, the `Compile` function returns a binding of the form

{ `Xxx.o` = <*compiled implementation*> }

where the value <*compiled implementation*> is a text containing the compiled version of the implementation module. (The "`.o`" naming convention is inherited from Unix.)

We presented a simplified version of `Compile` above; the actual signature of `Compile` is more complicated:

```
Compile = FUNCTION
    source              (*Text*),
    MM_enable_profiling (*Boolean*),
    MM_checking         (*Boolean*),
    MM_loupe            (*Boolean*),
    MM_registers        (*Integer*),
    MM_assembly_code    (*Boolean*),
    MM_object_code      (*Boolean*),
    Env_inst_set        (*Text*),
    Env_platform        (*Text*),
    ...                 (*returns Binding*)
```

The six additional parameters control individual compiler options. For instance, if `MM_assembly_code` is `TRUE`, `MM$Compile` of `IMPLEMENTATION MODULE Xxx` returns a binding that includes

{ `Xxx.s` = <*assembly code*> }

where the value <*assembly code*> is a text containing assembly code produced by the compiler. Each of the additional parameters corresponds to a command-line switch of the `mm` compiler.

It is important to note how the `MM` bridge, like all Vesta bridges, uses Vesta's naming facilities to simplify the programming environment. In a traditional Unix environment, there are many different name spaces that a programmer must manage:

- pathnames for files containing source and derived information;
- environment variables (e.g. search paths);
- macro variables in makefiles;
- target names in makefiles;
- C preprocessor symbols;
- globally exported names in source modules (e.g. an extern declaration in C);
- linker symbols.

These name spaces overlap and interact in *ad hoc* ways, and the tools for managing the name spaces are non-uniform and lacking in functionality. All the name spaces (except for pathnames) are flat, and surprising things may happen if a program mistakenly redefines a name implicitly defined elsewhere.

Vesta's name binding and scoping provide a single yet powerful mechanism that bridges like `MM` use to replace the functionality of some of these disparate name spaces. `MM$Compile` does not use search paths to find imported interfaces; it uses Vesta naming. `MM$Compile` does not take parameters from environment variables; it uses Vesta naming. Vesta naming replaces both macro variables and target names found in makefiles. As a result, a programmer can understand and control Vesta and the `MM` bridge more easily and uniformly than the corresponding Unix tools.

Converting the `mm` compiler into an implementation of `MM$Compile` is quite straightforward.

First, change the compiler to accept all of its input parameters by calling the evaluator's `Lookup` procedure. The compiler takes some of its inputs from environment variables, some from the command line, and some by reading from the file system (locating the file to read using search paths.) All of these different mechanisms are subsumed by `Lookup`.

Second, change the way the compiler produces results. The compiler writes its results into repository derived objects rather than into the Unix file system. And the compiler constructs and returns a Vesta binding to give the derived objects sensible names.

A third, optional, change is to perform repository caching of results. The compiler computes the derived UID for one of its results by fingerprinting an S-expression containing the type of result (e.g. "`object code`" or "`assembly code`"), the identity of the function ("`compiler`"), the version number of the compiler ("`3.2`"), and all the parameters to the function (the fingerprint of the source object, the fingerprints of the imported interfaces, the fingerprinted values of compiler options). Then the compiler looks for an existing derived object with this UID in the repository. If all of the objects to be produced by the `Compile` function already exist, the compiler returns them without any more computing. If objects do not already exist, the compiler creates them, performs the compilation, and writes the desired results.

5.1.2 Other MM functions

We won't describe the other functions of the `MM` bridge in detail, but we summarize what they do:

- `Bind`: constructs a library from a list of objects.
- `Prog`: constructs an executable from a list of objects.
- `RPCStubs`: constructs RPC stub modules from a Modula-2+ interface.
- `Bundle`: constructs a Modula-2+ interface and implementation module that bundles together a collection of text values. This allows programs to link in arbitrary data, e.g. default values of program resources such as

fonts and cursors. These data values are captured in the system model, as required by the Vesta principle of complete descriptions.

The `Bind` and `Prog` functions take a parameter that is a list of object modules, exploiting Vesta's support for lists. These two functions actually accept lists of lists, to any depth of nesting, avoiding the need to flatten lists when writing models.

5.2 C: an encapsulating bridge

Often you want to build a bridge that incorporates an existing program without modifying it. This is especially useful when you have the executable to the program but don't have the source, or have the source but don't want to modify it because you want the freedom to upgrade to the next version. We call a bridge that operates in this way an *encapsulating bridge*, because it encapsulates an existing tool.

We developed a technique for writing encapsulating bridges in our environment, and used it in writing our C bridge (among others).

5.2.1 C$Compile function

We'll illustrate a general technique for writing encapsulating bridges by the example of our `C` bridge's `Compile` function, whose simplified signature is similar to that of `MM$Compile`:

```
Compile = FUNCTION
    source       (*Text*),
    Env_inst_set (*Text*),
    Env_platform (*Text*),
    ...          (*headers*)
                 (*returns Text*)
```

Our `C` bridge encapsulates several Unix C compilers. A Unix C compiler takes its parameters in three ways: through environment variables, on the command line, and through the file system. It is easy enough to control the compiler's environment variables and command line; the design challenge for the bridge is controlling the files the compiler sees.

Since the Taos operating system implementors were just down the hall, we worked out a small extension to Taos that gives Vesta the control it needs. When a Taos parent process is creating a new child process, the parent can associate a *path map procedure* with the child. Each time the child process or any of its descendants makes a system call that involves a pathname, Taos calls the parent's path map procedure, passing the pathname, the working directory, and an indication of whether or not the child's system call can modify the state of the file system. The parent's path map procedure returns a pathname for the system call to use instead of the one specified by the child.

Given this path map facility, the `C` bridge's `Compile` function works as follows. The `C` bridge starts a Unix C compiler as a child process with a path map procedure. The details of this path map procedure are elaborate, but the basic ideas are simple. The path map procedure allows the compiler to write files in its working directory (which is created by the bridge and used by one compiler at a time) and in /tmp. The path map procedure allows the compiler to read files that it has previously written in its working directory and /tmp. The path map procedure translates any other read (e.g. the read of an include file or the execution of a subprocess of the compiler) into a call of the evaluator's `Lookup` procedure. The result of this `Lookup` should be a Text value. If the text is represented by a repository object, the path map procedure translates the UID of this object into a pathname and returns it. Otherwise the path map procedure creates a new file in a private place, writes the text to the new file, and returns its pathname. When the compiler exits, the `C` bridge copies the result files in the working directory to derived objects in the Vesta repository. The `C` bridge computes a name for the derived objects much as described previously for the `MM` bridge.

The `C` bridge includes a cross-compilation capability. When `Env_inst_set` is "`R3000`" the bridge can perform R3000 compilations for both the Ultrix and OSF/1 operating systems. The cross-compilation is implemented transparently to the Vesta user in the following low-tech fashion. First the `C` bridge runs the C preprocessor locally, i.e. on the machine running the bridge, to produce a C source file with no includes. (The bridge takes care to use the default symbol definitions of the target system in doing this preprocessing.) Then the bridge selects an R3000 machine running the desired operating system. The bridge runs the C compiler for that machine, using very basic protocols: telnet-style remote execution (`rsh`) and ftp-style file transfer (`rcp`).

A `C` bridge can perform repository caching just as the `MM` bridge does. However the benefits are much less because of the unstructured nature of the C preprocessor's include facility. A Modula-2+ bridge can determine the imports of a module by reading the module header, and can quickly compute the derived object name. A `C` bridge has to do more work to determine the includes of a C source, because `include` directives can occur anywhere within a C source. It is difficult to determine the included files more efficiently than the C preprocessor does it. Therefore the `C` bridge always runs the C preprocessor. When the preprocessor exits, the bridge can compute the name for the derived object to be produced by the compiler, and short-circuit the compilation if the object exists. But when compiling C programs that use large libraries, the majority of the time can be spent in the C preprocessor reading headers. Therefore repository caching is not highly effective in speeding up a `C` bridge. Our `C` bridge did not bother with it.

The Taos path map facility was a very cost-effective way for us to implement encapsulating bridges; extending Taos took only a couple of person-days of design and implementation. We could build encapsulating bridges on other operating systems using different techniques. Here are two possibilities:

- The `C` bridge could implement an NFS server [Sandberg] and constrain the C compiler's file accesses to be within this file system using the `chroot` system call.
- The Mach 3.0 system provides a general mechanism for trapping and reinterpreting system calls in the context of a user process [Black *et al.*]. This mechanism could be used to emulate the path map facility.

Our encapsulating `C` bridge is 2500 lines of Modula-2+ code. About 2000 of these lines are common to all of our encapsulating bridges; the remaining 500 lines are specific to C. So the effort of writing a new encapsulating bridge is small.

The simple path map technique produces a function because the programs we are encapsulating are well behaved. More elaborate techniques would be required for less well behaved programs, such as a program that communicates with a server via sockets. It is possible that encapsulation could get so complicated that it would no longer be attractive.

5.2.2 Other C functions

We won't describe the other functions of the `C` bridge in detail, but we summarize what they do:

- `Preprocess`: runs the C preprocessor.
- `Prog`: constructs an executable from a list of objects.

5.3 Shell: a bridge for low-effort extensibility

Bridges are not extremely difficult to write, but are not extremely easy to write, either. While writing a bridge to incorporate a new language translator is easy enough, writing a bridge is certainly a heavyweight technique for adding a small tool to Vesta.

The `Shell` bridge is a mechanism for adding new tools to Vesta with a minimum of effort. Any tool that is easy to encapsulate (behaves like a function that takes all its parameters through environment variables, the command line, or the file system) can be run from Vesta using the `Shell` bridge.

5.3.1 Shell$Sh function

A simplified signature of the `Shell Sh` function is:

```
Sh = FUNCTION
    script       (*Text*),
    ...
                 (*returns Binding*)
```

`script` contains commands to be executed by the programmable command interpreter `sh` [Kernighan and Pike]. The file system environment in which `script` executes is quite different from the norm.

`script` executes in a virtual working directory that contains every Text- and Binding-valued name in the Vesta environment in which the `Sh` function was called. `script` sees a text-valued name in this environment as a Unix file, and sees a binding-valued name as a subdirectory of its working directory. This mapping is applied recursively, so a subdirectory contains subdirectories if the corresponding binding contains bindings. If `script` tries to read a file in its working directory that corresponds to another type of Vesta value (e.g. a number or a list) it receives a "file not found" error.

`script` is allowed to write files in its working directory and in /tmp (and in subdirectories of these directories), but nowhere else. By writing in its working directory, `script` does not alter the Vesta environment in which the `Sh` function was called. `script` is allowed to read any file it has written in a given call of the `Sh` function, obtaining the results of the earlier write.

If `script` exits with error (non-zero) status, or writes to `stderr`, the `Sh` function returns a Vesta evaluation error. Otherwise, the `Sh` function returns a binding containing a name for each file `script` wrote into its working directory, plus the special name `stdout` bound to whatever `script` wrote to its standard output stream. For example, the call

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

returns the binding

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

`script` may construct subdirectories of its working directory, which the `Sh` function returns as bindings. For example, the call

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

returns the binding

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

These examples don't show how values are given to the names `echo` and `mkdir` that occur in the scripts. It would violate Vesta's principle of complete description to take these values from /bin. Therefore our standard environment model [Levin and McJones] defines `Shell_Utils`, a binding containing a set of standard utility programs to run from shell scripts. Here are two ways of using `Shell_Utils` to complete the first example:

```
LET { Shell_Utils } IN Shell$Sh("echo abc > x")

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

The complete signature of the Shell Sh function is:

```
Sh = FUNCTION
    script                  (*Text*),
    Shell_argv              (*List of Text*),
    Shell_stdout_treatment  (*Text*),
    Shell_stderr_treatment  (*Text*),
    Shell_exit_on_error     (*Boolean*),
    Shell_verbose           (*Boolean*),
    Shell_echo              (*Boolean*),
    Shell_unset_var_is_error (*Boolean*),
    ...
                            (*returns Binding*)
```

In our oversimplified version above we described the behavior of `Sh` with all parameters but `script` bound to default values. The most interesting parameters other than `script` in most `Sh` function calls are `Shell_stdout_treatment` and `Shell_stderr_treatment`. These parameters give control over how the bridge responds to output on `stdout` and `stderr`:

value	meaning
"`ignore`"	Output is discarded.
"`feedback`"	Output is displayed in the Vesta user interface.
"`feedbackError`"	Output is displayed in the Vesta user interface and is treated as an error; that is, the call of `Sh` returns `ERROR`.
"`value`"	Output is returned as part of the result of the function, bound to the name `stdout` or `stderr` as appropriate.

The bridge defaults `stdout` to "`value`" and `stderr` to "`feedbackError`." But many programs write warning messages to `stderr` so `Shell_stderr_treatment` = "`feedback`" is a common override.

The `Sh` function is implemented by encapsulating `sh` using the same techniques described for the `C` bridge. Thus, the "virtual working directory" mentioned above is not actually constructed—it is emulated using the path map facility.

The `Shell` bridge has been used heavily in Vesta models. We have observed these common usage patterns:

- Running test programs. Some programmers construct test programs in their models, and invoke them using the Shell bridge.
- Editing source code. Many situations demand mechanical editing of source code via `sed` or a similar tool. For example, programmers overcome the limitations of preprocessors (in our environment, the RPC stub generator) by performing postprocessing of the generated source. And programmers implement low-tech generic modules by writing a template module, then instantiating it several times.
- Running small tools. The environment contains many tools that don't have bridges of their own. Two examples are a document compiler that translates a textual document description language into PostScript and a font compiler that translates a text description of a font into a binary font representation known to the window system. Programmers use the `Shell` bridge to encapsulate these programs as Vesta functions so they can be invoked from models.

In the process of converting SRC software from makefiles to Vesta, we observed an interesting phenomenon. For a given piece of functionality implemented as a shell script (or lines of `make` actions), the Vesta version using `Shell$Sh` was usually shorter by a significant amount. This was true even counting the lines of the model used to invoke the script via the `Shell` bridge. It seems that many shell scripts are really trying to compute a function. Therefore facilities supplied by `Shell$Sh`, such as automatic creation and destruction of a unique working directory, reduce the length of shell script needed.

If the `Shell` bridge is so great, why isn't it the only encapsulating bridge Vesta needs? An encapsulating bridge that's tailored to a specific task, like the `C` bridge, has two advantages.

The first advantage of a tailored bridge is control over bridge versioning. As described in Section 4.4, a bridge implementor has the freedom to produce a new bridge that, from Vesta's point of view, computes the same function as a previous version.

In practice, most changes to a bridge change the function it computes in an upward compatible manner. For instance, fixing a bug that used to crash a bridge is upward compatible, as is improving a bridge error message. Without control over bridge versioning, the Vesta evaluator cannot use a derived object produced by an old bridge as the result of a call to an equivalent new bridge.

Programmers who run small tools using the `Shell` bridge comment on this lack of control. For instance, our font compiler is a small Modula program that seldom changes. But libraries the font compiler uses often change in upward-compatible ways. Each time a library changes, the font compiler, being a tool constructed by a Vesta model, changes from Vesta's point of view. So Vesta recompiles all the fonts.

The second advantage of a tailored bridge is that it can deal more effectively with tools that aren't as well behaved as we'd like.

Many programs access files that are (1) located outside of their working directories, (2) not /tmp files, and (3) not determined by command line parameters or environment variables. The `Shell` bridge has difficulty running such programs.

As a work-around to help run some existing programs, the Sh function translates a read from an absolute pathname 'p' outside of the working directory and /tmp into a read from the tail of 'p' in the working directory. This rule is not a panacea, but deals with programs that run standard tools using absolute pathnames (such as /bin/echo.)

A significant number of Unix tools are poorly behaved in other ways, such as not setting exit status correctly or using `stderr` inconsistently. These tools are clumsy to use through the `Shell` bridge.

The writer of an encapsulating bridge like the `C` bridge can develop ad hoc solutions to deal with poorly behaved tools. For instance, a bridge can incorporate special cases in its path map procedure, knowing what pathnames to expect and how to map them. The `Shell` bridge cannot be tailored in this way.

Given that it is significantly easier to add a new tool to Vesta via the `Shell` bridge than by writing a new encapsulating bridge, it appears that we should extend Vesta and the `Shell` bridge to eliminate most objections to using the `Shell` bridge. For instance, to address the first disadvantage of the `Shell` bridge we could add a primitive Vesta function that takes a function `f` and a version number `v` as parameters and produces a new function with version number `v` that is otherwise identical to the function `f`.

5.3.2 Shell$UnsafeSh function

The `UnsafeSh` function has the same signature as the `Sh` function. The difference between the two is that `UnsafeSh` permits non-functional behavior. `UnsafeSh` permits `script` to perform reads and writes outside of its working directory and /tmp.

There are two different reasons for using `UnsafeSh`.

First, `UnsafeSh` gives you a shortcut for running a useful functional tool that for some reason isn't easy to run using `Sh`. For instance, if the `Shell_Utils` binding does not include values for `nroff` and `tmac.an`, it is tempting to write something like

```
context.1.cat =
    Shell$UnsafeSh(
        "/usr/bin/nroff -man context.1")$stdout;
```

to produce the plain text form of a manpage. This use of the `UnsafeSh` loophole isn't likely to do any damage, and allows you to proceed. Later, the keeper of your local environment can enhance `Shell_Utils`, and you can switch to `Shell$Sh`.

Second, `UnsafeSh` allows you achieve non-functional behavior when that's what you actually want. A common case is for computing a version string for an application program ("`Postcard 5.4.2 of February 13 1992`"). Another case is for copying text values from Vesta into the Unix file system.

There is a downside to violating Vesta's assumption of functional behavior. Since Vesta doesn't know that you are calling `UnsafeSh` to achieve a side-effect, Vesta may optimize away the call using its cache. We considered adding a mechanism to Vesta that would suppress caching the results of non-functional functions, but we'd rather eliminate the use of these functions. For instance, file copying can be performed by the Vesta user interface after evaluation is complete, rather than as part of evaluation.

6 Vulcan: An advanced bridge

The Vulcan bridge is the most ambitious bridge built to date. Vulcan is a Modula-2+ programming environment with these design goals:

- Explore how Vesta helps (or hinders) the design of compilers, linkers, debuggers, and other standard tools.
- Use permanently stored abstract-syntax trees (ASTs) as the basis for the compiler, debugger, and many other tools that manipulate programs within a large-scale programming environment.
- Provide fast turn-around (less than ten seconds) for small changes to very large programs written by dozens of programmers, without sacrificing execution speed or the ability to fully debug any part of the program at any time.

No commercial or large-scale prototype programming environment fully achieves Vulcan's goals. In the end, neither did Vulcan. When work ended on Vulcan we were successful in achieving the first goal (integration with Vesta), partially successful in achieving the second goal (using ASTs), but had hardly started on the third goal (fast turn-around).

The side-effect-free Vesta language and Vesta's immutable-object repository enabled Vulcan to use techniques that otherwise would be difficult to implement robustly:

- Vulcan's tools use the Vesta language's naming uniformly, presenting a simpler interface to programmers and making the linker more flexible, simpler, and faster.
- The precise structure of compiled and linked programs is represented using opaque values and ASTs, and that structure is available to other tools via the Vesta language.
- The compiler, linker, and debugger are robustly implemented as a server with in-memory caching of persistently stored ASTs.
- The programmer can always faithfully and quickly debug his entire program.

6.1 Vulcan bridge functions

The Vulcan bridge uses Vesta's naming and opaques to a greater extent than the other bridges. In Section 5.1.1 we pointed out how Vesta naming reduces the number of name spaces a programmer must manage; as we shall describe, Vulcan takes this idea farther by using Vesta naming for linking libraries and programs. Vulcan uses bridge opaques to faithfully represent the complete compile- and link-time structures of programs and libraries. It makes these structures accessible in the Vesta language via bridge functions, enabling the construction of simple tools that navigate those structures.

Vulcan's `Compile` function uses the same naming conventions as `MM$Compile`. When `Compile` is applied to a definition module (interface) named `M` that imports interface `I`:

```
Compile(
    "DEFINITION MODULE M; IMPORT I; END M." )
```

it requires an implicit argument `I.d` representing the result of compiling interface `I`; `Compile` finds `I.d` in the Vesta name scope enclosing the call to `Compile`. The compilation of interface `M` yields a one-element binding that names the compiled interface `M`:

{ `M.d` = <*compiled interface*> }

The name of the binding element, `M.d`, is generated from the identifier `M` in the module's source. The value <*compiled interface*> is a bridge opaque value with the following components:

- the name M
- a text containing the interface's source code
- a sequence of opaque values representing the imported compiled interfaces
- a text containing the interface's abstract-syntax tree (AST)

The interface's source code generally lives in a repository source object, and the interface's AST is always written to a derived object, so these two texts are almost always represented by their object UIDs.

When `Compile` is applied to an implementation module `M` with imported interface `I`:

```
Compile(
    "IMPLEMENTATION MODULE M; IMPORT I; END M." )
```

it requires two implicit arguments, `M.d` and `I.d`, representing the result of compiling interfaces `M` and `I`. The result of compilation is a binding that names the compiled implementation for `M`:

{ `M.o` = <*compiled implementation*> }

The value <*compiled implementation*> is a bridge opaque value with the following components:

- the name `M`
- a text containing the implementation's source code
- a sequence of the opaque values representing the imported compiled interfaces
- a text containing the implementation's abstract-syntax tree (AST)
- a text containing the code of the compiled implementation

The implementation's AST and code are written to two separate derived objects. The bridge opaque value packages them in a single, convenient bundle.

Now for linking. Vulcan makes a clean separation between binding and the other functions that are normally provided by a linker. *Binding* the imported interfaces of an object module is the process of identifying other modules that export interfaces matching the imports; binding produces a *bound module*. Separate functions take collections of bound modules and produce executables or libraries (shared or unshared). This separation of concepts has several advantages: It eliminates redundancy in the implementation, clarifies the semantics of linking, and provides the option of hierarchical linking (linking within software packages before linking at the library or application level) without sacrificing performance.

When `Bind` is applied to an object module `M` that imports interface `I`:

```
Bind( M.o )
```

it looks in the enclosing Vesta scope for an implicit argument `I.i`, which represents a bound implementation of `I`. The result is a binding:

{ `M.i` = <*bound implementation*> }

where `M.i` represents a fully bound implementation of the interface exported by `M`, with all of `M`'s imports bound to other bound implementations. The value <*bound implementation*> is a bridge opaque value with the following components:

- the name `M`
- the opaque value of the compiled implementation of `M`
- a sequence of the opaque values representing the imported bound implementations

(For this discussion, we simplified the description of `Bind`—the actual `Bind` is a bit more complicated. It must deal with multiple interfaces exported by a single implementation and with implementations that import each other.)

The Vulcan bridge exports many other functions; a complete description is beyond the scope of this paper.

6.2 Vulcan's opaque values

The opaque values produced by `Compile` and `Bind` serve several purposes.

First, they enable the Vulcan bridge functions to type-check their arguments reliably. Since only the Vulcan bridge can create Vulcan opaque values, the bridge functions are assured that the contents of the values are well-formed.

Second, they encapsulate information needed by `Bind` in small, efficiently accessed units. `Bind` needs only the information stored in the opaque values passed as arguments—it needn't read large derived objects containing ASTs or object code.

Third, they make it easy for Vulcan to implement utility functions such as cross-referencers that analyze the structure of programs and libraries.

Finally, they allow Vesta programmers to access the structure of compiled and bound programs from within the Vesta language. Vulcan provides a bridge function, `ToBinding`, that discloses the contents of a Vulcan opaque value as a binding containing all of the opaque's components. For example, if the value of `M.o` is a compiled implementation, then

```
ToBinding( M.o )
```

returns a binding of the form:

```
{name           = ''M'',
 source         = <a text containing M's source code>,
 imports        = <a list of opaque values representing
                  M's imported compiled interfaces>,
 AST            = <a text containing M's AST>,
 objectCode     = <a text containing M's object code>}
```

Using `ToBinding`, a Vesta programmer can write functions that navigate the structure of compiled and linked programs. For example, we wrote an "exception lint" that would check for unhandled exceptions in a program using a global data-flow analysis. Exception lint was implemented as a Vesta function that enumerated the ASTs in the program and then, using the `Shell` bridge, invoked an AST tool that actually performed the data-flow analysis.

6.3 More flexible linking

By using Vesta naming and opaque values to represent arguments and results of `Bind`, Vulcan avoids the flat name-space problems of traditional linkers. In a traditional linker, the linker symbols are derived from (but not identical to) the global identifiers in the program source. The flat name space makes it difficult to construct a program from source files that contain duplicate names. For example, a programmer might want to include two different versions of a module, or he might want to link with two libraries from different vendors that happen to have modules with the same name.

Continuing the example from Section 6.1, suppose that the programmer wants to bind the import `I` of module `M` to a bound module that's called `J`. He simply writes:

```
LET { I.i = J.i } IN Bind( M.o )
```

`Bind` looks up the value of the identifier `I.i` in its enclosing scope, and it doesn't care if that value corresponds to a source module whose name is `J`. The programmer has effectively renamed `J` within the scope of the `LET`.

Thus a programmer has complete control over which modules are bound to which, and he can easily avoid arbitrary name conflicts between modules. He uses the single, simple language- and tool-independent naming of Vesta to effect this control. (Compare this approach with that of C/Mesa, a special language for configuring collections of Mesa modules [Mitchell *et al.*].)

6.4 More efficient linking

Using opaque values and Vesta name scopes not only provides the programmer with more flexible linking, but they also make the linker simpler and faster. Vesta naming and opaques together replace the traditional linker symbol table.

All the information needed for binding is represented in the small opaque values, and Vesta provides bridges with fast access to those values. Unlike a traditional linker, Vulcan needn't read derived objects containing object code during the binding phase of linking a program—Vulcan only reads the object code at the very end of linking when it is actually producing an executable or library. Traditional linkers either make multiple passes over the object files or read all the object files into memory for random access; Vesta reads the derived objects containing object code one at a time and appends them to the executable.

6.5 Generics

Modules parameterized by their imported interfaces are easily implemented with Vesta/Vulcan. An imported interface is essentially a formal parameter that is instantiated to an actual parameter at compile/bind time.

For example, consider a list package that should be parameterized by the element type of the list. The `List` module imports an interface `Element`:

```
DEFINITION MODULE List;
IMPORT Element;
TYPE T = POINTER TO
             RECORD head: Element.T; tail: T; END;
```

The type `List.T` defines a linked list of elements of type `Element.T`.

To define a list of integers, one merely defines an interface that provides a type `T`:

```
DEFINITION MODULE Integer;
TYPE T = INTEGER;
```

and passes the `Integer` module as an actual parameter to the compilation and binding of the `List` module:

```
LET {Element.d = Integer.d} IN
    LET {Compile( List.def );
        Compile( List.mod )} IN
            LET {Element.i = Integer.i} IN
                Bind( List.o )
```

The `List` module could be instantiated with different values of `Element`. Of course, the details of compiling and binding the `List` package can be hidden from clients by encapsulating them within a function that takes `Element` as a parameter.

Generics are "free" in Vulcan, involving no change to the Modula-2+ language or change to the tools. Once we decided that there shouldn't be a flat name space of modules and that `Bind` should get its arguments from the Vesta name scope, then generics were automatically enabled. We have implemented only a few examples of generic modules, so we have only limited experience with their use, but the fact that they come for "free" is an indication of how simple and powerful Vesta naming is for bridges that choose to take full advantage of it.

We have more experience with another generics facility, Modula-3 generics [Nelson]. Generic modules are implemented within the Modula-3 language as compile time text substitution. Modula-3 generics are compatible with traditional Unix style linkers; the current Modula-3 implementation uses `ld` for linking.

Users of Modula-3 generics report that managing the flat name space of instantiated modules is a big headache. Also, `make` provides no convenient method of encapsulating the compilation of instantiated modules. These shortcomings would be avoided in a Vulcan-like implementation of Modula-3 based on Vesta.

6.6 Servers, caching, and persistent storage

Using abstract-syntax trees (ASTs) as a framework for building compilers, debuggers, and other tools offers a number of advantages to the implementor of a programming environment, but they present a major problem as well: ASTs tend to be large, an order of magnitude larger than the corresponding source or object code. Consequently, recent attempts to use ASTs have focused on storing the ASTs persistently and caching them in memory in a long-lived server providing compilation and other services [Jordan]. Compared to a traditional compiler, a compiler server that caches ASTs in memory can speed up compilation quite a bit, by a factor of two or more for interface-based languages like Modula or Ada. We expected similar speedups for other clients of ASTs.

Similarly, recent fast turn-around systems based on incremental compilation and linking use a server that retains state from past builds [Quong and Linton].

These attempts to use servers that cache persistently stored objects or retain state from past builds suffer from a lack of configuration management. In these systems the inputs to a build are stored in mutable files, so detecting what inputs have changed is either expensive (read the entire file) or unreliable (check the file's modification time). These systems may not record build dependencies on environment variables or command-line options. `make` compares file modification times to decide when an existing object file is valid for the current build; this technique can accept an invalid object file.

Using ASTs places a premium on consistency. ASTs are complicated structures with many cross links between them. Detecting inconsistencies and providing sensible diagnostics to users makes an AST implementation slower and more complicated. Similarly, the data structures needed for incremental building tend to be fairly complicated and susceptible to corruption by inconsistent building.

Vesta provides a simple, robust framework for implementing bridge servers that cache persistently stored derived objects like ASTs. Vulcan is an example of such a server. We know of no other programming environment that persistently stores ASTs and shares them correctly between users and projects.

The Vesta framework helps in several ways. Vesta ensures consistent building, and the Vesta functions provided by a bridge (e.g. `Compile`) are by definition side-effect free, computing their results only from their inputs; i.e. Vesta explicitly presents the bridge with the full compilation environment for each application of `Compile`. Derived objects in the results are uniquely named by a Vesta derived UID, and the UID encodes all the inputs that were used to compute the derived object (as described for the `MM` bridge in Section 5.1.1). Thus, an in-memory cache of derived objects can be keyed by their UIDs, enabling the server and its cache to correctly service multiple builds by different users, even if those users are building the same software package with slightly different compilation parameters.

Vulcan stores each AST as a separate derived object. A cross link from one AST to another, such as a link representing an imported interface, is represented by the AST's UID. When an AST is read into the in-memory cache, the other ASTs it references are read in only on demand when clients access particular cross links.

To reduce the size of stored ASTs, Vulcan observed that many clients access only small portions of an AST. In particular, a client such as a debugger tends to access only the top level of the AST representing the top-level names in a module, and it infrequently references the bodies of procedures. Similarly, incremental compilation of a changed procedure need access only the top-level scope of the module.

Since the procedure bodies of an AST were infrequently referenced after the initial code generation, they weren't stored in the on-disk AST representation.

When a client such as the debugger tried to access a procedure body, it would be reconstructed on the fly by recompiling just that procedure from the original source object. Truncating the ASTs like this reduced their size by factors of three or more, not only saving disk space but, more importantly, making ASTs faster to write and read.

It would have been impossible to implement truncation correctly without Vesta's guarantees of immutability and repeatable building. With Vesta, it was a simple matter to record in an AST its source object's UID and all the parameters to the compilation.

Vulcan fast turn-around based on incremental building was never fully designed or implemented. But we would have relied on the same Vesta features for persistently and correctly storing state from past builds. By definition, incremental building relies on that past state, so ensuring its exact contents is essential for correct incremental building, especially in the presence of multiple users and projects sharing software packages.

6.7 The debugger

The Vulcan debugger continued and improved upon a theme of the Systems Research Center: "You can always debug" [Redell]. On large, multi-programmer systems projects, especially those involving operating systems or long-lived servers, it's very important to be able to debug the system at any time. Bugs in these large systems are often hard to reproduce and their detection often requires collaboration of two or more programmers working on disparate parts of the system.

In traditional environments, programmers often encounter older versions of programs to which they no longer have sources, making debugging difficult. Even if they have saved the sources (say using `rcs`), they often aren't sure which sources go with the program being debugged.

Further, programmers often strip large, imported libraries of their debugging information before linking against them. Since the debugging information is several times larger than the object code, and since the debugging information is stored in the object code to ensure reliable access to it, linking with debugging information is quite a bit slower. So programmers strip the libraries, guessing that most bugs will occur in their own code and that they'll be more productive with faster turn-around instead of whole-program debugging capability. This is fine if the libraries are stable and bug-free, but in a large, multi-person project, that assumption is frequently false.

A primary requirement of the Vulcan debugger was that you can always debug all parts of your program. Since debugging was part of the turn-around cycle, we also wanted the invocation of the debugger to be fast. Thus we needed new techniques for recording debugging information.

The Vulcan debugger is a teledebugger (i.e., a cross-address-space debugger) [Redell] that lives in the Vulcan bridge process. The debugger accesses ASTs for all debugging information, and the debugger and bridge share the same in-memory cache of ASTs. Our experience shows there is a fair amount of locality of reference to ASTs not only between consecutive debugging sessions but also between the compiler and the debugger—programmers tend to debug code they've just modified and vice versa. Caching ASTs in the server across debugging sessions sped up invocation of the debugger.

Vulcan used another important technique for improving debugger performance. Per-procedure debugging information, such as maps from procedure offsets to statements and local-variable locations, tends to be voluminous. These maps often constitute more than half of all debugging information, especially with sophisticated code generators that allow variables to live in more than one register or stack location.

During normal compilation, Vulcan doesn't generate per-procedure debugging information. When the debugger needs the information for a particular procedure, it recompiles that one procedure and saves the information on the in-memory AST. Since a compiler can process several hundred source lines per second, the pauses for recompilation aren't usually noticed by the user.

Generating per-procedure information on the fly saves time, since during any one debugging session, only a small fraction of the debugging information is ever accessed. The normal compile-link cycle doesn't generate or write the information, and the debugger generates only what it needs on demand. (In a traditional environment, link time is directly proportional to the size of the object modules, so storing debugging information for every module slows down the turn-around cycle by a factor of two or more.)

When constructing a program, the Vulcan linker generates another derived object that maps program locations to their corresponding ASTs. The UID of this map is written in the program executable, allowing the debugger to access it. Vesta ensures there is no possibility of getting an incorrect map or AST for the program.

Vulcan's approach wouldn't be feasible without a robust configuration management system like Vesta. Vesta allows the debugging information (ASTs and sources) to be stored separately from the program executable. In a traditional environment, debugging information (other than source code) is stored with the executable by the compiler and linker to ensure consistency with the program; storing debugging information separately would increase the opportunities for inconsistency, and programmers are more than a little touchy about incorrect debugging information. Further, without the functional and immutable guarantees of Vesta, it would be very difficult to generate correct per-procedure debugging information on the fly.

6.8 Vulcan experience

We committed a classic mistake in planning the Vesta and Vulcan research projects. Vulcan started simultaneously with Vesta, long before Vesta was fully designed. As a result, the Vulcan implementors spent roughly half of their time working on basic Vesta issues that weren't particular to Vulcan's goals.

Vulcan spent a lot of time refining the bridge interface, particularly the treatment of bridge opaque values. Since a primary goal of Vulcan was to handle very large programming projects, Vulcan ended up converting a large amount of pre-existing Modula-2+ software into Vesta. Vulcan also explored the mechanics of developing Vesta bridges within Vesta, such as writing bridge models, bootstrapping, and cross compiling for different architectures. Finally, as the first client of Vesta, Vulcan helped debug it and analyze basic performance problems.

As a result, Vulcan never completed the design and implementation of fast turn-around. While Vulcan laid the basic framework of ASTs, fast linking, and fast debugging, Vulcan barely started on incremental compilation and did only part of the performance analysis and tuning of linking and debugging.

Vulcan built only a few prototype AST tools other than the compiler, debugger, and prettyprinter. In particular, Vulcan only started to explore how to make ASTs accessible to simple tools invoked from Vesta models.

We performed some performance measurements comparing the Vulcan bridge to the `MM` bridge. Vulcan generally cut the elapsed time for building application programs in half.

The Vulcan experience gave us confidence that the Vesta bridge interface is quite good for building a compile/link/debug server that caches persistent objects. With help from the functionality provided by Vesta, Vulcan became real enough to support dozens of users developing thousands of modules; projects exploring similar territory have stopped short of supporting users [Fyfe *et al.*] [Linton *et al.*]. Writing a caching bridge for an existing AST-based compiler would be straightforward and would take only a minuscule fraction of the time needed to write the compiler itself.

7 Final comments

We have discussed Vesta only as a configuration management system for software development. Vesta's conceptual framework is general enough to apply to other domains, such as CAD and document production. In fact, we used Vesta to manage the documentation for all Vesta-managed software. This was especially convenient for programmer's documentation that included material extracted mechanically from program text.

Section 4.1 described some implications of Vesta bridges being programs, managed by Vesta. A companion paper on experience using Vesta to solve specific problems in configuration and release management [Levin and McJones] explores the consequences of this decision more deeply.

The SEI's TCA project has recognized the need to manage configurations of tools [Dart]. Vesta bridges address many of the issues to be addressed by TCA, including configuration management, tool coexistence, tool evolution, and tool installation.

In conclusion, our experience with Vesta has established two main points about bridges. Existing tools such as compilers, linkers, and shells can be turned into bridges at low cost, gaining significant advantages. And new tools written specifically for Vesta can provide improved performance and increased functionality by taking advantage of the Vesta framework.

8 Acknowledgements

Bill Kalsow and Dave Detlefs were hardy pioneers in bridge construction, helping us to (re)design the Vesta/Vulcan interface.

References

[Black *et al.*] David L. Black, David B. Golub, Daniel P. Julin, Richard F. Rashid, Richard P. Draves, Randall W. Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. "Microkernel Operating System Architecture and Mach". In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures* (Seattle, WA, April 1992), USENIX Association, Berkeley, CA, pp. 11-30.

[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, pp. 143-152.

[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.

[Dart] Susan Dart. "Tool Configuration Assistant (Position Paper)". In *Proceedings of the Second International Workshop on Software Configuration Management* (Princeton, NJ, Oct. 1989), distributed as *ACM SIGSOFT Software Engineering Notes* 17, 7 (Nov. 1989), pp. 110-113.

[Feldman] Stuart I. Feldman. "Make—A Program for Maintaining Computer Programs". *Software—Practice and Experience* 9, 3 (March 1979), pp. 255-265.

[Fyfe *et al.*] Alastair Fyfe, Ivan Soleimanipour, and Vijay Tatkar. "Compiling from Saved State: Fast Incremental Compilation with Traditional UNIX Compilers". In *Proceedings of the Winter 1991 USENIX Conference* (Dallas, TX, Jan. 1991), USENIX Association, Berkeley, CA, pp. 161-171.

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

[Hisgen *et al.*] Andy Hisgen, Andrew Birrell, Timothy Mann, Michael Schroeder, and Garret Swart. "Availability and consistency tradeoffs in the Echo distributed file system". In *Proceedings of the Second Workshop on Workstation Operating Systems*, (Pacific Grove, CA, Sept. 1989), IEEE Computer Society Press, Los Alamitos, CA, pp. 49-54.

[Jordan] Mick Jordan. "An Extensible Programming Environment for Modula-3". *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments* (Irvine, CA, Dec. 1990), ACM, New York, pp. 66-76.

[Kernighan and Pike] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Levin and McJones] Roy Levin 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.

[Linton *et al.*] Mark A. Linton, Russell W. Quong, and Paul R. Calder. "The Design of the Allegro Programming Environment". In *Proceedings of the USENIX C++ Workshop* (Santa Fe, NM, Nov. 1987), USENIX Association, Berkeley, CA, pp. 268-273.

[McJones and Swart] Paul R. McJones and Garret F. Swart. "Evolving the Unix System Interface to Support Multithreaded Programs". *Proceedings of 1989 Winter USENIX Technical Conference* (San Diego, CA, Jan. 1989), USENIX Association, Berkeley, CA, pp. 393-404.

[Mitchell *et al.*] James G. Mitchell, William Maybury, and Richard Sweet. "Mesa Language Manual, Version 5.0". *Technical Report CSL-79-3*, Xerox Palo Alto Research Center, Palo Alto, CA, April 1979. Chapter 7 describes C/Mesa.

[Nelson] Greg Nelson (Ed.) *Systems Programming in Modula-3*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991. Section 2.5.5 describes generics.

[Quong and Linton] Russell W. Quong and Mark A. Linton. "Linking Programs Incrementally". *ACM Transactions on Programming Languages and Systems* 13, 1 (Jan. 1991), pp. 1-20.

[Redell] David Redell. "Experience with Topaz Teledebugging". *Proceedings of ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* (Madison, WI, May 1988), ACM, New York, pp. 35-44.

[Rovner] Paul R. Rovner. "Extending Modula-2 to Build Large, Integrated Systems". *IEEE Software* 3, 6 (Nov. 1986), pp. 46-57.

[Sandberg] R. Sandberg. "Sun Network Filesystem protocol specification". Technical report, Sun Microsystems, Mountain View, CA, 1985.

[Schroeder and Burrows] Michael D. Schroeder and Michael Burrows. "Performance of Firefly RPC". *ACM Transactions on Computer Systems* 8, 1 (Feb. 1990), pp. 1-17.

[Thacker *et al.*] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite Jr. "Firefly: A Multiprocessor Workstation". *IEEE Transactions on Computers* 37, 8 (Aug. 1988), pp. 909-920.

[Thomas] Ian Thomas. “Version and Configuration Management on a Software Engineering Database”. In *Proceedings of the Second International Workshop on Software Configuration Management* (Princeton, NJ, Oct. 1989), distributed as *ACM SIGSOFT Software Engineering Notes* 17, 7 (Nov. 1989), pp. 23-25.