SOFTWARE CONFIGURATION MANAGEMENT: THE VESTA APPROACH

R Levin

Rapporteur: Frode Sandnes

SOFTWARE CONFIGURATION MANAGEMENT: THE VESTA APPROACH (Extended Abstract)

Roy Levin
Digital Equipment Corporation
Systems Research Center
Palo Alto, California 94301

This paper briefly introduces the field of software configuration management (SCM), then presents an overview of the Vesta project, which addresses some long-standing SCM problems with a novel approach.

What is Software Configuration Management?

Regrettably, we can't give a crisp definition for SCM. It is an ill-defined term that covers many aspects of a software development environment, including at least the following:

- *version management:* the process of naming a series of temporally related data elements, usually files, and supporting retrieval of those data by name. The names are generally mechanically produced by some numbering scheme (sometimes linear, sometimes hierarchical), and the numbers are generally attached by appending to a root name chosen by a human. Version management applies to source material (e.g., code written by programmers) and derived material (e.g., compiler output), and may use different techniques to manage naming and storage of the two kinds of information.
- *source control:* the process of controlling the production of new versions of source data, generally files. The operations commonly associated with source control are *checkout* and *checkin*, which respectively reserve a version number for a specified use (or, more typically, user) and supply the data for a previously reserved version. Source control usually is coupled with concurrency management as well, so that the action of performing a checkout operation on a particular version limits the ability of others to perform checkout operations on other versions related in some way.
- *configuration management:* the process of defining which (versions of) software components are combined to produce larger components or entire systems. These definitions typically include instructions for performing the construction process as well.
- *building:* the process of acting upon a configuration description to produce a resulting system or component. That is, the building process is an operational realization of a configuration description.
- *life-cycle management:* an umbrella term for a variety of activities surrounding the actual production of code, including bug-tracking, quality metrics, etc.
- *process management:* the process that specifies how a particular change to a software system is to be effected, i.e., what source- and version-control steps are permitted or

required, who may or must take specific actions related to code review, quality assurance, etc., and other actions intended to provide for orderly modification of design documents, specifications, implementations, etc., of a software system.

- *specific tools:* the programs that developers use to develop and evolve parts of a software system, such as tools for front-end design, documentation, analysis, testing, compiling, and so on.

Clearly, the bounds of SCM aren't very clear. In fact, these preceding "definitions" leave plenty of scope for interpretation as well. Our purpose is only to characterize the area spanned by the term SCM, not to define it precisely.

Despite its imprecision, this list enables us to focus our attention on particular aspects of SCM. The Vesta project concentrates on the activities that fall under the first four bullets above. We consider these to be the core aspects of SCM, with other topics being of secondary technical importance. We hasten to add that all of these topics must be addressed to some extent in real-world development settings; we claim only that without a firm foundation established by the first four items, a sound structure for the remaining ones is not possible.

Because each of these areas has significance for practical software development, each has received considerable attention over the years. Regrettably, these areas have generally been pursued in isolation, or nearly so. As a result, each is rife with its own concepts, jargon, and techniques. Recently, there have been some attempts to bring solutions in these individual areas together by superficial integration techniques (e.g., a common user interface). As one might expect, such approaches haven't produced a comprehensive SCM solution.

The SCM problem

To work toward a solution, we must first have a clear idea of the problem.

For SCM, the problem is the initial construction and subsequent evolution of a software system. To be concrete, we will take the construction of a system to begin when code is first written. (This choice of starting point is more restrictive than is actually necessary, since some of the stages that often precede code production, e.g., specification, fit into our framework quite easily. Moreover, what constitutes "code" is somewhat flexible, but we take it to mean input that can be non-trivially manipulated by computer-based tools. A formal specification certainly satisfies this definition.)

When coding starts, we obviously need basic development tools — an editor, compiler, linker, debugger — and a file system to store things in. But, in most cases, we need some SCM facilities as well. (There are some exceptions: a small, single file application intended only for the author's use can probably be successfully developed without any real SCM support.) The facilities required are typically in proportion to the size of the task; simple SCM facilities work quite well for smallish systems with only one developer and a small set of friendly customers. But successful systems like this tend to "grow up," and they acquire more developers, more code, and more complexity. They outgrow the simple tools and require more comprehensive SCM. Here are some specific "growing pains" and the role that comprehensive SCM plays in allaying them:

- *Multiple files*: The observation that developers need to keep track of which file versions go together is quite obvious. Yet, amazingly, very few commercial version management systems address this need directly. The most popular ones provide virtually no help in grouping files; they are really programmer-controlled archiving systems. Comprehensive SCM keeps track of true configurations of files.
- *Multiple developers*: Modern large-system integration has reinvented the quaint charm of punched-card batch processing systems, in which jobs were submitted for overnight execution. Nowadays, developers of large software systems often are forced to submit their changes to an "integration group" whose function is to combine the changes and attempt to build the resulting system, typically overnight. This inefficient situation arises because the lack of comprehensive SCM has made it impractical for an individual developer to build and test a version of the system containing only his changes. It's either too hard to specify the environment, or the system can't be built in isolation from other developers, or both. Comprehensive SCM lets individual developers build as much of the system as they need in order to test their component, and to perform that testing without affecting the activities of their colleagues. Integration of multiple developers' work occurs only when it is functionally required.
- *Multiple customers*: A company purveying a software product often seeks to deliver it on multiple computer platforms in order to broaden the market appeal and penetration. Moreover, there may even be an ongoing market for different versions of the product on the same platform (e.g., CorelDRAW!, for which three successive releases are now sold as distinct products). Comprehensive SCM supports these marketing possibilities by providing for branched and/or parallel development of software components.

None of these situations is particularly new or unusual, so, at first blush, it might seem surprising that software development organizations put up with the lack of comprehensive SCM. Some of the blame can be put on simple short-sightedness (and a chronic underinvestment in software tools), since the traditional view of SCM is as a collection of unconnected tools whose importance in the programming environment is secondary to the editor, compiler, linker, and debugger. But the real difficulty is that comprehensive SCM, is quite difficult to implement on a scale adequate for real-world software development. As we noted above, there has been a historical tendency to attack parts of the SCM problem piecemeal, and more than twenty years of experience with that approach hasn't produced a comprehensive solution. Those who have recognized the central importance of SCM have proposed some appealing approaches, which, when implemented at all, have only worked on modest demonstration examples, making them regrettably impractical for commercial application.

Motivation for Comprehensive, Scaleable SCM

The Vesta system concentrates on the four technical areas mentioned earlier — version management, source control, configuration management, and building — providing the necessary infrastructure for tools that address other, ancillary areas. Moreover, Vesta implements facilities in these core areas on a scale that is large enough to be used in the real world.

Specifically, the design center for Vesta is a source code base of about 20 million lines, meaning that a single-version "snapshot" of the system has that size. Experience shows that a system of this size will have about 5 million derived files and consume about 125 Gbytes of disk storage. A Vesta implementation on this scale is capable of handling the development of large operating systems such as OSF1, VMS, or Windows NT, a full-featured relational database system, or a telephone switch. If not the absolute largest, such systems are certainly among the largest code bases requiring consistent construction.

One might take the view that, while some large software development projects may require such an ambitious SCM system, there aren't very many 20 million line programs. Indeed, systems of this class form a niche market, but the Vesta approach is both applicable and desirable for much smaller systems. For example, the development of even medium-sized programs, e.g., a compiler, often stresses existing SCM tools beyond their abilities. There is plentiful anecdotal evidence of the need for better tools for projects of intermediate size (say, 0.5M source-lines or smaller); one need only follow relevant Internet bulletin boards for a week or so to amass tens of requests for Vesta-like facilities.

Before surveying those facilities, let's look at the problems that an SCM system must handle in real-world development.

- *Escalating functionality.* Software systems keep getting bigger and more complicated. Vendors add functionality based on their perception of what their customers want and what they expect/fear their competitors will provide. One need only look at the long lists of "check-off" features on word-processors or spreadsheets or other mass-market software to see this effect.
- *Parallel development.* Not only is functionality increasing, but vendors feel pressured to get it to market faster and faster. This leads to parallel development and overlapping release cycles, in which work on version N+1 begins while version N is still underway, and while bug fixes for version N-1 are still being developed and shipped.
- *Broad integration.* Fewer and fewer applications and systems are stand-alone. Instead, more system layers or components have to be built together consistently. What constitutes a "system" for configuration management purposes may include a suite of applications, a set of libraries that they share, and perhaps even an underlying operating system.
- *Multiple platforms.* To reach the most customers, vendors must make their software run on multiple platforms, with differing hardware, operating systems, networks, or perhaps all of these.
- *Geographically dispersed development.* Big systems are increasingly developed at multiple sites. Because of wide-area bandwidth limitations, a central site storing all the files is impractical. Replication becomes necessary, and with it comes the risk of inconsistencies in the replicas.

It is evident that most of these problems are non-existent in small systems and grow inexorably with system size. A practical SCM approach must deliver a manageable solution for medium-sized systems and scale naturally as the system grows.

The Vesta Axiom

Most of the key design choices in the Vesta system follow from a single assumption, which might reasonably be called the "Vesta axiom," which is:

> *Complete, modular, source-based system descriptions are an essential foundation for configuration management.*

We need to examine the meaning of these terms a bit more closely and understand some of their immediate consequences.

- *System descriptions are source-based.* Vesta makes a precise distinction between source and derived files. Derived files can be mechanically generated or regenerated by a Vesta system when they are needed, while source files cannot. Source files are usually handmade by humans, but may also be programs or data produced remotely that cannot be regenerated by local means. For example, a distribution of an operating system is viewed as a collection of derived files at the site that produces it, but is generally treated as source by its purchasers, who lack the means to construct it.
- *System descriptions include source files and building instructions.* System descriptions explain exactly how to build a system. In principle, Vesta builds every system from scratch. To be practical, of course, it is essential that Vesta reuse previous results (i.e., derived files), although formally these are just optimizations.
- *System descriptions are self-contained.* System descriptions tell the entire story of system construction, capturing every relevant detail of the environment. Of course, every source file is explicitly mentioned. Moreover, every version of every tool and every switch and option and flag is specified. These details are in terms of source, so the tools are specified not as executable programs but by giving the system descriptions that construct them. (Clearly this is a recursive process. Of course, the descriptions really don't go all the way back to the Big Bang — there must be a practical limit — but the basis for the recursive description of tools is chosen by the users of the Vesta system; it is not an implementation limitation.)
- *System descriptions are immutable and immortal.* System descriptions, and therefore the source files they reference, cannot be changed once created. Because the descriptions are complete, they retain the same meaning forever, and a system build using those descriptions has identical results (i.e., the same derived files) whenever it is performed. Furthermore, the immortality of system descriptions assures that a system build can always be performed, because no piece of it is ever lost. (One might believe that there are pragmatic limits here, too, since it would be too expensive to retain all versions of all source files forever. Limits do indeed exist, but they are far more liberal than those typically used or imposed in conventional version management systems. The economics of modern disk systems make it practical to retain essentially all versions of source files indefinitely.)

- *System descriptions are modular.* To be manageable, the description of a large system cannot be monolithic; it must be composed from smaller ones. Thus system descriptions and the language in which they are expressed must be designed to support composition, meaning that information necessary for consistent building must be easily transmitted between system description modules.

The Vesta system descriptions that are characterized in this way are called *system models,* a term which henceforth we use interchangeably with system descriptions.

Let's now look at the Vesta core facilities that implement the consequences of this axiom.

Vesta's Core Facilities

Figure 1 shows a block diagram of Vesta, showing its logical components. This isn't an implementation diagram; we'll get to that a little later. This diagram introduces the functional components of Vesta and their overall relationships.

Clearly we need a place to store things: the source files and system models that go into a system and the derived files that are constructed by the building process. The Vesta *repository* provides that storage facility. It does so in a way that integrates closely with the file system; roughly speaking, it hides under the file system abstraction, although, as the picture suggests, a small amount of it peeks out. (The *replicator* is also closely related to the repository; we'll discuss it later.)

We also need an engine for the construction process. This is the interpreter of system models, which we also call the *builder.* It is the builder's job to perform the construction process incrementally, reusing suitable pieces of previous builds. It needs to call on particular tools to produce derived files; these tools are compilers, linkers, macro processors, etc. We call them *bridges.* It is convenient to think of these bridges as

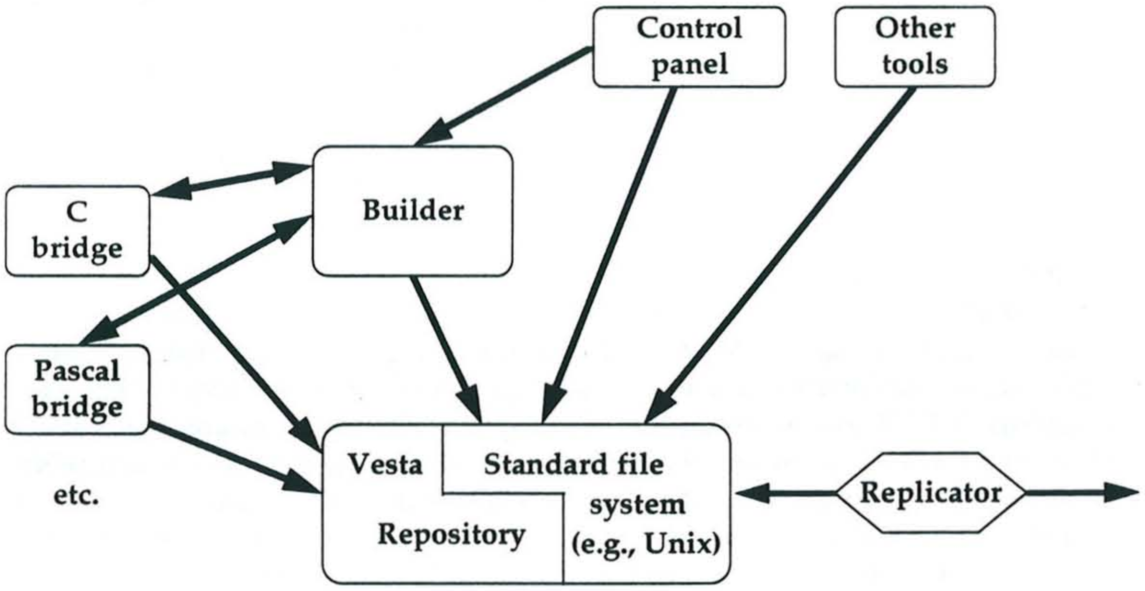

Figure 1: Vesta Functional Diagram

grouping the tools associated with a particular programming language, although this grouping is only a convention.

Finally, we need a way to communicate with these core facilities. The *control panel* and ordinary shell commands provide the necessary human interface.

Let's now look at these facilities in more detail.

The Repository

The Vesta repository appears to the user as a file system extension, meaning that the repository provides access to the files it stores through the ordinary file system interface. This has the attractive property that files stored by Vesta are accessible by ordinary applications that don't know anything about the repository, and is in contrast with the approach taken by "add-on" versioning facilities like SCCS or RCS, which store versioned files in a separate name space inaccessible to ordinary programs. The Vesta repository encodes versioning information in ordinary file names, so it remains visible to the user. This is in contrast with other SCM systems, e.g., Atria's ClearCase, which generally hide the versioned names from users and establish a "view" that allows the user to access a single version at a time through an unversioned name. In such a scheme, the meaning of a unversioned name varies with time and with the user or program that utters it, while the Vesta repository traffics only in versioned names, whose meaning never changes.

Although the repository goes to considerable lengths to make its facilities available through the ordinary file system interface, it cannot provide all its functionality in this way. Consequently, a secondary interface is provided for tools that exploit this functionality. We'll see an example of this functionality, appendable directories, shortly.

The name space of a Vesta repository thus consists of a directory tree, just like a regular file system. However, the directories behave somewhat differently. Vesta actually supports three kinds of directories: immutable, appendable, and mutable. To explain the purpose of these directories, it's convenient to introduce the notion of a package.

A *package* is simply a group of files that, for the purposes of system construction, are useful to keep together. Typically, the group is checked out of and into the repository as a unit, and thus logically worked on by a single developer. A package generally requires some sort of construction in order to be part of a system, and hence has a system model as one of its files.

An *immutable directory* generally holds a version of a package. The version may be archival, or may be a snapshot of a state during a development session. Either way, immutability applies; the directory is created all at once, and cannot be subsequently altered.

A Vesta repository consisting entirely of immutable directories wouldn't be very useful. A Vesta repository also has *appendable directories*, which are generally used to build up a suitable naming hierarchy. Typically an appendable directory is used to group all the versions of a package, which are then collectively called the *package family*. Because the family directory is appendable, new versions may be created and added, but old versions may not be deleted. The structure of a package family may be linear, with a single directory holding versions 1, 2, 3, ... of the package, or it may be more tree-like, with appendable subdirectories that represent separate lines of development, e.g., different major releases, or other axes of variation. Whatever the structure, the immutable directories at the leaves of the tree hold the individual package versions.

Vesta must provide a way to build up incrementally the files that are placed in an immutable directory. For this purpose, the repository provides ordinary *mutable* directories. In fact, there is no logical need for the repository to implement mutable directories — those provided by the ordinary file system are perfectly adequate — but there are some performance advantages to be gained in doing so.

The repository permits labeling of files and directories with *attributes*, which are mutable. Attributes are convenient for a variety of tools that are not directly involved in the building process. For example, an attribute might characterize the extent to which quality-assurance testing has been performed on a particular package version. Attributes can also be a convenient communication mechanism between users and tools that run on their behalf. For example, consider a tool that builds an updated system model by revising references to packages that satisfy some predicate on attributes. Such a tool might take as input a request like the following: "Make a system model in which all the package family references are the same as in Standard_Environment version 12, but whose versions have a Code_Reviewed attribute of "yes" and a Checked_In_Date attribute later than last Thursday." Since attributes are not used by the Vesta builder, their mutability cannot comprise reproducible construction.

These facilities (directories and attributes) give users considerable flexibility in structuring their system components, while providing the necessary guarantees for incremental, reliable building (discussed in the next section). But the physical arrangement of files is of importance too, since it may be impractical to place all the users of a Vesta repository in close physical proximity. The bandwidth available between sites will typically be significantly lower than within a site, so that cross-site references are unappealing. For this reason, the Vesta repository supports selective copying between sites. The unit of replication is the directory (only appendable and immutable ones), meaning that all the names at one site are replicated at another. However, Vesta doesn't necessarily copy all the values associated with those names; selected files (or subdirectories) are copied, and the other names are marked at the destination site as "stubbed off'. Selective replication is often desirable because one site uses the end-products of another, so intermediate derived files need not be replicated. Of course, this improves latency of replication too.

Vesta's replication scheme doesn't imply any particular time-grain for the replication, and different sites may choose different techniques according to their needs. At one end of the spectrum is replication-on-demand; at the other, magnetic tapes and the postal service. An interesting intermediate point is the "siphon", a background task that replicates packages between sites according to a set of standing orders from the relevant system administrators.

One final aspect of the repository deserves brief mention: the management of file storage. Immutability can only be achieved asymptotically, since disk storage at a site is finite and can be filled faster than new disks can be acquired. In practice, the chief concern is derived files, whose size dwarfs the source files. Because, by definition, these files can be mechanically reproduced by the Vesta builder, their retention is a space-time tradeoff. Vesta employs a background task called the "weeder" that selectively reverses the trade-off by deleting derived files according to standing orders from the users and the administrators. Thus, derived files are retained not out of functional necessity, but on a space-available basis.

The Builder

The essential property of the Vesta builder is the *reliable, incremental* construction of systems, which follows from the complete, immutable system descriptions promised by the Vesta axiom. This property doesn't follow trivially by any means; a practical implementation of the builder is the central challenge of the Vesta project.

We should understand why more conventional builders, typified by UNIX 'make', aren't adequate. When we look carefully at the way 'make' is used to build substantial systems, we discover that developers simply can't trust its incremental technique to get the right answer. They only use it to build "from scratch", which makes *in vivo* testing of individual components by their developers impractical. System building is entrusted to a special "integration group", long turnaround cycles ensue, and productivity goes rapidly downhill.

We can identify several reasons for 'make's inability to cope with substantial systems:

- A dependency recorded by 'make' corresponds to the use of a file with a particular timestamp in the course of building a given target (derived) file. 'Make' compares the timestamp of the existing target file with the timestamps of all the files on which it depends, and rebuilds the target if it is older than any of its dependencies. This simple test makes no allowance for the multiple versions of files that occur during parallel development. Consequently, when parallel development streams are merged, 'make' may erroneously judge a target as up-to-date when in fact it requires rebuilding. This situation arises frequently in multi-person development projects and is a primary reason for mistrusting 'make's incremental building .
- The 'make' notion of dependency is incomplete, since it reflects only dependencies on files. Some versions of 'make' recognize other dependencies, e.g., command line switches, but none implement practical means for expressing dependencies on (versions of) tools.
- Checking a timestamp requires a file system operation, which, in systems involving hundreds of files and thousands of dependencies, limits turnaround cycle performance. It is not unusual in such systems, following a change to one source file, for 'make' to take more time to determine what to rebuild than to recompile the file in question.
- The language used to express building actions to 'make' does not support any notion of modular composition or hierarchical description. It is these facilities that permit dependencies to be checked at the component level. Since they are absent in 'make', dependencies have to be checked at the individual file level, which can be quite time-

consuming in a system containing tens of components and hundreds of files. Once again, this produces poor building performance.

Actually, we shouldn't be surprised that 'make' exhibits these deficiencies. It was never designed for the classes of systems to which it is routinely (mis)applied today. To be sure, there are evolutionary descendants of the original 'make' that adequately address some of these problems. None of them, however, eliminate the performance problems or permit reasonable modular decomposition of large system descriptions.

The inescapable conclusion is that 'make' is too limited for large-scale system building. It just doesn't have a scaleable way of describing systems that permits reliable incremental construction. A better form of description is necessary.

Vesta System Models and the Description Language

As we've seen, system descriptions must be complete and immutable in order to be practical for real-world development. But the structure of system models is far from determined by these considerations alone. How the models are structured depends on the size and scope of the system being described, as well as the organization that builds it and the software methodology they employ. Organizations should be able to control things like the structure of libraries and the interfaces they provide; what's in a package; what constitutes a test of a package and when and how it is run; the relationship of documentation to the code it references; and a myriad of similar considerations.

It would be unwise, therefore, to "hard-wire" decisions of this sort into the description language. Instead, Vesta adopts a very basic language with only broad, generic facilities whose chief purpose is to support tailorable extensions. This is certainly not a novel technique; many programming systems, starting with Lisp, have used it (though admittedly for different reasons). Just as only a few hardy souls program in pure Lisp, only a few people use Vesta's base language directly. Most Lisp users program in an *extension*, containing syntactic extensions to the base language and a comprehensive collection of library functions. Similarly, in Vesta, most users write system models with the *standard extension,* which offers conveniently packaged facilities for utilizing bridges, and enables system models to be written in a largely declarative form. Moreover, if the standard extension proves inadequate in some situation, it can be tailored to the specific need or, if the organization prefers, entirely replaced, since it is simply a program written in Vesta's base language.

Let's turn, then, to the base language itself. It is a small functional programming language based directly on the λ-calculus. A functional language is well-matched to Vesta's incremental building approach, since the semantics readily permit results of function applications to be cached with dependencies directly derived from the arguments of the application. This caching is at the core of Vesta's incremental building; a "hit" in the function application cache permits a portion of a system build to be bypassed by substituting the cached result. The precise representation of dependencies in the cache enables reliable construction. Of course, significant engineering is required to implement this cache on a scale that works for systems whose models contain thousands of function applications.

The base language is in the spirit of Lisp, but with a less spartan syntax. Functions (actually, closures) are first-class entities in the language. The language has "readable" static scoping. Values are typed, but checking is not static. (There is no point in static checking, since "runtime" for the Vesta language is construction time of the system being described, and thus type errors in system models will manifest themselves as failed construction, though possibly with less-than-obvious diagnostics.)

The language's syntax and primitive operations emphasize the manipulation of composite values, typically lists or *bindings* (i.e., sets of <name, value> pairs). Values of this sort arise very frequently in configuration descriptions, for example, interface definitions, implementation modules, libraries, compiler options, package versions, etc.

Vesta's modular descriptions are really built by tasteful use of functions. This allows the construction of packages to be described independently and precisely parameterized, permitting them to be composed in a well-defined way. It is careful parameterization that permits separation of the environment from the package description. The Vesta approach is in sharp contrast to typical 'make' usage, in which many references to the environment are implicit and difficult to spot by reading the descriptions.

The base language is methodology-neutral; it does not embody any "name brand" programming methodology. It contains no operations that are peculiar to software construction, which leaves to the standard (or other) extension all decisions about the structure of libraries, applications, releases, and the like. Specific building steps, such as compilation, are simply represented as functions supplied by bridges included in the extension. Therefore, the base language is also insensitive to the programming language or languages chosen for implementation of the system being described.

The Standard Extension

Let's turn now from the base language to the standard extension that most users see as the actual description language. We want it to be very easy to say ordinary things, but allow considerable flexibility for saying less common things, like "compile this module with these non-standard switches", or "override the standard string library with this one". The guiding principle is that simple things should be simple, while complex things should nevertheless be possible.

The standard extension provides a comprehensive set of tools (that is, bridges) for programming in a number of common languages, plus a *consistent* set of libraries and the interfaces for using them. It thus provides a comfortable set of facilities that we expect to find in a well-equipped program development environment. The standard extension, or environment, is defined by a Vesta system model, which means it identifies precisely and immutably the versions of the tools, libraries, interfaces, etc., that it includes. As long as a user's system model refers to a particular version of the environment, all of the environmental facilities it uses remain unchanged. Thus, the individual user is in complete control of his programming environment; new facilities appear only as a result of the conscious act of using a different version of the standard environment.

As we noted earlier, the standard environment encourages the user to write his system models in a largely declarative form. By this we mean that, roughly speaking, the environment provides some functions that perform common construction actions, such as building an application program or a library, and the user's model simply specifies the function to be invoked and declares the parameters, which are normally lists of files. Many common situations are handled by invoking a single construction function in this way, so simple things are indeed simple. Figure 2 illustrates one such case.

Without analyzing this example in detail, we can readily see that there is a single function application that builds a C program. It accepts four parameters: the name of the program (`Hello`), a list of header files specific to the program (here, just `hello.h`), a list of source code files (two in this case), and a list of libraries from the standard environment (here, the single library `c-lib`). The function `Program` compiles the source code files having established an environment containing `hello.h` as well as the header files needed to use the functions in the `c-lib` library. The results of these compilations are then linked together with the object code of the library to produce the executable program `Hello`. Notice, therefore, that a library is not just a pile of code. Rather, it is a composite object that includes both interfaces and implementations. This is a methodological choice provided by the standard extension, and some organizations might prefer to arrange things differently.

The `Program` function is an example of a *wrapper,* since it "wraps up" the functionality of two basic tools (compiler and linker) in a single convenient bundle. Such wrappers are naturally a part of language-specific bridges, in this case, the C bridge. Since the C bridge is part of the standard environment, the user's system model isn't cluttered with the details of individual compiling and linking steps.

Before leaving this example, we should comment on its apparent violation of the completeness requirement of the Vesta axiom. The system model in figure 2 makes no reference to the version of the environment it is using, which we argued earlier was an essential property of the Vesta approach to system descriptions. This is because the builder isn't asked to evaluate this system model *in vacuo*. Rather, it evaluates a quite trivial model that does nothing more than reference this model and a version of the standard environment, then specify that the former is to be evaluated in the context of the latter. The system model of figure 2 is actually an implicit function parameterized by the environment, and the "$" preceding the name `c-lib` is some syntactic sugar that refers to that environment. The user controls the contents of the model presented to the builder, and therefore specifies what version of the environment is to be used. This is typically

```
DIRECTORY
  hs = [ hello.h ];
  cs = [ hello.c, hellosub.c ]
LET
  name = "Hello";
  libs = [ $c-lib ];
  prog = $C$Program( name, hs, cs, libs)
IN
  { !name, !prog }
```

Figure 2: System model for "Hello, world" program

done through the Vesta control panel, which we discuss in the next section.

We've seen how the standard environment makes it easy for a user to perform simple building operations. The standard extension also accommodates some complex things. For example, it is possible to specify, on a file-by-file basis, that non-default compilation options are to be used. This is accomplished by allowing the parameter that specifies the list of source file inputs to be more elaborate than a simple list; it may be a list of bindings (that is, in Lisp terms, a list of association lists). The standard extension also allows for selective invocation of pre- and post-processors surrounding the compilation step, permitting the inputs to be more general than simply source programs in a single programming language. There are other facilities as well, but the point should be clear: complex things are possible.

Nevertheless, not *all* complex things are possible within the standard extension. For example, there are only limited facilities for building an executable program whose pieces are written in several programming languages. Most organizations don't require this facility, which is why the standard extension doesn't attempt to provide it. But, unquestionably, it is important in some situations. It is for that reason that the standard extension is just a system model — it can be modified as necessary to meet a particular organization's requirements. It can even be entirely replaced with a different environment that more directly supports the organization's system structuring preferences and needs. The goal of the standard extension is to spare most organizations the work of constructing their own environment when a tried-and-true one is satisfactory.

The Control Panel

The Vesta control panel provides a convenient graphical user interface for automating certain aspects of producing system models and invoking the builder. Most system models are simply text files that users create with an ordinary text editor. Sometimes, however, specialized tools can streamline this process, and in these situations it is helpful to have a more appropriate user interface than a text editor can provide.

We already saw one example of this situation in the previous section. By convention, the system model that is typically presented to the builder is quite stereotyped; it says, in effect: "evaluate the most recent version of my package in this version of the standard environment". There are only two pieces of information here: the two versioned names. Typically, a user generates a series of versions of a package as he incorporates and debugs some new feature, and it is convenient to have the control panel keep track of the next version to be assigned. Moreover, the version of the standard environment used throughout such a development session typically remains unchanged, so the user can fill in a field on the control panel with this information at the start of the session. Thus, at the push of a button, the control panel can generate the desired system model and invoke the builder.

We encountered another example of specialized model editing in conjunction with the repository's attribute mechanism. The control panel can provide a convenient user interface for specifying a predicate on attribute values that is used to generate a model referencing selected versions of other packages. Of course, a text editor could be used for this purpose, but the control panel interface streamlines an otherwise tedious manual procedure in a situation that occurs frequently in hierarchically organized system descriptions.

Specialty Tools

A comprehensive programming environment typically includes many tools to assist developers with various aspects of software production, and a number of these tools benefit from Vesta's core facilities. For example, the semantics of the Vesta repository's directories and versioning make it easy to produce helpful "differencers" for combining parallel development branches and resolving conflicting source code modifications. We won't attempt to enumerate the many possible tools that can exploit Vesta's storage and construction semantics. However, we should comment on tools whose *raison d'être* is Vesta's unique form of system description.

Vesta's system description language presents an "entry barrier" for users who wish to move an existing system that uses 'make' into Vesta. Consequently, Vesta includes two tools to make this transition easier by permitting the system to be converted a piece at a time, with some makefiles being rewritten as Vesta system models, and others remaining unchanged for a while.

The first tool is an *interpreter* for the input that 'make' expects ("makefiles"). An organization can choose the portions of its software system that will first be converted to Vesta and produce system models for them. The other portions continue to be described by makefiles, and the Vesta-supplied interpreter is invoked at the appropriate point in the system model(s) to build the unconverted components in the old way. This interpreter acts, in effect, as a Vesta function whose input is the standard environment and a makefile, and restricts all file accesses from the makefile to the environment. While it doesn't provide the incrementality of 'make' (which, as we've noted, is often suspect anyway), it does permit a component described with a makefile to be built reproducibly as part of a larger, Vesta-based system build.

The second tool is a *translation assistant* that helps convert makefiles to Vesta descriptions. Regrettably, because make files have relatively little useful static structure, the tool can't translate them automatically to intelligible Vesta models that users would want to evolve subsequently. However, the tool can produce a good approximation of such a model, doing most of the mechanical work and leaving only a small amount for the user to do by hand.

Using these tools, an organization can partially convert a system for development in Vesta and leave it that way indefinitely, continuing to use makefiles and the Vesta makefile interpreter for selected components. Because the interpreter doesn't perform incremental construction of those components, such a structure won't yield the full benefits of Vesta. However, if those pieces don't change frequently, the tradeoff might be attractive. Moreover, if the unconverted components must be shared with other organizations that don't use Vesta, it may be quite helpful to retain their building descriptions as makefiles.

Implementation Overview

Having surveyed the key components of the Vesta system, let's now take a brief look at the structure of the implementation. Space prevents anything more than a cursory examination; our goal is to understand where the pieces of functionality reside and why they might plausibly scale to handle large systems.

Figure 3 illustrates the implementation of the repository. The implementation is divided into a *clerk,* which exists in each client of the repository, and a *server*, which resides somewhere on the local area network. There are also file servers that implement the regular file system, which may also be accessed through clerk code on the client workstations (the diagram omits this detail).

Most file accesses to the Vesta repository, including open/read/write/close operations, are handled entirely by the clerk code, which, in turn, accesses the appropriate file server(s). That is, most operations on repository files can be translated into ordinary file operations without accessing shared, global state. Hence, these requests don't involve the Vesta repository server, ensuring that it doesn't become a bottleneck.

The repository server does become involved in operations that inherently require serialization of user activities. For example, the checkout operation reserves a version name for later checkin by a designated user, and some coordination is required to ensure that the same name is not given out to multiple requesters. While it is certainly possible to implement this logically centralized function in a distributed way, the cost and complexity are much greater than a physically centralized implementation. Since checkout is relatively infrequent, the simpler technique can handle the load without strain.

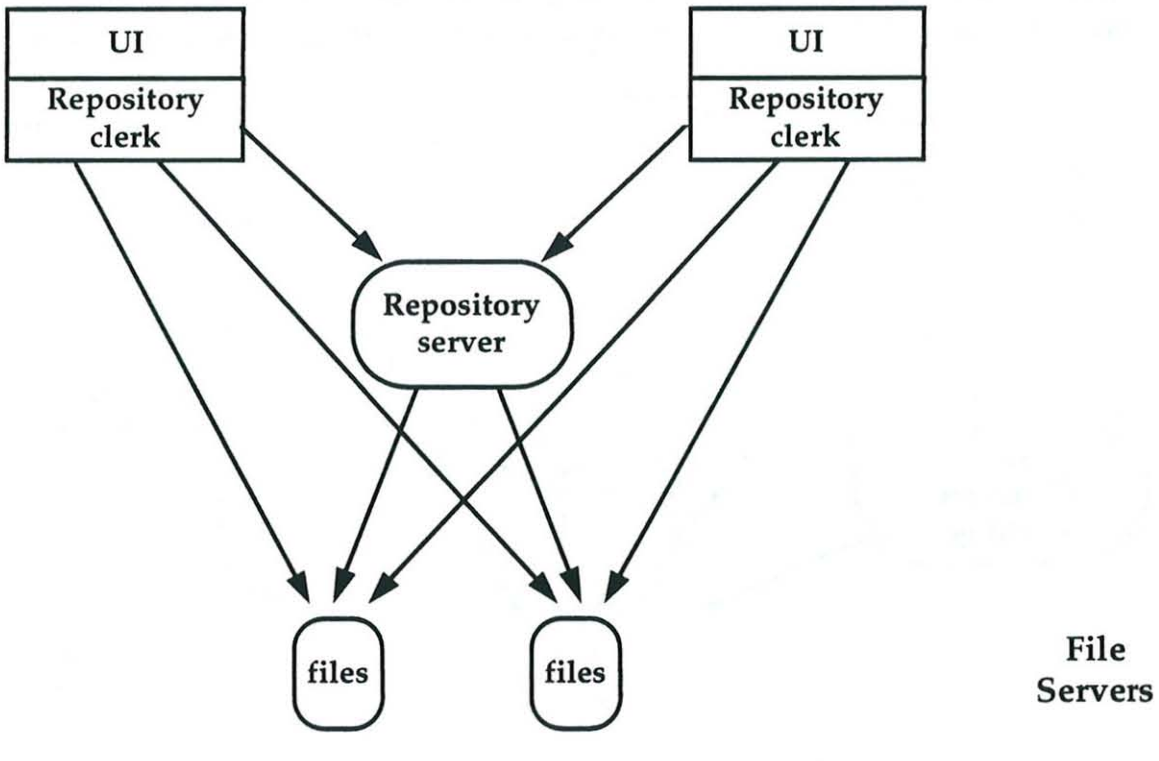

Figure 3: Vesta Implementation: Repository

Figure 4 shows three possible ways to run the builder. It is possible, although unlikely, that all of these would be used simultaneously at the same site. As in figure 3, workstations appear at the top of the diagram and file servers are at the bottom. In between are various servers that may be participate in a Vesta building operation.

At the upper left is a workstation on which the builder is executing. The builder makes use of the Vesta *function caching service*, which is the large oval at the left. It is implemented as a distributed service, as suggested by the nested ovals. This service provides a persistent store for the results of function applications in the Vesta description language, enabling the builder to work incrementally. The function cache is physically partitioned across multiple servers, and the builder presents its individual cache lookup/enter requests to the appropriate server. The configuration of servers implementing the cache can be changed while a builder is running and can perform dynamic load balancing in response to a persistently uneven pattern of requests. If a server fails, its load can generally be absorbed transparently by other servers. Consequently, the partitioned implementation of the function caching service provides natural scaling without reducing availability.

Turning from the function caching service to the other components in the figure, we see that the builder at the upper left is executing on the workstation of the user who invoked it, as are the bridges needed by the building operation. The only remote facilities required are the regular file system and the function cache. This is a common arrangement when the user has a reasonably powerful workstation and the bridges that are necessary to build the system under development can execute on the workstation platform.

Looking now at the second workstation, we see that the bridges are being executed remotely. This might be necessary if the system under construction is being compiled for a

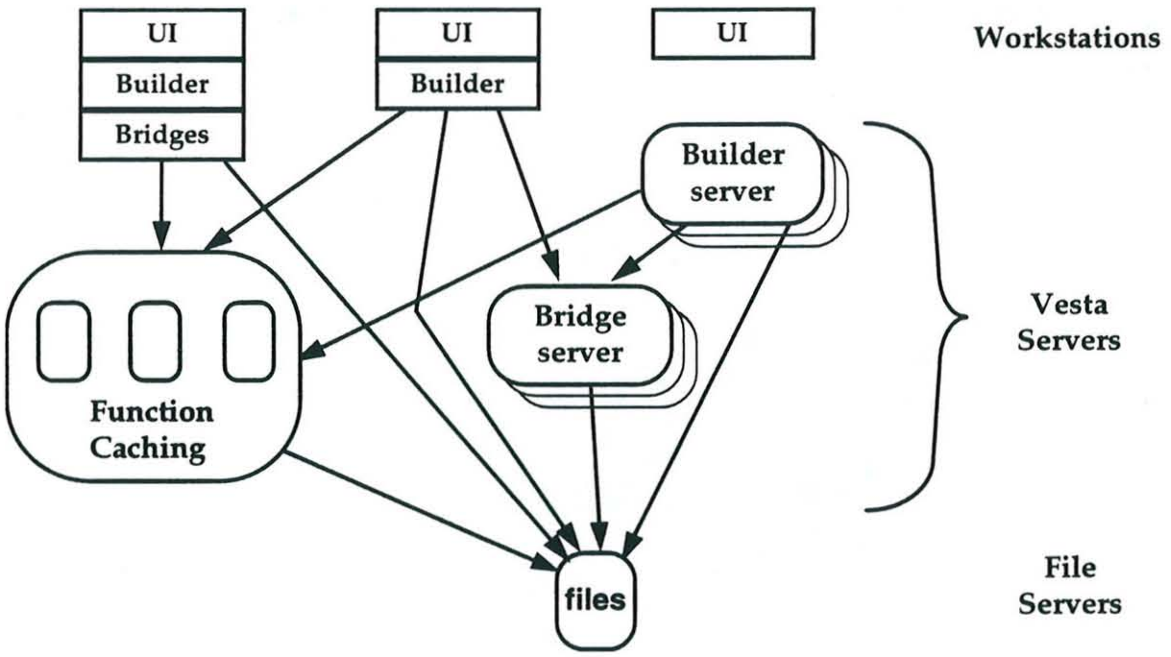

Figure 4: Vesta Implementation: Building

different kind of platform than the workstation performing the build, and cross-compilers that can execute on the workstation aren't available. The Vesta implementation provides a simple service so that the builder can run bridge components (tools) as necessary on appropriate server machines.

For the build being performed at the upper right, the workstation is adequate to run only the user interface, perhaps because of limited functionality or performance. In this case, the builder as well as the bridges are run remotely on appropriate servers.

Project Status and Conclusion

The preceding description of the Vesta system is written in the present tense, suggesting that all of it actually exists. That's a slight overstatement — let's correct it now.

Most of the functionality described above was implemented in a prototype system and deployed in December, 1990. It was used in earnest for about 15 months by a group of 25 programmers who were developing a sophisticated experimental software environment. This project included: a custom operating system; a file server; a novel compiling system; an experimental windowing system; a number of mathematical, graphical, and symbol-manipulation libraries; and a spectrum of applications, including a text editor, a graphical editor, ray-tracing software, and some mathematical analysis tools. Collectively, this code base comprised 1.4 million source lines. Most of the components were built with the experimental compiling system (which was continuously evolving), and were targeted for two different platforms: a VAX multiprocessor with the custom operating system, and a MIPS uniprocessor running OSF1. In short, this system provided a serious test of the Vesta approach on a scale comparable to many real-world development projects.

The results conclusively demonstrated that the approach is workable and that the Vesta axiom can be practically implemented on this scale. (These results are reported in four research reports available from the author.)

The Vesta prototype implementation did not support all of the functionality mentioned earlier. Most notably, it lacked facilities for wide-area replication and could not scale to handle a system of 20 million source lines. A second implementation that remedies these deficiencies is underway.

In conclusion, the SCM problem is significant and timely for real-world systems of all sizes. No comprehensive solution is presently available commercially. The Vesta approach, which follows from a novel axiom, offers a comprehensive solution whose practicality has been validated by a substantial prototype. A forthcoming implementation will provide this functionality for systems of essentially any size.

DISCUSSION

Rapporteur: Frode Sandnes

Lecture One

Professor Kopetz asked if Vesta can monitor hardware changes. Dr Levin replied that Vesta can be used to monitor hardware changes to a certain degree. In particular it has been used on systems to monitor microcode. There are restrictions, hence Vesta can only check whether the software is running on the correct hardware.

Dr Dicker wondered if Vesta has got an API (applications programming interface). Dr Levin stated that it has.

Lecture Two

Professor Randell pointed out that there is a need for mutual attributes. Dr Lewin indicated that such facilities are still evolving.

Dr Lewin announced that the nature of the official release of the Vesta software is yet unknown. It is either to be based on licencing or a public domain version.

Dr Levin added the comment that inefficiencies in some compilers makes it necessary to patch the object code by hand prior to linking, and Vesta is fully able to track such actions.

Professor Randell asked if Vesta could be used for non-programming purposes, for example building documents. Dr Lewin replied that Vesta has been used for
a range of other purposes similar to the make utility.