Staying Afloat in a Sea of Versions

or

Software Configuration Management: The Vesta Approach

Roy Levin
Allan Heydon
Tim Mann
Yuan Yu

Digital Equipment Corporation
Systems Research Center

April 22, 1996

This talk will address a particular but pervasive aspect of systems: how we build them and keep track of what we have.

In the jargon of the field, the problem I want to discuss goes under the name of "software configuration management". I'll be telling you about some research work -- the Vesta project -- that addresses this problem and demonstrates the practicality of some novel solutions.

Outline

- The Software Configuration Management problem
- Vesta's approach to a solution
- Key technical components of Vesta
- Summary

This is the outline for the talk.

I'll briefly explain what the software configuration management problem is and why it's worthy of research effort, then I'll discuss Vesta's approach to the problem. Next, I'll talk about the key technical components of the Vesta solution and conclude by explaining where the project is at present.

I'm also prepared to demo some of the facilities.

What is Software Configuration Management?

- An ill-defined term that covers many aspects of a software development environment:
  - ❋ version management (1...2...3)
  - ❋ source control (checkout/checkin)
  - ❋ configuration management (what goes with what)
  - ❋ building (source files => derived files)
  - – life-cycle management (bug tracking, metrics)
  - – process management (who does what when)
  - – specific tools (front-end design, documentation, analysis, testing, etc.)
  - – . . .

3 digital Systems Research Center

Since we're going to talk about Software Configuration Management, we'd better define what we're discussing. Well, that's not so easy.

<explain points on the slide>

Asterisks denote Vesta's area of emphasis, about which I'll say more a little later. <These are the essential points; others are ancillary to the core of configuration management, as I hope will become clear.>

Each of these aspects has undergone considerable development over the years, but generally in isolation or only limited cooperation with the others. Each is rife with its own concepts, jargon, and techniques. Recent attempts to bring these together by spreading system glue over it all have been relatively unsuccessful.

Rather than getting caught up in the particulars of these individual problem areas, let's take a step back and begin by discussing what problem we are actually trying to solve.

The SCM Problem

- **Question: When do you need SCM?**
  Answer: If your system has
  - **more than one file, or**
  - **more than one developer, or**
  - **more than one customer (or perhaps target platform).**

➨ **The more files, developers, or customers you have, the bigger your configuration management problem is.**

4 digital Systems Research Center

The problem is the construction of a software system. We can quibble about when "construction" starts, but for the purposes of today's discussion, let's agree that it starts when you begin to write code. (I think you'll see that earlier stages, e.g., specification writing, will fit into the organization I will describe. It's just a little easier to start out by thinking primarily about managing code.) When coding starts, you obviously need basic development tools -- an editor, compiler, linker, debugger -- and a file system to store things in. Do you need anything else?

My answer: nearly always, yes.

If you are writing a small, single file application for yourself, you probably don't need SCM. And, with today's customer-oriented PC-based programming systems (like Visual Basic), this happens occasionally. But nearly everything else needs SCM to some degree or other.

Of course, you don't need a sledge-hammer to kill an ant. Some quite 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": they acquire more developers, more code, and more complexity. They outgrow the simple tools.

Let's consider what good SCM does for you when this happens. <next slide>

How Good SCM Helps

- **Multiple files: what goes with what?**
  - **version management of related groups of files**
- **Multiple developers: who is modifying what?**
  - **change management of source files**
- **Multiple customers: who is using what?**
  - **change management of derived files**

5 digital Systems Research Center

Multiple files: the observation that you 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. Good SCM keeps track of true configurations of files.

Multiple developers: the "integration group" story. If you are old enough to remember the days of punch cards and batch processing, you have fond memories of submitting a deck and getting back the results (or lack of results) of the run. Modern large-system integration has reinvented that approach, since developers submit their changes to the integration group which builds the system overnight. Why is this? Because it is impractical for an individual to build and test a version of the system containing only his changes. It's too hard to specify the environment, or the system can't be built in isolation from other developers, or both. So, it's back to the punched card days. But it's even worse now, because the success or failure of the run depends not just on your submission, but everyone else's as well. Good SCM lets individual developers build as much of the system as they need in order to test their component.

Multiple customers: Versions on multiple platforms are attractive because they broaden the market penetration. Versions on the same platform may appeal to different market segments (e.g., Corel Draw, which now sells version 3, 4, and 5 as distinct products). Good SCM supports branching and parallel development.

Why Is Good SCM Hard?

- **Scale! Software keeps getting bigger and more complicated.**
- **Bigger systems driven by (perceived) functional demands.**
- **Parallel development driven by time-to-market.**
- **More layers that must be built consistently.**
- **Multi-target (hardware and OS) systems make configurations more complex.**
- **Organizations that build large systems are often geographically dispersed.**

6 digital Systems Research Center

OK, what are the technical challenges? What must an SCM system cope with, in real-world development?

Well, systems just keep getting bigger. Vendors add functionality based on their perception of what their customers want and what they expect/fear their competitors will provide. They also want customers to "subscribe" to yearly releases. One need only look at the long lists of "checkoff" features on word-processors or spreadsheets or other mass-market software to see this effect.

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.

More system layers have to be built consistently, meaning that 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 (E.g., Microsoft).

To reach the most customers, vendors must make their software run on multiple platforms, which complicates their SCM needs.

Big systems are increasingly developed at multiple sites. Because of wide-area bandwidth limitations, a central site storing all the files is impractical. Replication is necessary, and with it comes the risk of inconsistencies. Good SCM deals with this problem.

What Has Been Tried?

- **Most approaches have been "bottom up"**
  - **address only part of the problem**
  - **achieve only a "local optimum"**
- **"Top-down" approaches have been idealized**
  - **fail to deal with real-world code**
  - **fail to scale**

7 digital Systems Research Center

This all seems pretty obvious, right? So why haven't tools been created that do all this good stuff? It seems quite surprising that software development organizations put up with the lack of good SCM, which contributes directly to their development. I think there are two reasons.

First, I think many organizations actually haven't recognized how central SCM is to the development of large systems. They tend to treat their editors, compilers, etc., as the core tools and SCM as ancillary. I believe the opposite is true: SCM must lie at the core; it must *manage* the process of software construction.

Second, it's hard to build SCM tools that deal with all of these issues for systems of realistic size. Moreover, you can't get there by taking the "bottom-up" approach, which continues to put SCM on the periphery, writing tools that nibble away at one aspect of the problem, e.g., version management, in isolation. 20 years of unsatisfactory results have shown us that bottom-up solutions aren't solutions at all.

Some people *have* understood the central importance of configuration management and instead have adopted a "top-down" approach, trying to organize a development environment around the problem of building software, not compiling and linking. The approaches to date have often been appealing, but regrettably impractical, with only modest demonstration examples where implementations exist at all.

You've probably figured out where I'm leading with this polemic...

Outline

- The Software Configuration Management problem
- **Vesta's approach to a solution**
- Key technical components of Vesta
- Status of the project

Here's where we are.

Vesta's Goals

- **Solve the central problem of software configuration management**
  - **version management (1...2...3)**
  - **source control (checkout/checkin)**
  - **configuration description (what goes with what)**
  - **building (source files => derived files)**
- **Provide technical base for solving other SCM problems**
  - **life-cycle management, etc.**

9 digital Systems Research Center

Vesta takes the top-down approach and concentrates on the four topics that I mentioned earlier, while providing the infrastructure for tools that address the others. Moreover, it implements this approach on a scale that is large enough to be used in the real world. Let me say briefly what I mean by these topics.

* version management: chiefly a naming issue, but not entirely trivial because of derived files (e.g., many can be produced from the same source by varying compilation switches) and sharing between users.

* source control: keeping track of versions, etc., includes keeping multiple users out of each other's hair by managing the name space and ensuring that there's no confusion over source and derived files.

* configuration management: how files are aggregated into useful larger units, and how the aggregation relates to source control and version management, including the management of derived files.

* building: describing how derived files are produced from sources.

Vesta's Goals

- **Do comprehensive SCM for real-world systems; design center:**
  - **20 million source-code lines**
  - **about 5 million derived files**
  - **about 125 GB of disk**
- **Must work well for smaller systems too!**
  - **more typical systems: 50,000–500,000 lines**
- **Portable and industrial-strength implementation**

10 Systems Research Center

As I said, to be successful, these problems have to be addressed in a way that works on real-world software. That is, it can't be a toy. Ideally, a comprehensive SCM system would handle development projects of essentially any size. To make that quantitative, we established some specific goals for Vesta's SCM approach, as shown here. <cover points on slide>

To put these numbers in perspective, the design center of Vesta is appropriate for large operating systems such as OSF1 and VMS, a full-featured relational database system, or a telephone switch. There may be development projects that require larger code bases than these, but I'm not aware of them.

However, there aren't a lot of systems of that size. Vesta must work comfortably on a smaller scale as well. That means not too much mechanism, boilerplate, user overhead, etc.

Is This Feasible?

- **Yes! We built a prototype and used it for over a year.**
  - **25 programmers (at one site)**
  - **Code base of 1.4 million source lines**
  - **10 GB of disk for source and derived files**
- **Reliable, incremental development at all system levels**
  - **integration and system test vastly strengthened**

11 digital Systems Research Center

Solving this collection of problems sounds like it might be challenging. Instead of keeping you in suspense, I'll give you a quick summary of some "hard results" and say a little more about them at the end of the talk.

We developed an approach to SCM that I'll begin to describe in a minute. To test the feasibility of our approach, we built a rather complete prototype SCM system and used it in earnest for about 15 months. 25 programmers developing a complex development environment used it for their daily work. They were building an operating system, a file server, a novel compiling system, an experimental windowing system, a number of libraries, and a spectrum of applications, including a text editor, a graphical editor, ray-tracing software, and some mathematical analysis tools. To add to the complication, most of these pieces were being built with the constantly evolving compiling system, and were targeted for two different platforms: a VAX multiprocessor with the custom operating system, and a MIPS uniprocessor running OSF1. In short, it was a *mess!*

The good news: it worked. Of course, it was a prototype and had rough edges and limitations, but it conclusively demonstrated that a comprehensive approach to SCM could work effectively on real-world development problems. Moreover, it had a qualitative effect on the style of development <cover the second point on slide>.

Enough advertising. Let me tell you how we approached the problem.

The Vesta Axiom

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

- **system descriptions are source-based;**
- **they include source files and building instructions;**
- **they are self-contained;**
  - » **no environmental dependencies**
- **they are immutable and immortal;**
  - » **same meaning forever**
  - » **nothing is ever lost**
  - » **reproducible builds**
- **large descriptions can be composed from small ones.**

12 Systems Research Center

I'll start by stating the "Vesta axiom", that is, the premise that underlies all of the key design choices made in the Vesta system. I like to call this an axiom because (a) axioms are supposed to be self-evidently good, and (b) you must not do anything in your system to make the negation of the axiom true, or your system will be unsound. <read slide>

Source/derived distinction: hand-made vs. machine-made.

Source-based description means that the description explains how to build a system from scratch. In principle, every system construction is done from scratch. To be practical, of course, it is essential to reuse previous results (derived files), although formally these are just optimizations.

Complete descriptions tell the whole story, capturing every relevant detail of the environment. Complete means *complete*; every version of every tool and every switch is specified. Moreover, these details are in terms of source, so the tools are specified giving the system description that constructs them, and so on, back to the Big Bang (well, in theory at least).

So, I hope it's clear that a system with these properties is a Good Thing. That's why this is an axiom. Before I tell you what we derived from this axiom, I'd like to spend a couple of minutes considering alternative axioms, for those who might disagree that this one is self-evidently good. (Or, in the vernacular, "different strokes for different folks.")

Vesta Choices

- **Versions, not views**
- **Files and directories, not databases**
- **Self-contained system-building descriptions, not templates, rules, search paths and the like**

Not everyone finds the Vesta axiom self-evident or even appealing. People have different development styles, and some of them assume environments quite antithetical to Vesta's. I'll point out some possibilities:

* Some people don't like versioning to be too visible. They just want a simple name that points to "the right" version and is rebound over time as appropriate; that is, a "view" into the versioned name space. With views, the meaning of a name changes over time. When and how does it change? Well, generally the answer to that question is quite complicated. (This is the usual Unix way. <Give example of dependency on other people's stuff.>)

* Some people like data bases. There's lots of stuff related to keeping track of software development: bug reports, project plans, budgets, etc., for which databases are a more natural medium. I like databases too -- some of my favorite systems are databases -- but they aren't well-suited to keeping an exhaustive historical trace in which one can easily return to any previous state. They're designed to move forward in atomic steps of varying size, always presenting a single, consistent, current state. Also, databases aren't ideal for storing large, unstructured things like source files, so an auxiliary file system is used. Hmm, sounds like Vesta. Auxiliary DBs, however, are good.

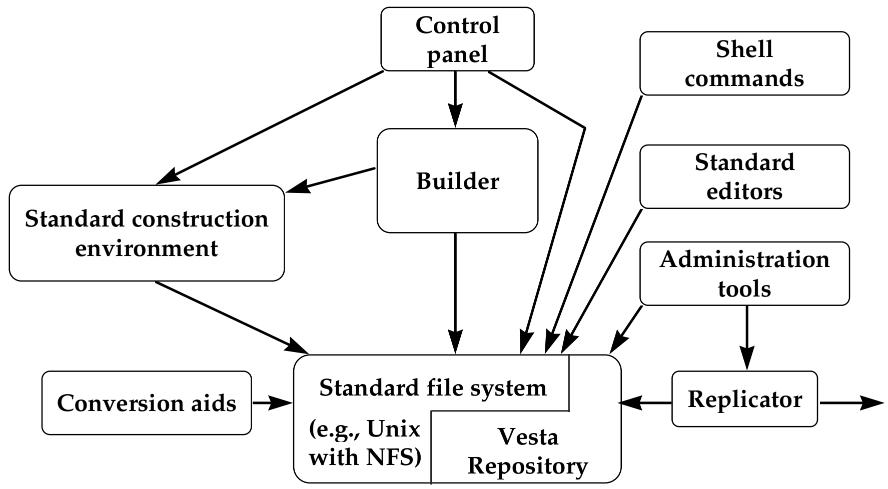
* The choice is between explicit complete descriptions and templates whose actions are altered by the environment around them. (Latter is bad; irreproducible. Macros processors are delicate: look at 'make')

Outline

- The Software Configuration Management problem
- Vesta’s approach to a solution
- **Key technical components of Vesta**
- Status of the project

Here’s where we are.

A User's View of Vesta



15 digital Systems Research Center

Here are the pieces of the Vesta system from the perspective of a user. At this point, I want to introduce you to these pieces and give you a sense of how they fit together.

Let's start at the bottom. Clearly we need a place to store things: the source files and descriptions 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. We'll see more about this shortly.

We also need an engine for the construction process. This is the interpreter of system descriptions, which we also call the builder. It needs to call on particular tools to produce derived files; these tools are compilers, linkers, macro processors, etc. We call them bridges; they are part of the standard construction environment.

Finally, we need a way to communicate with this stuff. The control panel and ordinary shell commands provide these facilities. The control panel isn't really central -- you can use shell commands instead -- but it makes some common operations more convenient.

I'll say more about the remaining pieces later.

A Simple Scenario

Sally is responsible for a standard utility program that sorts the lines of a file. In the shower one morning, she thinks of a clever optimization that will speed it up.

Now that we've seen the pieces, let's look at a simple scenario to get a little bit of intuition about how they work. Then we'll come back and discuss them in more detail.

A Simple Scenario

- **Upon arriving at the office, Sally uses the Vesta control panel's Checkout button to check out the sorter package.**
  - ❖ **A *package family* is a directory stored in the Vesta repository. The repository is just a directory tree; let's call it `/vesta/pa.dec.com/`**
  - ❖ **The directories under the package family directory are *package versions*. They are immutable.**
  - ❖ **A package version directory contains source files only.**
  - ❖ ***Checking out* a package reserves a new version number and establishes two other directories, a *session directory* and a *working directory*.**

The choice of repository name is a convention; putting the site name in the path makes replication easy later. I won't have time to go into that now, but can chat with anyone who is interested offline.

Contrast with conventional file-based checkout, in which individual files are checked out to a directory of the user's choice. Vesta versions whole directories and extends that versioning right down to each run of the builder. In fact, the builder can only process immutable sources, as we shall see.

The reserved directory created by Checkout is called a stub. It is a placeholder for an immutable directory that will be created later.

A Simple Scenario

- **By checking out the package, Sally reserves a specific version number. (Actually, it's the name of the directory she will eventually check in.)**
  - ❖ **In Vesta, directories, not files, are versioned.**
  - ❖ **Let's call the reserved directory `/vesta/pa.dec.com/sorter/5`**
  - ❖ **The working directory is where Sally edits the files. It is `/vesta-work/sally/sorter` and initially contains files identical to those in `/vesta/pa.dec.com/sorter/4`**
  - ❖ **The session directory is where intermediate versions appear as Sally tries out her changes. It is `/vesta/pa.dec.com/sorter/checkout/5`**

The versions of interest to others show up in the main repository, while those created during checkout sessions (in the /checkout/ directory) are generally not interesting to anyone but the developer. (This isn't always true -- developer A may want developer B to try something out before it "goes public".)

The working directory has conventional Unix semantics, so any tool can be used to manipulate files here. The directory is implemented by the repository, which enables us to add both performance and functionality.

A Simple Scenario

- **Sally clicks the Edit button on the Vesta control panel, which runs her favorite editor on a selected file in the working directory.**
  - ❖ **Let's call it `sorter.c` (in `/vesta-work/sally/sorter`)**
- **After finishing her changes, she clicks Build on the Vesta control panel.**
  - ❖ **Vesta creates a snapshot of the working directory as a new immutable directory under the session directory. It is /vesta/pa.dec.com/sorter/checkout/5/1**
  - ❖ **Vesta attempts to build the package in this directory, using instructions in the file build.ves.**

digital Systems Research Center

Alternatively, she could use cd <working directory> and run the editor from a shell. We'll see that in the demo later.

The editor doesn't know anything about Vesta. It is just editing an apparently ordinary file in a working directory.

The snapshot exploits the fact that the working directory and session directory are both implemented by the repository code. This makes the snapshot very efficient, and much faster than it would be if it were a conventional Unix directory. (Details on request.)

The builder can only work on immutable files (per the Vesta axiom). That's why the snapshot is made. However, it's often handy to have versioning during a development session -- e.g., to back out of something that turned out to be a mistake.

A Simple Scenario

`sorter/build.ves` **looks like this:**

```
files
  hs = [ sorter.h ];
  cs = [ sorter.c, sortersub.c ];
{
  name = "sorter";
  libs = [ ./C/libc, ./C/libX11 ];
  return
    ./C/Program( name, hs, cs, libs );
}
```


digital Systems Research Center

This build description says

"Build a C program by compiling and linking the two source files sorter.c and sortersub.c with the libraries libc and libX11. The compiling environment is to include the file sorter.h plus all the header files associated with libc and libX11. Moreover, the C tools (compiler and linker) plus the libraries (both implementations and header files) are to be taken from an environment that is an implicit parameter to this construction -- it is named '.' The program produced by all of this is to be called 'sorter'. "

Now I don't expect you to infer all that from what's written on the page. It reflects some assumptions about the standard construction environment, especially the way libraries are organized. I'll have more to say about that later.

When Sally builds sorter/checkout/5/1/build.ves, it is likely that only sorter.c will be compiled. The builder is incremental and Sally has only modified sorter.c. The result of compiling sortersub.c at some previous time is likely to be cached. Vesta manages this cache automatically; I'll say more about this later. Sally doesn't have to be concerned about the location or name of this file (sortersub.o) -- that's Vesta's problem.

A Simple Scenario

- **Sally tries out the version of sorter she has just built and discovers it isn't quite right. She edits `sortersub.c` then clicks Build again.**
  - ❖ **This produces a new version in the session directory: `/vesta/pa.dec.com/sorter/checkout/5/2`**
- **This time it works, and Sally clicks Checkin to check in the result.**
  - ❖ **As a result, the originally checked out directory: `/vesta/pa.dec.com/sorter/5` now has the same contents as `/vesta/pa.dec.com/sorter/checkout/5/2`**

21 digital Systems Research Center

The second build is also incremental and only sortersub.c is compiled.

The pieces stay around in checkout/5 even after the checkin is completed. This can be useful, but eventually it causes clutter. There's a way to deal with the clutter without compromising the Vesta axiom; I'll be happy to talk about it offline if you're interested. (This is the conversion of a directory to a ghost, and the issue of whether such a directory's name should be visible or not.)

OK, that's the end of the scenario introducing some of the Vesta components. Let's now look at them in more detail.

Key Vesta Facilities

- Repository (storage system)
  - immutability
  - replication
- Builder (language interpreter)
  - incremental construction from immutable sources
- Standard construction environment
  - compilers, linkers, etc.
  - libraries and their interfaces
- Specialty tools
  - administration
  - Vesta-specific utilities (e.g., makefile translators)

As we've seen, the repository is mostly seen as a slightly peculiar part of the file system. It supports creation of immutable directories with the snapshot and checkin mechanisms. Replication is used to move files between sites (often sites that have limited connectivity).

The builder is pretty much as I described -- the construction engine. From the user's perspective, its key property is incrementality; that is, it builds only what's necessary, no more, and certainly no less.

The standard construction environment provides the files you expect to find in standard directories, such as /usr/include, /bin, etc. More precisely (and more interesting), the standard construction environment consists of a collection of system descriptions that build these directories. These descriptions are heavily parameterized, so it's easy to build environments customized for a particular purpose, e.g., debugging or testing of new versions of library components.

A provocative, and essentially correct way to think about the standard construction environment system descriptions is that they produce, every time you build, an entire file system customized for the build at hand. The customization extends to compilers, their parameters, libraries, the works. Of course, this environment is built incrementally, and generally it's the same as it was the last time, so there's no work to do. But the opportunity exists, on every build, to produce precisely what's needed.

Finally, there are some specialized tools that are peculiar to the maintenance of the Vesta environment or the migration of development to/from it.

Repository

- **File system extension (leverages standard utilities)**
  - **Primary interface to repository is through file system operations (but not all operations can be used everywhere).**
  - **Supplementary interface for tools and operations that don't fit the usual file system mold.**
- **Each site has a collection of directory trees (of source files) plus a derived file pool.**
- **Three kinds of directories**
  - **immutable (e.g., a package version)**
  - **appendable (e.g., package family, session directory)**
  - **mutable (a working directory)**

23 digital Systems Research Center

By "file system extension" I mean that the repository provides access to the files it stores through the ordinary file system interface, thus making them available to ordinary applications (like the editor) that don't know about Vesta. As we've seen, versions are encoded in ordinary file names and remain visible to user (contrast with a view-oriented scheme, e.g., ClearCase or an explicit separate name-space scheme, e.g., RCS).

Not all repository facilities can be conveniently made available under the file system interface, so a secondary interface is provided for tools that exploit those facilities. That's how Checkout and Checkin work.

A site is the unit of administration, e.g., SRC. Within a site, sources and deriveds are segregated. Derived storage is site-wide. Only the results of the build (like the sorter program) tend to be interesting and are typically made visible. <Defer discussing how.>

An immutable directory holds a version of a package. As we've seen, the version may be archival, or may be a snapshot of a state during a development session. Either way, immutability applies.

Since immutable directories are created all at once, we need some way to create incrementally the files that will make up an immutable directory. As we've seen, this is the thing (indeed, the only thing) mutable directories are used for.

Appendable directories can include other appendable directories. For example, both .../sorter and .../sorter/checkout are appendable.

Repository

- **Directories and files have an open-ended set of (mutable) attributes**
  - **not used by the builder; system descriptions can only reference immutable directories**
  - **hook for various higher-level tools, such as specialized editors for system descriptions.**
- **A directory can be replicated for wide-area access**
  - **partial replication is possible.**
  - **directory's "master" site synchronizes changes.**
  - **time-grain of replication is configurable (e.g., on demand, siphon, the postal service).**

24 digital Systems Research Center

To support various kinds of tools, the repository permits labeling of files and directories with attributes, which are mutable. (Example of uses of attributes: link working directory to session directory; identify buggy versions post facto so that model-revision tools will skip over them. Maybe replication too.) These aren't used for building, so mutability doesn't compromise reproducible construction.

Vesta expects that the bandwidth available between sites will typically be significantly lower than that within a site, so that cross-site references are unappealing. Availability may be a concern as well. Because cross-site reference is unappealing, Vesta supports selective copying between sites. Partial replication is possible; unreplicated names are "stubbed". Selective replication is often desirable because one site uses the end-products of another; intermediate derived files need not be replicated. Of course, this improves latency of replication too.

Builder

- **Reproducible, incremental construction**
  - **eliminates “the nightly build”**
  - **permits individual developers to do real testing**
- **Performance competitive with ‘make’**
  - **Individual dependency tests must be cheap!**
  - **Must avoid “going all the way to the leaves”**
- **Users benefit from each others’ builds**
  - **When building a component for a user, must reuse work performed on behalf of others.**

25 digital Systems Research Center

Anyone can do incremental construction -- look at ‘make’. The trick is to get the right answer. ‘make’ doesn’t cut it for large systems, which is why large groups abandon incremental construction. Leads to lack of individual testing, long turnaround cycles with many trivial integration bugs, etc.

Conventional ‘make’ fails to be adequate for large systems for several reasons. First, it depends on a notion of “the current version”, which fails when parallel development is needed. Second, its notion of dependency is incomplete, even when extended by dependency generators (i.e., files only). Third, dependency checking is expensive, requiring a file system operation. Fourth, there is no notion of hierarchy, which permits dependency checking at the component level rather than the file level. <explain these in a bit more detail> All these deficiencies prevent *reliable, incremental* construction.

I should point out that the picture isn’t quite as black as I’ve painted it. The best modern derivatives of ‘make’ (that is, systems that retain ‘make’ syntax and (mostly) semantics) can’t fix all of these problems. ClearCase comes closest, but still has fundamental performance problems and no notion of hierarchical description.

Implications

- ‘make’ and its descendants are inherently limited in scale.
- A different way to describe configurations is necessary.

26 digital Systems Research Center

The inescapable conclusion is that ‘make’ is too limited for large-scale construction. It just doesn’t have a scaleable way of describing systems that permits reliable incremental construction. A better form of description is necessary. Let’s consider what characteristics this new form of description should have. <next slide.>

System Descriptions

- Structure of system descriptions is affected by many considerations, e.g.
  - Size and scope of system being described
  - Structure of the organization building it
  - Methodology of the organization
- Vesta must therefore support a broad spectrum of descriptions.
  - Start with a foundational language with fairly generic facilities intended to support extension.
  - Supply at least one fairly comprehensive extension.
  - Extension can be tailored to organizational requirements (or entirely replaced).

27 Systems Research Center

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/how it is run; the relationship of documentation to the code it references, etc.

=> A bad idea to wire this stuff in the description language.

=> Instead, we put in generic facilities as the basis, intended to be extended in an organization-specific way rather than used raw. (Familiar parallel: Emacs; <explain>).

=> Of course, to be usable, the system must provide at least one fairly comprehensive extension as a starting point for everyone (and a reasonable ending point for many). This also ensures that the base facilities are equal to the task.

By the way, the challenges in creating robust extensions of this sort are well-known, and the Vesta context shares most of them with other environments. For example, we must work out ways to provide decent error reporting at a suitable level of abstraction, and to avoid burdening the user with a steep learning curve that makes small enhancements of the standard extension expensive.

Description Language

- **Functional programming language**
  - **C-like syntax**
  - **firm semantic foundation (unlike 'make')**
- **Strong typing, dynamically checked**
- **Modular**
  - **can separate environment and package descriptions**
- **Methodology-neutral**
  - **operations peculiar to software construction aren't wired-in:**
    - **structure of libraries, applications, releases**
    - **specific building actions, e.g., compilation**
    - **programming-language independent**

28 digital Systems Research Center

So, what is the base language? It's a functional programming language, which matches the incremental building approach well, since one can easily (in principle) cache results at the function application level.

The language is in the spirit of LISP, with lambda calculus as the semantic underpinnings, but with a more familiar, C-like syntax. Functions (closures) are first-class entities in the language. The language has "readable" scoping (no dynamic binding). Values are strongly typed, but checking is not static. There's no point in static checking, since "runtime" is construction time of the program being described.

Language emphasizes manipulation of composite values, typically lists or bindings (sets of <name, value> pairs). Bindings are essentially cheaply-constructed directories, and Vesta exploits the similarity by making it easy and inexpensive to pass a binding to a tool (e.g., a compiler) and fool it into treating it as a directory .

Modular descriptions allow packages to be described independently and clearly 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. Contrast this approach with typical 'make' usage, in which many references to the environment are implicit and difficult to spot by reading the descriptions.

In fact, the basic language has very little that is specific to system-building. <elaborate on final three bullets>

The Standard Environment

- **"Simple things should be simple, complex things should be possible."**
- **Provides a complete, versioned description of a consistent set of tools, libraries, etc.**
  - **described by a system model**
  - **builds a (large) directory tree that is like a consistent, customized snapshot of the relevant parts of a file system**
- **An extension built atop the generic description language.**
  - **provides functionality for many common cases**
  - **can be modified by users for unanticipated situations**

29 Systems Research Center

Let's turn now from the base description 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". To quote a familiar maxim, <first bullet>.

Specifically, the extension enables the user to describe ordinary configurations by lists of components (source files, libraries, etc.). There are sensible defaults for compiling sources and combining the results in libraries, or executable programs, or whatever. The defaults can be selectively overridden with compact, intuitive notation and there are selected "escape hatches" where a user-supplied function, written in the description language, can be inserted. If all of that proves inadequate, the entire extension is written in the description language, so it's just a bunch of system models, and can be modified or replaced by a more demanding customer.

Well, complex things should be possible, yes, but not necessarily *every* complex thing. At some point, the user has to modify the extension rather than working within it. (Emacs analog, again.) Where that point is is a question of organizational sensitivity and sophistication. We've tried to provide a sufficiently comprehensive extension that most organizations can go pretty far before they have to modify it.

Standard Environment

- Building for multiple platforms, either natively or with cross-development tools
- Building in parallel on multiple machines
- Building a personalized version of a system
  - with selected libraries replaced
  - with selected components built with a special compiler/linker or with unusual options
- Automated testing, including regression, as part of the build

30 digital Systems Research Center

Here are some examples of the kinds of activities supported by the standard environment. We used most of these in the prototype system, in some cases, extensively.

The Standard Environment

- **Language-specific tools grouped as "bridges"**
  - **Bridge construction for ordinary languages is very simple (written entirely in Vesta description language)**
  - **"Wrapper functions" (e.g., `./C/Program`) are largely platform-independent**
- **Standard models provide templates for common situations**
  - **build a library, build an application, etc.**
  - **build a hierarchy of libraries**
  - **common overrides (e.g., compiler switches) easily expressed**

31

digital Systems Research Center

Returning to the needs of the average user, I want to say a little bit more about the way the standard extension provides access to tools like compilers and linkers.

We use system models to produce versioned snapshots of the compilers and other tools, the library interfaces and implementations, related documentation, etc. Each such snapshot represents an immutable, consistent version of the construction environment.

Major tools are packaged as language-specific bridges. The basic functions (e.g., compile, link) are available in the language, but most users will prefer to use "wrappers" that conveniently bundle frequently used functionality (e.g., build a program from these sources and libraries). The definitions of wrappers, being in the environment, don't clutter the user's system descriptions, which now reduce largely to named lists of files.

We saw an example of a wrapper in the earlier scenario: ./C/Program. Because of defaulting, we didn't see the full power of the wrapper. The wrapper permits each member of the source module list to be tagged with optional compiler switches, and has a defaultable parameter with loader switches. The library names refer by default to the environmental snapshot, but can be overridden if desired.

Properly constructed wrappers give functionality that is largely platform-independent.

Specialty Tools

- **Control panel**
  - **a GUI for top-level model construction**
  - **conveniently separates package definition from environment of construction, without sacrificing completeness**
- **Administration**
  - **repository embedding in file system**
  - **server location and loading**
  - **“siphon” (background replicator)**
  - **“weeder” (like a garbage collector)**

32 Systems Research Center

Control panel: Vesta philosophy is that all construction occurs by interpreting a system model. The control panel provides a GUI for automating the construction of certain uninteresting models, for example, the model that says “evaluate the most recent version of my package in this version of the standard environment”. There are only two pieces of information here; the latter changes rarely, and the former is easy to compute. So, the model can be generated trivially upon request.

Administration: I won’t say much about this, except to observe that the user convenience of an immutable file system imposes additional administrative burdens, so some extra tools are necessary to assist the administrator. <explain items on slide>

Specialty Tools

- **Interoperation with 'make' to support gradual conversion**
  - **interpreter**
  - **translator**

33

digital Systems Research Center

There's one other specialty tool that's worth mentioning today. Since Vesta doesn't use the same description language as 'make' (for all the reasons I gave earlier), there is an "entry barrier" for users who wish to move an existing system that uses 'make' into Vesta. We provide two tools to make this easier. Our goal is to permit an existing system to be converted a piece at a time, with some makefiles being rewritten as Vesta descriptions, and others remaining unchanged for a while.

We provide an interpreter for makefiles to deal with the parts of the system that aren't going to be converted right away. We also provide a translation assistant that helps convert makefiles to Vesta descriptions. Regrettably, because make files have relatively little useful static structure, we can't translate them automatically to intelligible Vesta models. However, we can make a running start at it, doing the first 80% automatically and leaving only a small amount for the user to do by hand.

If they choose, users can continue to use makefiles indefinitely, relying on the Vesta interpreter. Of course, they won't get the full benefits of Vesta; in particular, they get reliable, but not very incremental building. And, for parts of the system that are shared with organizations that don't use Vesta, it might be handy to retain descriptions as makefiles.

Summary

- No comprehensive SCM solution is presently available commercially for large systems.
- The Vesta technology supports a comprehensive solution.
- The Vesta prototype proves its feasibility.
- A demonstration system will be running this summer.
- Publications available (based on the Vesta prototype):
  - SRC Research Reports 105–108
    http://www.research.digital.com/SRC/publications/src-rr.html

So, to summarize: <briefly cover points on the slide>